Extending Oracle Commerce

F48913-01
January 2022

ORACLE

Extending Oracle Commerce,

F48913-01

Copyright © 1997, 2022, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government’s use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

1 Understand Extension Features

2 Use the REST APIs

Learn about the APIs 2-1
REST API authentication 2-2
Use the APIs on instances running multiple sites 2-6
CORS support 2-6
REST API query parameters 2-8
Response filters 2-12
Error messages 2-15
Register applications 2-16

3 Use Webhooks

Understand webhooks 3-1
Configure webhooks 3-4
Secure webhooks 3-5
Troubleshoot webhooks 3-7
Understand webhooks and PCI DSS compliance 3-7
Use the REST API to configure webhooks 3-9
Reduce the size of webhook requests 3-11
Manage failed webhook calls 3-15
Return webhook errors in an endpoint response 3-19
Summary of webhooks involved in order submission 3-20

4 Develop Server-Side Extensions

Access Node.js servers 4-1
Configure an SSE 4-2
Implement an SSE 4-5
Upload an SSE 4-7

Validate SSEs 4-7

5 Manage Shopper Profiles

Understand shopper profiles and shopper types 5-1
View a shopper profile 5-1
Create a shopper profile 5-3
View a shopper type 5-10
Add custom properties to a shopper type 5-12
Set custom properties on a shopper profile 5-14
Create custom properties for addresses 5-15
Access custom properties using the UserViewModel 5-18

6 Access SKU Properties through Widgets

Understand APIs for accessing SKU properties 6-1
Create an element to display SKU properties 6-3
SkuPropertiesHandler example 6-5

7 Create Custom Promotions

Understand PMDL discount rules 7-1
Create a promotion 7-4
View promotions created with the REST API 7-6
Create custom promotions for shared carts 7-7
Sample promotions 7-11
PMDL Example 7-20
Create custom properties for promotions 7-27
Assign and manage coupons 7-33
Set up promotion upsell messages 7-35

8 Manage Multiple Inventory Locations

Access inventory data 8-1
Create locations 8-4
Create inventory data for locations 8-7
Retrieve inventory data for locations 8-9

9 Manage Inventory for Preorders and Backorders

Understand inventory 9-1
Enable preorder and backorder functionality 9-1

Access and update inventory data 9-2
Update widgets for preorders and backorders 9-3
Customize email templates for preorders and backorders 9-3

10 Manage Orders

Integrate with an order management system 10-1
Understand order states 10-11
Create custom properties for orders 10-14
Implement robust order capture 10-17
Support zero-cost orders 10-18
Support shopper-initiated order management 10-19
Enable returns on partially fulfilled orders 10-23
Support add-on products 10-24

11 Customize Order Line Items

Understand customization of order line items 11-1
Create custom properties for line items 11-2
Understand view model support for line items 11-4
Implement a custom cart summary widget 11-5

12 Ship an Order to Multiple Addresses

Understand view model support for split shipping 12-1
Implement split shipping UI controls 12-3
Understand REST support for split shipping 12-16
Customize email templates for split shipping 12-17
Retaining shipping group information 12-22
Extending the CartItem and ShippingGroupRelationship view models 12-23

13 Exclude Items from Shipping Methods and Costs

Exclude items from shipping methods 13-1
Exclude items from shipping cost calculations 13-1
Create collections for the excluded items 13-2
Update shipping methods 13-2
Update the Order Summary – Checkout widget 13-3

14 Manage Countries and Regions for Shipping and Billing Addresses

Understand countries and regions 14-1
Retrieve a list of countries and regions 14-2
Create and update countries and regions 14-3
Delete countries and regions 14-6
Customize address formats using the API 14-6
Work with address types 14-15
Customize address validation 14-17

15 Configure Buy Online Pick Up In Store

Understand buy online pick up in store 15-1
Manage inventory for in-store pick up 15-3
Configure layouts and widgets for in-store pick up 15-4
Configure products and SKUs for in-store pick up 15-7
Customize email templates for in-store pick up 15-7
Configure payment processing for in-store pick up 15-8
Understand tax processing and in-store pick up 15-10
Configure the Picked Up Items webhook 15-11

16 Create Scheduled Orders

Configure an invoice payment gateway for scheduled orders 16-1
Configure the scheduled order service 16-2
Configure page layouts for scheduled orders 16-4
Update prices in a scheduled order 16-5
Notify shoppers about scheduled order activity 16-6
Understand shopper tasks for scheduled orders 16-6

17 Notify Shoppers When Items are Back in Stock

Understand back in stock notifications 17-1
Create and upload the notification extension 17-1
Add the Notify Me element to the Product Details widget 17-6
Configure the scheduler to send the back in stock emails 17-6

18 Enable Purchase Lists

Understand the difference between wish lists and purchase lists 18-1
Work with purchase lists 18-3
Work with the purchase list API 18-5

Share purchase lists 18-9

19 Enable Order Approvals

Allow a delegated administrator to control order approvals 19-1
Configure a deferred payment gateway for order approvals 19-1
Set the frequency of canceled order clean up 19-3
Configure page layouts for order approvals 19-4
Manage the checkout flow for orders requiring approval 19-6
Display a contact's purchase limit in a widget 19-12
Integrate with an external system for order approvals 19-12

20 Assign Catalogs and Price Groups to Shoppers

Configure the External Price Group and Catalog webhook 20-1
Create a custom shopper context widget 20-3

21 Implement Storefront Single Sign-On

Understand storefront SSO message flow 21-1
Configure storefront SSO 21-2
Understand storefront SSO limitations 21-6
Implement storefront SSO for account-based shoppers 21-7
Renew your SAML 2.0 SSO certificate 21-12

22 Implement Single Sign-On for Internal Users

Configure SSO with OpenID Connect 22-1
Configure SSO with SAML 2.0 22-4

23 Configure Sites

Understand site objects 23-1
Create a site 23-4
Update a site 23-6
Delete a site 23-8

24 Work with Carts that are Shared Across Multiple Sites

25 Work with Loyalty Programs

Implement loyalty points 25-1
Create a custom currency for loyalty points 25-1
Configure a site to use loyalty programs 25-3
Understand tax and shipping calculations with loyalty programs 25-5
Display tax and shipping in currency for points-based orders 25-6
Redeem loyalty points 25-8
Understand currency exchange rates 25-11
Use custom properties in loyalty integration 25-13

26 Integrate with External Shipping Calculators

Work with externally priced shipping methods 26-1
Upgrading from external shipping methods to externally priced shipping methods 26-5
Work with external shipping methods 26-6
Enable fallback shipping methods 26-10

27 Integrate with an External Pricing System

Create the widget 27-1
Configure the webhook 27-6
Use promotions from an external system 27-7

28 Customize Email Templates

Download and edit email templates 28-1
Customize tax display in templates 28-2
Customize line-item display in templates 28-3
Add company name and logo to account-based email templates 28-5
Notify a contact of multiple account or role changes in a single email 28-6
Customize recommendations in templates 28-7
Add a site to a template 28-11

29 Upload Third-Party Files

Create folders for third-party files 29-1
Upload third-party files to folders 29-2
Upload a Google site ownership verification file 29-6
Upload an Apple Pay merchant identity certificate 29-6
Delete third-party files 29-7

Manage files on multiple sites 29-8

30 Manage Guest Checkout

Example for restricting guest checkout 30-1
Note about preventing self-registration in account-based storefronts 30-3

31 Manage Saved Carts

Understand saved carts 31-1
Create a widget to support saved carts 31-2
Customize emails for saved carts 31-10

32 Manage the Use of Personal Data

Configure consent requests 32-1
Delete shopper information 32-11

33 Implement Access Control for Internal Users

Use and modify roles for internal users 33-1
Create security criteria 33-5
Create generic access rights 33-10
Control access to shopper data 33-12
Understand access control for applications 33-16

34 Manage an Account-based Storefront

Manage account-based shopper profiles 34-1
Implement access control in business accounts 34-3
Create custom properties for accounts 34-8
Add delegated administration to your storefront 34-15
Ensure PayPal shoppers provide first and last name 34-18

35 Integrate With a Procurement System

Understand punchout 35-1
Enable punchout for an account 35-2
Work with the punchout server-side extension 35-7
Configure your storefront for punchout shoppers 35-12

36 Perform Bulk Export and Import

Understand Bulk Exporting And Importing 36-1
Export data endpoints 36-3
Import data endpoints 36-5
Understand export and import endpoint parameters 36-8
Export and import account data 36-10
Export and import profile data 36-14
Export and import product data 36-19
Export and import catalog data 36-23
Export and import category data 36-27
Export and import inventory data 36-29
Export and import promotion data 36-30
Export and import price data 36-32
Import address data 36-34
Import relationship data 36-37
Export and import CSV files 36-39
Delete bulk import or export files from repository 36-50
Convert registered shoppers to account-based shoppers 36-51
Improve performance in large bulk imports 36-56

37 Create a Credit Card Payment Gateway Integration

Understand the credit card payment gateway workflow 37-1
Create a credit card extension 37-1
Install the extension and configure the gateway 37-5
Credit card payment properties 37-6

38 Create a Generic Payment Gateway Integration

Understand the generic payment gateway architecture 38-1
Supported payment methods and transaction types 38-1
Send custom properties to a payment gateway 38-4
Incorporate 3D-Secure support 38-6
Support stored credit cards 38-14

39 Integrate with a Gift Card Payment Gateway

Understand the gift card payment gateway workflow 39-1
Create a gift card extension and configure the webhook 39-1
Customize the Gift Card widget 39-3

Gift card payment properties 39-4

40 Integrate with a Store Credit Payment Gateway

Create a store credit extension and configure the webhook 40-1
Add a Store Credit payment option to the checkout page 40-3
Store credit payment properties 40-3

41 Integrate with a Loyalty Point Payment Gateway

Understand the loyalty point payment gateway workflow 41-1
Create a loyalty point extension and configure the webhook 41-1
Add a loyalty point payment option to the checkout page 41-3
Loyalty point payment properties 41-3
Use Loyalty Points and Pay with alternate currency 41-16

42 Integrate with a Cash Payment Gateway

Understand the cash payment gateway workflow 42-1
Create a cash payment extension and configure the webhook 42-1
Cash payment properties 42-2

43 Integrate with an Invoice Payment Gateway

Understand the invoice payment gateway workflow 43-1
Create an invoice payment extension and modify the checkout page 43-1
Invoice payment properties 43-3

44 Integrate with a Web Checkout System

Overview of web checkout system integrations 44-1
Initiate the order 44-1
Retrieve the order 44-4
Complete the order 44-6

45 Enable Split Payments

Understand split payments 45-1
Use the Split Payment widget 45-2
Use webhooks with split payments 45-3
Customize the Split Payment widget 45-3

46 Configure Tax Processors

Integrate with an external tax processor 46-1
Monitor tax processors 46-14

47 Configure Search Features

Understand which search features can be configured 47-1
Understand how to execute endpoints 47-2
Understand ZIP format and JSON format 47-2
HTTP methods for configuring search features 47-3
Delete resources 47-4
Understand system-generated object attributes 47-5
Export and import all search configuration 47-5
Configure individual resources using ZIP format 47-7
Back up and restore all application configuration 47-7
Migrate configuration of all search features 47-8
Apply configuration changes to your live storefront 47-8
Configure a thesaurus 47-8
Configure keyword redirects 47-12
Optimize URLs for search engines 47-23
View your changes 47-26
Specify which index fields are included in searches 47-26
Index and Query Popular Searches 47-38
Modify data structures to enhance searches and navigation 47-44
Configure which properties of aggregated records and their members are accessible to front end applications 47-50
Configure the order of facets 47-52
Configure the order of facet values 47-54
Order facet values by statistical significance 47-57
Add metadata to facet values 47-60
Create custom range facets 47-62
Configure the ranking of records in search results 47-64
Link additional content to search results 47-77
Search non-catalog data 47-79
Machine learning for search 47-87
Sample Search and Navigation REST API requests using cURL 47-88

48 Use Developer Utilities

Download the Commerce SDK 48-1
Use the Design Code Utility 48-1

Use the JavaScript Code Layering User Interface feature 48-19
Toggle JavaScript minification in preview 48-21
Reduce the size of page responses 48-21
View client-side error logs 48-23
Restore or upgrade the storefront framework version 48-25

49 Improve System Performance

Measure performance often 49-1
Monitor your Commerce environments 49-1
Improve performance in REST API Calls 49-2
Use cc-storage for Safari private browsing mode 49-2
Avoid console.log() statements 49-2
Avoid using ko.observable() 49-2
Update observable JavaScript arrays 49-3
Use Knockout data-binds syntax to attach events to DOM elements 49-3
Use onLoad and beforeAppear correctly 49-3
Use the fields parameter 49-3
Use persistent filters 49-4
Use minified versions of libraries and widget JavaScript 49-4
Localize endpoints 49-4
Enable queueing simultaneous endpoint calls 49-4
Improve performance in custom widgets 49-4
Optimize Search 49-5
Use preFilter parameter with fields parameter to improve endpoint performance 49-5
Speed up system response on Product Listing and Product Details 49-6
Enable asynchronous orders flow 49-7
Improve Storefront Performance for Large Carts 49-8
Prevent Site Traffic Slowdowns 49-14
Improve performance with large numbers of addresses for profiles or accounts 49-15

50 Improve Storefront Performance

Optimize First Meaningful Paint 50-1
Lazy load images 50-1
Improve Storefront Performance for Large Carts 50-3
Add a page level spinner 50-9
Enable prioritized loading of Storefront page content 50-9
Avoid synchronous AJAX calls 50-9
Avoid hiding elements with CSS styling 50-9
Remove unused UI elements completely from layouts 50-10

Use viewport specific layouts for mobile 50-10
Keep contents of header and footer regions consistent 50-10
Limit DOM node creation 50-10
Use ccLink binding for quicker page loading 50-11
Resize images using the ccResizeImage binding 50-11

1

Understand Extension Features

Oracle Commerce provides several sets of tools that you can use to extend the capabilities of the system. The primary ones are these:

- An extensive set of REST APIs allows external applications to make calls into the Oracle Commerce server. These APIs are supplemented by webhooks that the server can use to make calls out to external applications. For example, you can create an integration with an order management system in which Oracle Commerce uses webhooks to send order data to your OMS, and the OMS uses the Oracle Commerce REST APIs to update the order's status information as the order is processed.
- Custom widgets allow you to extend the functionality of your storefront by communicating with the Oracle Commerce server to access features that are not exposed by default. Custom widgets can also enhance the storefront by communicating with external systems such as social media sites.

Note that these tools are not mutually exclusive; you may need to use all of them to accomplish your objectives. For example, a custom widget might make a REST call to the Oracle Commerce server to request data for the storefront, and the server might then execute a webhook to obtain that data from an external system.

This manual focuses on building specific customizations using REST APIs, webhooks, custom widgets, and other tools. For general information about developing custom widgets, see Create a Widget.

2

Use the REST APIs

Oracle Commerce includes REST web service APIs you can use to create integrations with other products, and to build extensions to the administration interface and the storefront.

Learn about the APIs

The Oracle Commerce REST APIs consist of several sets of endpoints.

The Commerce REST API endpoints include the following:

- The Store API endpoints provide access to store functionality on the storefront server. A subset of the Store API endpoints, the Store Extension API, enables integrations and server-side extensions to access data that is not exposed to shoppers.
- The Admin API endpoints provide access to administrative functionality on the administration server. Two subsets of the Admin API endpoints, the Search Admin and Configuration and the Search Data and Indexing API endpoints, provide access to search functionality on the search server.
- The Agent API endpoints provide access to agent functionality on the administration server.
- The Social Wish Lists API endpoints are used to configure wish list features. This API is not described in this manual.

Each set of endpoints is different, although in many cases similar endpoints are available in multiple APIs. For example, the Store, Agent, and Admin APIs all have endpoints for working with orders, though they differ in the functions that they can perform.

Note: You should not make calls to the Admin API or the Agent API from a storefront application. If your application needs access to functionality or data provided by these endpoints, you can use the Store Extension API endpoints instead. These endpoints can be used by store integrations and server-side extensions, but should not be called from a browser.

Authentication is handled separately for each API. For example, logging into the Admin API does not give you access to the Agent endpoints. In addition, each API's endpoints differ in terms of which user roles provide access to them. For example, an account with CS Agent permissions does not necessarily also include Administrator permissions. See Configure Internal User Accounts for more information.

Note that each API is available only in certain environments:

- The Admin API and Agent API are available on the administration server only.
- The Store API is available primarily on the storefront server. It is also available on the administration server for previewing unpublished changes to the store.

You can find information about individual endpoints in the REST API documentation that is available through the Oracle Help Center. Be sure to select the version of the REST API documentation that matches the version of Oracle Commerce you are using.

ccdebug REST client

In your test environment, the administration server includes a REST client for making calls to the Commerce APIs. This client is available at the following URL:

`http://admin-server-hostname/ccdebug`

Note that this client can make calls only to the administration server it is running on. You can use it to access the Admin API and Agent API, and to access the Store API in preview mode. If you want to access endpoints on other servers, you can use a third-party client tool such as Postman.

REST API authentication

Oracle Commerce REST APIs use OAuth 2.0 with bearer tokens for authentication.

The REST APIs support two authentication approaches:

- To enable an external application such as an integration or server-side extension to be authenticated, the application must first be registered in the administration interface, as described in Register applications. As part of the registration process, an application key is generated. During authentication, the application key must be passed to Oracle Commerce using a `POST` request to the appropriate login endpoint.
- To authenticate an internal user or storefront shopper, the user login and password must be passed to Oracle Commerce using a `POST` request to the appropriate login endpoint.

In either case, if the authentication succeeds, the endpoint returns an access token that must be supplied in subsequent requests. Note that application keys and access tokens are long base64-encoded strings.

Use the application key for authentication

When you register an application, Oracle Commerce automatically generates a JSON Web Token called an application key. You send the application key in the authorization header of a `POST` request, and Oracle Commerce responds with an access token that the application must supply in subsequent requests.

Note: Application keys should be stored securely and all requests that include them must be sent via HTTPS. They should be used by integrations and server-side extensions only, and should not be sent by a browser.

Send the authorization header in a `POST` request to the appropriate login endpoint:

- Use `POST /ccadmin/v1/login` if your application makes calls to the Admin API endpoints on the administration server.
- Use `POST /ccapp/v1/login` if your application makes calls to the Store Extension Admin endpoints on the storefront server.
- Use `POST /ccappagent/v1/login` if your application makes calls to the Store Extension Agent endpoints on the storefront server.

The `Content-Type` header value must be set to `application/x-www-form-urlencoded`, and the body of the request must include the grant type `client_credentials`. For example:

```
POST /ccapp/v1/login  HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Authorization: Bearer <application_key>

grant_type=client_credentials
```

The following example shows the server's JSON response, which includes the access token:

```
{
    "access_token": "<access_token>",
    "token_type": "bearer"
}
```

Now whenever the application needs to access a secured endpoint, it must issue a request with an authorization header that contains the access token. The following example shows an authorization header for a request that returns orders:

```
GET /ccapp/v1/orders  HTTP/1.1
Authorization: Bearer <access_token>
```

Use login credentials for authentication

When an individual user such as a shopper logs in, there is no application key, so the user login and password must be supplied in the body of the request. The following example illustrates logging into a shopper account on the storefront server:

```
POST /ccstore/v1/login  HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=johndoe@example.com&password=g4dEj3w1
```

The response includes an access token to use in subsequent requests. Each API you log into returns a separate access token. The following example shows the server's JSON response, which includes the access token:

```
{
    "access_token": "<access_token>",
    "token_type": "bearer"
}
```

Multi-factor authentication

Logging into the Admin API or Agent API as an internal user involves multi-factor authentication. For example, to log into the Admin API, you issue a `POST` request to the `/ccadmin/v1/mfalogin` endpoint, and include the username, password, and passcode in the body of the request. For example:

```
POST /ccadmin/v1/mfalogin  HTTP/1.1
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=password&username=admin1@example.com&password=A3ddj3w2&totp_
code=365214
```

To obtain passcodes, the login account must be registered with the Oracle Mobile Authenticator app. See Access the Commerce administration interface for more information.

Note that account passwords and passcodes may expire or be changed, so you must make sure you have up-to-date values when you log in.

Refresh an access token

Each access token expires automatically after a predetermined period of time. Tokens associated with an application key expire after 5 minutes. Tokens associated with user credentials expire after 15 minutes.

To avoid being logged out of an API, you can replace the current token by issuing a `POST` request to the API's `refresh` endpoint. Include the current access token in the authorization header, just as you would for any other authenticated request. Oracle Commerce generates and returns a new token and restarts the clock. You then use the new token in the authorization headers of subsequent requests. Note that you may need to refresh the token multiple times (every 5 minutes for a login with an application key, every 15 minutes for a login with user credentials) if you need to remain logged in for an extended period of time.

The following example is an authorization header that refreshes an access token for the Admin API:

```
POST /ccadmin/v1/refresh  HTTP/1.1
Authorization: Bearer <old_access_token>
```

The following example shows the body of the server's response, which includes the new token:

```
{
    "access_token": "<new_access_token>",
    "token_type": "bearer"
}
```

Change the token expiration period (Admin API only)

As mentioned above, the expiration period for tokens associated with user credentials is 15 minutes by default. For the Admin API, you can change the expiration period using the `saveAdminConfiguration` endpoint. For example, to change the period to 30 minutes:

```
PUT /ccadmin/v1/merchant/adminConfiguration  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
    "sessionTimeout": 30
}
```

You can set `sessionTimeout` to any integer from 3 to 120. Note that the value you set also specifies the session timeout period for the administration interface, which is the period of inactivity after which the user is automatically logged out.

Access preview through the APIs

You can use the Store API on the administration server to access your store in preview mode. This requires a multi-step authentication procedure.

First, log into the Admin API on the administration server using an account that has the Administrator role. Issue a `POST` request to the `/ccadmin/v1/mfalogin` endpoint, and include the username, password, and passcode in the body of the request. For example:

```
POST /ccadmin/v1/mfalogin  HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=admin1@example.com&password=A3ddj3w2&totp_code=4
43589
```

The response returned includes an access token:

```
{
    "token_type": "bearer",
    "access_token": "<access_token>"
}
```

Next, create a new preview user by issuing a POST request to `/ccstore/v1/profiles` on the administration server. (You can skip this step if you have previously created a preview user.) In the authorization header field of the request, pass in the access token that was returned by `/ccadmin/v1/mfalogin`:

```
POST /ccstore/v1/profiles  HTTP/1.1
Authorization: Bearer <access_token>
```

In the body of the request, specify the values of the profile properties, as described in Create a shopper profile.

Now log in as the preview user by issuing a `POST` request to the `/ccstore/v1/login` endpoint on the administration server. Include the username and password in the body of the request. In addition, in the authorization header field of the request, pass in the access token that was returned by `/ccadmin/v1/mfalogin`:

```
POST /ccstore/v1/login  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=previewuser@example.com&password=Test1234
```

The response returned by `/ccstore/v1/login` includes a new access token:

```
{
    "token_type": "bearer",
```

```
    "access_token": "<access_token_2>"
}
```

You can now make requests to the `/ccstore/v1` endpoints on the administration server, passing in *<access_token_2>* (the access token that was returned by `/ccstore/v1/login`). You can also use the original access token (returned by `/ccadmin/v1/mfalogin`) to access `/ccadmin/v1` endpoints and to create preview users with the `/ccstore/v1/profiles` endpoint.

Note that if your Commerce instance is running multiple sites, preview requires a specific site context. You can specify the site when you log in as a preview user and in subsequent calls to the Store API. If you do not specify a site, the default site is used. See Use the APIs on instances running multiple sites for information about specifying the site in API calls.

Use the APIs on instances running multiple sites

If you are running multiple sites on your Commerce instance, your REST calls need to specify which sites they apply to.

There are two ways to specify the site:

- For calls to any of the APIs, you can specify the site using the `x-ccsite` header in the request.
- For calls to the Store API, you can explicitly include the domain name of the applicable site in the URL.

Note that if you do not specify a site in a call to the Store API, the call is directed to the default site. (See Configure sites for a discussion of the default site.) If you do not specify a site in a call to the Admin API or the Agent API, the call is applied to the instance as a whole. For example, if you specify a site for the `getOrders` Admin endpoint, only orders associated with that site are returned; if you do not specify a site, orders associated with all sites are returned.

`x-ccsite` header

You can use the `x-ccsite` header to specify the site for an API call. For example, if you have two sites, `siteA` and `siteB`, you could use this call to return the orders for `siteB`:

```
GET /ccadmin/v1/orders  HTTP/1.1
Authorization: Bearer <access_token>
x-ccsite: siteB
```

CORS support

For security purposes, web browsers implement the *same-domain policy*, which prevents JavaScript on a page served from one domain from accessing resources on another domain. In some cases, you may want to selectively override this policy to allow specific domains to access data on your stores.

Note: You can allow access to Admin and Agent endpoints as well. See Configure CORS support for the Admin or Agent endpoints below for information.

To enable a browser displaying a page on an external domain to access resources in your storefront environment, Commerce supports CORS (cross-origin resource sharing), which is a standard mechanism for implementing cross-domain requests. For a detailed description of CORS, see:

`https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS`

You configure CORS support in Commerce by explicitly specifying the external domains whose pages are permitted to access data on your sites. When a cross-domain request is submitted, the web browser is responsible for determining if access is permitted. In some cases, prior to sending the actual request, the browser first sends an `OPTIONS` method preflight request with headers that specify the domain that the request originates from and the expected HTTP method. In this situation, Commerce responds to the preflight request by indicating whether the actual request can be sent.

You specify the domains and methods permitted to access a specific site by using the `PUT /ccadmin/v1/sites/{siteID}` endpoint to set the value of the `allowedOriginMethods` property on the corresponding site object. For example, the following call enables cross-domain access to the `siteUS` site from two external domains and specifies which HTTP methods are permitted from each domain:

```
PUT /ccadmin/v1/sites/siteUS  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
    "properties": {
        "allowedOriginMethods": {
            "http://www.example1.com": "GET,OPTIONS",
            "http://www.example2.com": "GET,PUT,POST,OPTIONS"
         }
    }
}
```

After setting the value of `allowedOriginMethods` on the site object, publish the changes so they apply to the live context.

Note that the domain entries for the `allowedOriginMethods` property must be fully qualified, and cannot include wildcards. You must provide a separate entry for each domain or subdomain you want to enable access for. For example, if you want to provide CORS access to a domain named `www.example1.com` that has subdomains named `shoes.example1.com` and `shirts.example1.com`, you need to create three entries.

Also, even if you enable cross-domain requests, access to a resource from an allowed domain may require authentication. For example, calls to the Admin API endpoints require authentication, as described in REST API Authentication.

Configure CORS support for the Admin or Agent endpoints

You can enable external domains to access the Admin and Agent endpoints by using `PUT /ccadmin/v1/merchant/adminConfiguration` (for configuring access to the Admin API) or `PUT /ccadmin/v1/merchant/agentConfiguration` (for access to the Agent API). Each of these calls has an associated `allowedOriginMethods` property for specifying the domains and HTTP methods.

For example, the following call enables access to the Admin API from two external domains:

```
PUT /ccadmin/v1/merchant/adminConfiguration  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
    "properties": {
        "allowedOriginMethods": {
            "http://www.example3.com": "GET,POST,OPTIONS",
            "http://www.example4.com": "GET,PUT,POST,DELETE,OPTIONS"
         }
    }
}
```

Note that for the Admin and Agent endpoints, the `allowedOriginMethods` values take effect in the live context immediately. You do not need to publish the changes.

REST API query parameters

You can use query parameters to control what data is returned in endpoint responses.

The sections below describe query parameters that you can use to control the set of items and properties in responses, and the order of the items returned.

Control the set of items returned

To prevent the response from becoming too large, the number of items returned is limited by default to 250. You can override this value by using the `limit` query parameter to specify a different number. For example, the following call limits the number of orders returned to 5:

```
GET /ccadmin/v1/orders?limit=5
```

To page through the results, you can use the `offset` parameter. For example, suppose you have returned the first group of 250 orders using this call:

```
GET /ccadmin/v1/orders
```

You can return the next group of 250 using the following call:

```
GET /ccadmin/v1/orders?offset=250
```

The default value of `offset` is 0, which means the listing begins with the first item. So setting `offset` to 250 means the listing begins with the 251st item.

You can use `limit` and `offset` together. For example, to return the 401st through 600th order:

```
GET /ccadmin/v1/orders?limit=200&offset=400
```

Control the set of properties returned

Another way to reduce the size of responses is to return only certain properties. For example, products can have a large number of properties, but you may need only certain ones.

You can use the `fields` parameter to restrict the set of properties returned to only those you explicitly specify. The properties are specified as a comma-separated list. For example, to return only the `id` and `displayName` properties of products:

```
GET /ccadmin/v1/products?fields=items.id,items.displayName
```

Note that `items` is the key for the array of objects returned, so top-level properties are referred to as `items.`*`propertyName`* (for example, `items.displayName`). Properties of nested objects are specified using additional period delimiters. For example:

```
GET /ccadmin/v1/products?fields=items.listPrices.defaultPriceGroup
```

You can also use a special field, `totalResults`, to return the total number of items available (such as the total number of products in the catalog). For example:

```
GET /ccadmin/v1/products?fields=items.id,totalResults
```

Note that if a call does not use the `fields` parameter, `totalResults` is included in the response by default. For calls that use the fields parameter, `totalResults` is suppressed unless it is explicitly listed as one of the fields to include.

As an alternative to the `fields` parameter, which explicitly specifies the properties to include, you can use the `exclude` parameter to include all properties except the ones specified. For example, to return all of the properties of products except `longDescription`:

As with the `fields` parameter, properties of nested objects can be specified for the `exclude` parameter using additional period delimiters (for example, `items.listPrices.defaultPriceGroup`).

If you use both the `fields` and `exclude` query parameters in the same request, the `fields` parameter is applied first to determine the initial list of properties to return, and then the `exclude` parameter is applied to remove properties from that list.

You can also create persistent response filters that store a list of the properties to include and the properties to exclude. See Response filters.

Control the order of items returned

By default, the items returned are sorted by a predetermined property that depends on the type of item. For example, products are sorted by `displayName`.

You can use the `sort` parameter to specify a different property to sort by. For example:

```
GET /ccadmin/v1/products?sort=id
```

You can append `:asc` or `:desc` to the property name to specify sorting in ascending or descending order. For example, to sort by `id` descending:

```
GET /ccadmin/v1/products?sort=id:desc
```

If you do not specify a sort order, it defaults to ascending.

You can specify multiple properties for sorting. The following call returns results sorted first by `listPrice`, and then by `displayName` (for items with identical `listPrice` values):

```
GET /ccadmin/v1/products?sort=listPrice,displayName
```

Note that sorting is done before applying `limit` and `offset` values, so it can affect not only the order in which items appear in the response, but also which items are returned. For example, if `limit=200` and `offset=400`, items 401 to 600 are selected from the sorted list of all items. If you change the sorting criteria, items 401 to 600 may not be the same ones as before.

Filter results

Many endpoints that return a list of items support the `q` query parameter. This parameter is used for specifying a filter expression that restricts the set of the items returned, based on criteria such as numeric comparisons or string matching with the values of the items' properties. For example, the following call returns only those products whose `orderLimit` property has a value of less than 10:

```
GET /ccadmin/v1/products?q=orderLimit lt 10
```

For most endpoints that support it, the `q` parameter accepts filter expressions that use the syntax described in Section 3.2.2.2 of the System for Cross-Domain Identity Management (SCIM) specification, which is available at https://tools.ietf.org/html/draft-ietf-scim-api-12. A few endpoints accept filter expressions that use RQL syntax instead, as discussed below.

Use SCIM expressions for filtering

The SCIM specification defines standardized services for managing user identities in cloud environments. These services include a querying language for filtering the results returned by REST endpoints.

In SCIM filtering expressions, text, date, and time values must be enclosed in quotation marks, with date and time values using ISO-8601 format. (Numeric and boolean values should not be quoted.) For example, the following call returns products whose description property starts with `pa`:

```
GET /ccadmin/v1/products?q=description sw "pa"
```

The operators are case-insensitive, as are strings used for matching. So, for example, the following calls return identical results:

```
GET /ccadmin/v1/products?q=displayName co "shirt"
GET /ccadmin/v1/products?q=displayName CO "sHIrt"
```

Note that filter expressions must be URL encoded, so you must ensure that characters such as the quotation mark (") are escaped properly.

SCIM also supports the logical operators `AND`, `OR`, and `NOT`. For example, the following call returns products whose `orderLimit` property has a value between 5 and 10:

```
GET /ccadmin/v1/products?q=orderLimit gt 5 and orderLimit lt 10
```

Restrictions on filtering

Not all properties can be used in filter expressions. The following are some limitations you should be aware of:

- You can use only top-level properties of items in filter expressions. For example, for product endpoints, you cannot include properties of subobjects such as child SKUs.
- You can use a property in filter expressions only if it is returned by the endpoint you are calling. For example, if a specific product property is not returned by the `GET /ccadmin/v1/products` endpoint, then the property cannot be used with the `q` parameter for that endpoint. Note, however, that equivalent endpoints in different APIs (for example, `GET /ccadmin/v1/products` and `GET /ccstore/v1/products`) may not return identical sets of properties, so a property that is not returned by one of these endpoints may be returned by the other.

Also, if you have multiple custom product types, and two or more custom types have a custom property with the same name, the property cannot be used in filter expressions. For example, if you have two custom product types called Shoes and Hats, and each has a custom property called `material`, then you cannot use `material` in filter expressions. If only one custom product type has a `material` property, you can use the property in filter expressions.

Use RQL expressions for filtering

As mentioned above, a few endpoints use RQL syntax for filtering instead of SCIM syntax. These are:

```
GET /ccadmin/v1/exchangerates
GET /ccadmin/v1/orders
GET /ccadmin/v1/posts
GET /ccadmin/v1/serverExtensions
GET /ccadmin/v1/webhookFailedMessages
```

You can find information about RQL syntax in the Oracle Commerce Platform documentation:

https://www.oracle.com/technetwork/indexes/documentation/atgwebcommerce-393465.html

See the *Repository Query Language* section of the *Repository Guide*.

For example, this call uses RQL syntax for a numeric comparison:

```
GET /ccadmin/v1/exchangerates?q=exchangeRate > 3.5
```

This call uses RQL syntax for a timestamp comparison:

```
GET /ccadmin/v1/webhookFailedMessages?q=savedTime=datetime("2018-9-22
12:05:54 GMT")
```

Note that the endpoints that use RQL syntax by default can optionally use SCIM instead. To enable SCIM syntax for one of these endpoints, use the `queryFormat` query parameter. For example:

```
GET /ccadmin/v1/orders?queryFormat=SCIM&q=profileId eq "110658"
```

Response filters

Response filters provide an alternative way to use the `fields` and `exclude` query parameters.

Rather than using `fields` or `exclude` to explicitly list properties in the URL of a REST call, you can create persistent filters that store the set of properties to include or exclude. You can then specify a filter by name in the URL using the `filterKey` query parameter. For example, you could create a response filter named `productSummary` that lists product properties to include, and then invoke the filter like this:

```
GET /ccadmin/v1/products?filterKey=productSummary
```

Note:A response filter is essentially a wrapper for the `fields` and `exclude` query parameters, and the properties returned by a filter are the same as they would be for equivalent `fields` and `exclude` expressions. If you include the `filterKey` query parameter and either `fields` or `exclude` (or both) in an API call, `filterKey` is ignored, and `fields` and `exclude` are applied.

To view a list of response filters, use the `listFilters` endpoint in the Admin API:

```
GET /ccadmin/v1/responseFilters  HTTP/1.1
Authorization: Bearer <access_token>
```

Note that by default there are four response filters included with Commerce:

```
{
  "links": [
    {
      "rel": "self",
      "href": "http://myserver.example.com:7002/ccadmin/v1/
responseFilters"
    }
  ],
  "items": [
    {
      "include":
"items.id,items.displayName,items.type,items.variantValuesOrder,
          items.productVariantOptions,items.defaultProductListingSku,

items.dynamicPropertyMapLong,items.route,items.primarySmallImageURL,

items.primaryImageAltText,items.primaryImageTitle,items.childSKUs,
          items.listPrice,items.salePrice,items.relatedProducts,
          category.displayName,items.description,totalResults,offset,
          totalExpandedResults",
```

```
      "exclude": "items.childSKUs.largeImage,items.childSKUs.largeImageURLs,
          items.childSKUs.fullImageURLs,items.childSKUs.listPrices,

items.childSKUs.mediumImageURLs,items.childSKUs.primaryLargeImageURL,
          items.childSKUs.primaryMediumImageURL,

items.childSKUs.primaryThumbImageURL,items.childSKUs.thumbImageURLs,
          items.childSKUs.salePrices,items.childSKUs.thumbnailImage,
          items.childSKUs.barcode,items.childSKUs.denomination,
          items.childSKUs.model,items.childSKUs.productFamily,
          items.childSKUs.productLine,items.childSKUs.unitOfMeasure,
          items.childSKUs.saleVolumePrices",
      "key": "PLPData"
    },
    {
      "include":
"childCategories(items).displayName,childCategories(items).route,
          childCategories(items).id,
          childCategories(items).childCategories.displayName,
          childCategories(items).childCategories.route,
          childCategories(items).childCategories.id,
          childCategories(items).childCategories.childCategories.displayName,
          childCategories(items).childCategories.childCategories.route,
          childCategories(items).childCategories.childCategories.id,

childCategories(items).childCategories.childCategories.childCategories",
      "key": "categoryNavData"
    },
    {
      "include": "items.id,items.displayName,items.productVariantOptions,
          items.defaultProductListingSku,items.dynamicPropertyMapLong,
          items.route,items.primarySmallImageURL,items.primaryImageAltText,
          items.primaryImageTitle,items.childSKUs.listPrice,
          items.childSKUs.salePrice,items.listPrice,items.salePrice,
          items.relatedProducts,items.childSKUs.dynamicPropertyMapLong,

items.childSKUs.repositoryId,category.displayName,items.description",
      "key": "collectionData"
    },
    {
      "include":
"id,active,saleVolumePrices,listVolumePrices,route,configurable,
          dynamicPropertyMapLong,productVariantOptions,primaryThumbImageURL,
          notForIndividualSale,displayName,childSKUs.repositoryId,
          childSKUs.active,childSKUs.listPrice,childSKUs.salePrice,
          childSKUs.primaryThumbImageURL,childSKUs.listingSKUId,
          childSKUs.saleVolumePrices,childSKUs.listVolumePrices,
          childSKUs.dynamicPropertyMapLong",
      "key": "productData"
    }
  ]
}
```

Each filter must have a `key` (which is used to identify the filter), and either an `include` array (equivalent to the `fields` query parameter) an `exclude` array (equivalent to the `exclude` query parameter), or both.

You can view an individual filter using the `getFilter` endpoint. For example:

```
GET /ccadmin/v1/responseFilters/productData  HTTP/1.1
Authorization: Bearer <access_token>
```

Note that you should not modify or delete the default response filters, as they are used by widgets provided with Commerce, and these widgets may not work properly if the response filters are changed. For information about these response filters and how they are used by widgets, see Filter REST Responses.

Create response filters

You can create your own response filters using the `createFilter` endpoint. For example, the following call creates a new response filter named `productLabels`:

```
POST /ccadmin/v1/responseFilters  HTTP/1.1
Authorization: Bearer <access_token>

{
  "key": "productLabels",
  "include": "items.id,items.displayName,items.description"
}
```

The following call uses the `productLabels` filter to restrict the set of properties returned for products:

```
GET /ccadmin/v1/products?filterKey=productLabels  HTTP/1.1
Authorization: Bearer <access_token>
```

The following shows a portion of the response:

```
"items": [
  {
    "displayName": "A-Line Skirt",
    "description": "The simple perfect A line",
    "id": "xprod2535"
  },
  {
    "displayName": "Acadia Wood Chair",
    "description": "Craftsman meets classic in this attractive wood
chair",
    "id": "xprod2148"
  },
  {
    "displayName": "Americana Nightstand",
    "description": "Classic American design",
    "id": "xprod2103"
  },
```

```
  ...

  ]
}
```

Modify response filters

You can use the `updateFilter` endpoint to modify response filters. For example, the following call changes the set of properties returned by the `productLabels` filter shown above:

```
PUT /ccadmin/v1/responseFilters/productLabels  HTTP/1.1
Authorization: Bearer <access_token>

{
  "include": "items.displayName,items.description,items.listPrice"
}
```

Note that when you modify a response filter, the changes to the filter do not take effect until your JSON cache is cleared. This cache is cleared each time you publish changes on your Commerce instance. Changes to response filters themselves do not require publishing, so to force the cache to be cleared, you need to modify a publishable asset (such as an item in the product catalog) and then invoke publishing.

Delete response filters

You can use the `deleteFilter` endpoint to delete a response filter. For example:

```
DELETE /ccadmin/v1/responseFilters/productLabels  HTTP/1.1
Authorization: Bearer <access_token>
```

Error messages

Commerce uses a standard message format for REST errors.

REST calls that produce errors return the following response fields:

- `message` – the error message
- `status` – the HTTP status code
- `errorCode` – the system error code that uniquely identifies the error

For example:

```
{
    "message": "Required header is missing: x-ccasset-language",
    "status": "400",
    "errorCode": "82001"
}
```

For 400 level responses, `message` is a localized text that describes why the request triggered the error, and `errorCode` is an identifier that is the same regardless of locale. For 500 level responses, you can submit `errorCode` to your Oracle representative for troubleshooting.

Some errors use the multiple-error format instead, which encapsulates one or more errors in an `errors` array object. Each entry in the array is a separate error, with its own `message`, `status`, and `errorCode` values. In addition, the format includes top-level `message` and `status` values that apply globally to all of the errors. For example:

```
{
    "message": "Error while retrieving the products",
    "errors": [
        {
            "message": "Product Id xprod100 is invalid or non-
existent.",
            "status": "400",
            "errorCode": "20031",
        },
        {
            "message": "Product Id xprod102 is invalid or non-
existent.",
            "status": "400",
            "errorCode": "20031",
        }
    ],
    "status": "400"
}
```

Register applications

External applications can use the Oracle Commerce REST web services APIs to provide integrations or extensions to the administration interface or the storefront.

You must register an application in the administration interface before it can access Oracle Commerce data. Registering an application automatically generates the following:

- An application ID that identifies the application internally.
- An application key that you use to authenticate the application.

The application key is a JSON Web Token (JWT) from the Oracle Commerce OAuth server. Your registered application exchanges the key for an access token as part of the authentication flow. For more information, see Use the application key for authentication.

To register an application:

1. Click the **Settings** icon.
2. Click **Web APIs** and display the **Registered Applications** tab.
3. Click the **Register Application** button.
4. Enter a name for the application.
5. Click **Save**.
   The application ID and application key are automatically generated and the application is added to the list on the Registered Applications page.

To acquire the application key:

1. Click the **Settings** icon.

2. Click **Web APIs** and display the **Registered Applications** tab.
3. Click the name of the application whose key you want to get.
4. Click the **Application Key** box to reveal the key.
5. Copy the key and provide it to the application developer.
   See Use the application key for authentication for more information.

To reset the key for a registered application:

1. Click the **Settings** icon.
2. Click **Web APIs** and display the **Registered Applications** tab.
3. Click the name of the application whose key you want to reset.
4. Click **Reset**.
   The new application key is automatically generated. The existing application key is automatically revoked and can no longer be used to authenticate the application.

To unregister an application:

1. Click the **Settings** icon.
2. Click **Web APIs** and display the **Registered Applications** tab.
3. Click the name of the application you want to unregister.
4. Click **Delete**.
5. Click **Save**.
   The application's ID is removed from the system and its application key is automatically revoked.

3

Use Webhooks

Oracle Commerce includes webhooks that enable the server to make calls to external APIs. For example, you can configure the Order Submit webhook to send data to an order management system every time a shopper successfully submits an order.

There are two versions of each webhook, preview and production. Production webhooks send information from your live store to production environments of your external systems, while preview webhooks send information from your preview environment to the test or sandbox environments of your external systems.

Understand webhooks

Oracle Commerce includes two types of webhooks, asynchronous event webhooks and synchronous function webhooks:

- Event webhooks are asynchronous; they are triggered by JMS (Java Message Service) events. An event webhook call returns an HTTP status code. An event webhook request can be sent to multiple URLs.
- Function webhooks are synchronous; they are invoked explicitly in code. A successful function webhook call returns JSON data. A function webhook request can be sent to only one URL.

Both types of webhooks are described below.

Understand event webhooks

An event webhook sends a `POST` request to URLs you specify each time a Commerce event occurs. The body of the request contains the data associated with the event, in JSON format. The external system that receives the `POST` request returns an HTTP status code indicating whether the data was received successfully. A 200-level status code indicates the `POST` was successful. Any other code indicates failure; if this occurs, Commerce sends the `POST` request again. The webhook is executed up to five times until it succeeds or gives up.

The external system can use the data from the webhook request body in requests to the endpoints of the Commerce REST API endpoints. For example, you can configure the Order Submit webhook to send a notification to your order management system (OMS) every time a shopper successfully submits an order. When a change occurs to an order in the OMS, the OMS can issue a PUT request to the Update Order endpoint to modify the order in Commerce.

Commerce includes the following event webhooks:

Webhook	Notification event
Account Create	A new account was successfully created by an administrator. See Configure Business Accounts for more information.

Webhook	Notification event
Account Update	An existing account was successfully updated by an administrator. See Configure Business Accounts for more information.
Cart Idle	A cart that contains items has been inactive for the number of minutes you specify on the Abandoned Cart Settings page. See Configure Abandoned Cart settings for more information.
Export Complete	A data export process successfully completed.
Import Complete	A data import process successfully completed.
Inventory Update	Out-of-stock SKUs are back in stock. See Understand inventory for information about inventory data that determines whether a SKU is in stock.
Order Cancel	An agent canceled an order.
Order Cancel Without Payment Details	An agent canceled an order. The body for this webhook does not include payment details. See Understand webhooks and PCI DSS compliance for more information.
Order Submit	An order was successfully submitted by a customer or an agent.
Order Submit Without Payment Details	An order has been successfully submitted by a customer or an agent. The body for this webhook does not include payment details. See Understand webhooks and PCI DSS compliance for more information.
Publish Complete	Changes were successfully published.
Remorse Period Start	An order's customer remorse period has started. See Set the customer remorse period for more information.
Remorse Period Start Without Payment Details	An order's customer remorse period has started. See Set the customer remorse period for more information. The body for this webhook does not include payment details. See Understand webhooks and PCI DSS compliance for more information.
Return Request Update	A return request was successfully processed by an agent.
Return Request Update without Payment Details	A return request was successfully processed by an agent. The body for this webhook does not include payment details. See Understand webhooks and PCI DSS compliance for more information.
Shopper Profile Create	A new shopper registered on your instance.
Shopper Profile Update	A registered shopper changed their account details.
Shopper Profile Delete	A registered shopper's account has been deleted. See Delete Shopper Information for more information.
Order Redact	An order's properties have been redacted. See Delete Shopper Information for more information.

Webhook	Notification event
Request Quote	A shopper requested a quote for an order on a store that supports an external product configurator.
Update Quote	A shopper accepted or rejected a quote, or the quote was canceled on a store that supports an external product configurator.
Account Request	An account-based shopper has submitted an account registration request. See Configure Business Accounts for more information.
Contact Request	An account-based shopper or anonymous shopper has submitted a contact registration request. See Configure Business Accounts for more information.

Understand function webhooks

Like an event webhook, a function webhook sends a JSON notification to a URL you specify each time something happens on your store. For example, you can configure the Shipping Calculator webhook to send a notification to an external shipping service every time a shopper requests shipping costs for an order.

While an external system only sends an HTTP status code in response to an event webhook `POST` request, a system must respond to a function webhook `POST` request with information in JSON format. You must implement the external system's API to write code that processes the request and sends a response to Commerce. For example the Shipping Calculator webhook expects a set of shipping methods and their prices, which are displayed to the shopper who has requested them.

Commerce includes the following function webhooks:

Webhook	Description
Shipping Calculator	Integrates shipping services (such as UPS, USPS, or FedEx) into your store. See Integrate with External Shipping Calculators for more information.
Credit Card Payment	Integrates custom payment gateways that let your store accept credit card payments. See Create a Credit Card Payment Gateway Integration for more information.
Generic Payment	Integrates custom payment gateways that let your store accept various payment types. See Create a Generic Payment Gateway Integration for more information.
External Price Validation	Validates prices with an external pricing system. See Integrate with an External Pricing System for more information.
External Tax Calculation	Integrates tax processors that calculate sales tax in the shopping cart. See Configure Tax Processors for more information.
Order Approvals	Integrates systems that determine if an order placed on an account-based store requires approval. See Integrate with an external system for order approvals for more information.

Webhook	Description
Catalog and Price Group Assignment	Integrates systems that determine which catalog and price group a shopper should use to create orders. See Assign Catalogs and Price Groups to Shoppers for more information.
Contact Accounts Retrieval	Returns a list of service account IDs for the current user.
Services Retrieval	Returns information about a services or assets associated with the current user.
Service Actions	Performs a modify, renew, or cancel action on a service or asset.
Custom Currency Payment	Integrates custom payment gateways that let your store accept loyalty points payments.
Return Request Validation	Validates whether items maintained in an external order management system are eligible for return.
Return Request Validation Without Payment Details	Validates whether items maintained in an external order management system are eligible for return. The body for this webhook does not include payment details. See Understand webhooks and PCI DSS compliance for more information.
Order Qualification	Performs order qualification operations prior to submitting the order.
Order Validation	Validates the contents of the submitted order after final pricing is performed.

Validate function webhook responses

As discussed in the previous section, you must ensure the system receiving a function webhook `POST` responds by sending the appropriate JSON data to Commerce. To determine whether the response data conforms to the correct schema, the ccdebug REST client on the administration server in your test environment includes a validation tool for function webhooks. To access this tool, go to the following URL:

```
http://<admin-server-hostname>/ccdebug
```

Select the Function Webhooks tab, and then log into the Admin API. Follow the instructions on the screen to validate the format of your response payloads.

Configure webhooks

This section describes how to configure webhooks in the Oracle Commerce administration interface.

Before you configure the webhooks, you must identify the URLs of the web application or third-party provider where the webhooks will send notifications. You must use HTTPS URLs. See Troubleshoot SSL certificates for information about configuring the SSL certificates you install on your external system's web servers.
To configure a webhook:

1. Click the **Settings** icon.
2. Click **Web APIs** and display the **Webhook** tab.

3. Click the type of webhook you want to configure.
4. For a function webhook, enter the URL where you want to send the `POST` requests. For an event webhook, enter one or more URLs. Separate multiple URLs with commas.

   (You must enter HTTPS URLs. See Troubleshoot SSL certificates for information about configuring the SSL certificates you install on your external system's web servers.)
5. If the external system you are integrating with requires a username and password, enter them under Basic Authorization.
6. (Optional) To add a new property to the header of the request, click **Add New Header Property** and enter the name and value for the new property.
7. Click **Save**.

Once the webhook is set up, Oracle Commerce can push data to the external system specified by the URLs you entered.

Secure webhooks

Webhook events are signed so that the system receiving the event can verify their authenticity.

Webhook `POST` requests include an HMAC SHA1 signature in the `X-Oracle-CC-WebHook-Signature` header. This signature is calculated using the secret key to generate a hash of the raw UTF-8 bytes of the body of the post. A base64 encoding is then used to turn the hash into a string. If your secret key has been disclosed or compromised, you can generate a new one.

To generate a new secret key:

1. Click the **Settings** icon.
2. Click **Web APIs** and display the **Webhook** tab.
3. Click the type of webhook you want to configure.
4. Under HMAC Authentication, click **Reset**.

The following is sample code for generating a HMAC SHA1 signature from a secret key and content. In the case of webhooks, the content String would be the complete, unaltered body of the webhook `POST` request.

This or similar code can be used to verify that the message was sent by someone with access to the private key (presumably Oracle Commerce), and that the body of the message has not been altered after the fact:

```
import java.security.SignatureException;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

/**
 * This class provides an example of calculating an HMAC SH1
 * signature in java.
 */
public class CalcHmacSignature {

  /**
   * Calculate an HMAC SHA1 signature.
```

```
   *
   * @param pSecretKey the secret key (in string form).
   * @param pContent the content to create a signture for. For Commerce
   * Cloud WebHooks this should be the complete, unmodified body of
the post.
   *
   * @return The Base64-encoded HMAC SHA1 signature.
   *
   * @throws java.security.SignatureException if there's a problem
   */
  public static String getSignatureForBytes(String pSecretKey, String
pContent)
    throws java.security.SignatureException {

      try {
        // HMAC SHA1 key from the raw key bytes
        SecretKeySpec keySpec = new

SecretKeySpec(javax.xml.bind.DatatypeConverter.parseBase64Binary(secret
Key),
        "HmacSHA1");

        // get the Mac instance for HMAC SHA1
        Mac mac = Mac.getInstance("HmacSHA1");

        // initialize with our key spec
        mac.init(keySpec);

        // generate the signature from the UTF-8 bytes of the content
        byte[] digest = mac.doFinal(pContent.getBytes("UTF-8"));
        // base64-encode the hmac signature... there's a pre-JDK-8 one
        // tucked away in javax.xml.bind. If using Java 8, use the new
        // java.util.Base64 class instead.
        return javax.xml.bind.DatatypeConverter.printBase64Binary(
          digest);
      } catch (Exception e) {
        throw new SignatureException("Failed to generate signature: " +
                                     e.getMessage());
      }
    }

  public static void main(String[] args) throws SignatureException {
    if (args.length != 2) {
      System.out.println("Usage: CalcHmacSignature key content");
      System.out.println();
      System.out.println("  (Note that one shouldn't really have the
key ");
      System.out.println("   passed in on the command line.)");
      System.exit(-1);
    }
    System.out.println("Signature: " +  getSignatureForBytes(args[0],
                                                            args[1]));
  }
}
```

Troubleshoot webhooks

This section helps you fix problems you might encounter when configuring webhooks.

If you encounter any issues with the Oracle Commerce push, one strategy is to use another website external to your OMS to identify the source of the issue. This external website needs to accept message posts whose contents are undefined, and then make the messages available for display to a website user.

If the messages pushed by Oracle Commerce look correct on the other website, the issue may involve your internal systems.

Troubleshoot SSL certificates

Invalid or incorrectly-configured SSL certificates cause most of the problems you might experience when configuring webhooks. The webhooks verify SSL certificates when delivering each request, so it is important that they are properly installed.

Oracle Commerce does not support self-signed SSL certificates. Your SSL certificates must be issued from a trusted Certificate Authority.

After you install the SSL certificate on your web server, verify the installation and diagnose any problems, using one of the many free, third-party tools that are available.

Understand webhooks and PCI DSS compliance

Not all external systems you integrate with Oracle Commerce will comply with the Payment Card Industry Data Security Standard (PCI DSS).

For example, while your order management system will likely comply with PCI DSS, systems that manage services like email marketing or customer loyalty programs might not be compliant.

Oracle Commerce provides three webhooks that exclude payment details from the order data you send to systems that do not comply with PCI DSS:

- Order Submit Without Payment Details fires when an order has been successfully submitted by a customer or an agent.
- Return Request Update Without Payment Details fires when a return request has been successfully processed by an agent.
- Return Request Validation Without Payment Details queries an external system to determine whether an order is returnable.

Important: Oracle Commerce does not verify that systems to which you send webhook notifications comply with PCI DSS. You are responsible for determining if target systems are compliant. If you know for sure the target system does not comply with PCI DSS, or if you are unsure whether it does, use the versions of the webhooks Without Payment Details.

The following table describes all the components of the `paymentGroups` object that are excluded from the request for non-PCI compliant versions of the webhooks.

paymentGroups Component	Description
`authorizationStatus`	An array of authorization status objects.

paymentGroups Component	Description
token	The payment token string. This component is valid only if the paymentGroupClassType is tokenizedCreditCard.
expirationMonth	The two-digit credit card expiration month. This component is valid only if the paymentGroupClassType is tokenizedCreditCard.
expirationYear	The four-digit credit card expiration year. This component is valid only if the paymentGroupClassType is tokenizedCreditCard.
paymentGroupClassType	The class type of the payment group. Valid values are tokenizedCreditCard or externalPaymentGroup.
creditCardNumber	The last four digits of the credit card number. This component is valid only if the paymentGroupClassType is tokenizedCreditCard.
submittedDate	The date the payment was submitted.

The following example shows the paymentGroups portion of an Order Submit webhook POST request.


```
"paymentGroups": [{
            "id": "pg30411",
            "amount": 277.97,
            "authorizationStatus": [{
                "amount": 277.97,
                "errorMessage": "Request was processed successfully.",
                "authorizationDecision": "ACCEPT",
                "transactionId": "bupovkdslhd8or1i869pj1bls",
                "reasonCode": "100",
                "transactionUuid": "75afb7640b5a43e88341572869adbda6",
                "transactionSuccess": true,
                "currency": "USD"
            }],
            "currencyCode": "USD",
            "token": "9997000108950573",
            "expirationMonth": "02",
            "expirationYear": "2019",
            "paymentGroupClassType": "tokenizedCreditCard",
            "creditCardNumber": "1111",
            "submittedDate": "2015-12-16T10:25:41.894Z",
            "billingAddress": {
                "middleName": null,
                "lastName": "Shopper",
                "ownerId": null,
                "state": "NY",
                "address1": "100 MyStreet Ave",
                "address2": null,
                "address3": null,
                "companyName": null,
```

```
                "suffix": null,
                "country": "US",
                "city": "MyTown",
                "faxNumber": null,
                "postalCode": "13202",
                "phoneNumber": "212-555-0100",
                "email": "shopper@example.com",
                "county": null,
                "prefix": null,
                "firstName": "Sally",
                "jobTitle": null
            },
            "amountAuthorized": 277.97,
            "paymentMethod": "tokenizedCreditCard"
        }]
```

The following example shows the `paymentGroups` portion of an Order Submit Without Payment Details webhook `POST` request.

```
"paymentGroups": [{
            "id": "pg30411",
            "amount": 277.97,
            "billingAddress": {
                "middleName": null,
                "lastName": "Shopper",
                "ownerId": null,
                "state": "NY",
                "address1": "100 MyStreet Ave",
                "address2": null,
                "address3": null,
                "companyName": null,
                "suffix": null,
                "country": "US",
                "city": "MyTown",
                "faxNumber": null,
                "postalCode": "13202",
                "phoneNumber": "212-555-0100",
                "email": "shopper@example.com",
                "county": null,
                "prefix": null,
                "firstName": "Sally",
                "jobTitle": null
            },
            "amountAuthorized": 277.97,
            "paymentMethod": "tokenizedCreditCard"
        }]
```

Use the REST API to configure webhooks

This section provides an overview of actions you can perform with the Oracle Commerce Admin API's Event Webhooks and Function Webhooks endpoints.

See Learn about the APIs for information about accessing the endpoint documentation.

The following table describes the endpoints for the Event Webhooks resource.

Endpoint	Description and URI
`getWebHook`	Gets a specified webhook. `GET /ccadmin/v1/webhooks/{id}`
`getWebHooks`	Gets an array of webhooks, which can be narrowed by server type, for example, production. Each element of the returned array follows the format of that returned by `getWebHook`. `GET /ccadmin/v1/webhooks`
`updateWebHook`	Updates the URL properties of a specified webhook and, optionally, resets the secret key. You can use the REST API to change the number of times a webhook gets resent and the number of seconds between resends. `PUT /ccadmin/v1/webhooks/{id}`
`updateWebHooks`	Updates the URL properties of an array of existing webhooks and, optionally, resets the secret keys. You can use the REST API to change the number of times a webhook gets resent and the number of seconds between resends. `PUT /ccadmin/v1/webhooks`
`webhookOperation`	Resets the secret key of a specified webhook. `POST /ccadmin/v1/webhooks/{id}`

The following table describes the endpoints for the Function Webhooks resource.

Endpoint	Description and URI
`getFunctionWebHook`	Gets a specified webhook. `GET /ccadmin/v1/functionWebhooks/{id}`
`getFunctionWebHooks`	Gets an array of webhooks, which can be narrowed by server type, for example, production. Each element of the returned array follows the format of that returned by `getWebHook`. `GET /ccadmin/v1/functionWebhooks`
`updateFunctionWebHook`	Updates the URL properties of a specified webhook and, optionally, resets the secret key. `PUT /ccadmin/v1/functionWebhooks/{id}`
`updateFunctionWebHooks`	Updates the URL properties of an array of existing webhooks and, optionally, resets the secret keys. `PUT /ccadmin/v1/functionWebhooks`
`functionWebhookOperation`	Resets the secret key of a specified webhook. `POST /ccadmin/v1/functionWebhooks/{id}`

The following table describes the endpoints that you can use to manage failed event webhook messages. These are described in more detail in the next section.

Endpoint	Description and URI
`deleteFailedMessage`	Deletes a specified webhook message that failed to send. `DELETE /ccadmin/v1/webhookFailedMessages/{id}`
`getFailedMessage`	Gets a specified webhook message that failed to send. `GET /ccadmin/v1/webhookFailedMessages/{id}`
`getFailedMessages`	Gets an array of webhook messages that failed to send. Each element of the returned array follows the format of that returned by `getFailedMessage`. `GET /ccadmin/v1/webhookFailedMessages`
`updateFailedMessage`	Specifies a failed webhook message to resend. `PUT /ccadmin/v1/webhookFailedMessages/{id}`
`updateFailedMessages`	Specifies an array of failed webhook messages to resend. `PUT /ccadmin/v1/webhookFailedMessages`

Reduce the size of webhook requests

In some cases, the request payload from an Oracle Commerce webhook may be very large. This can cause performance issues, and external systems that impose size limits may reject the payload.

This section describes two ways Commerce provides to reduce the size of certain webhook payloads:

- Limit the number of items in payloads
- Send changes rather than complete data

Limit the number of items in payloads

You can configure webhooks to truncate certain properties in their request payloads. This capability is limited to specific Map and Collection properties, because these properties are the ones most likely to result in very large payloads. For example, the Shopper Profile Update webhook has two properties that can be truncated, `secondaryAddresses` and `shippingAddresses`.

To truncate these properties, you configure the webhook using endpoints in the Admin API, and set the value of `subEntityTruncationSize` to specify the maximum number of records to return for the property. (By default, the value of this setting is null, which means there is no limit.) You can set this property to an integer from 100 to 50000. For example, the following sets the threshold to 200:

```
PUT /ccadmin/v1/webhooks/production-updateProfile  HTTP/1.1
```

```
{
    "subEntityTruncationSize": 200
}
```

This means that if the `secondaryAddresses` or `shippingAddresses` property has 200 records or fewer, the addresses are included in the property as usual. But if one of these properties has 201 or more records, the property is omitted from the payload and is replaced with an indication that it has been truncated. In the following example, the `shippingAddresses` property has been truncated:

```
{
  "profileId": "110000",
  "profile": {
    ...
    "id": "110000",
    "shippingAddressesIsTruncated": true,
    ...
  },
  "siteId": "siteUS",
  "type": "atg.dps.ProfileUpdate"
}
```

The special property indicating that a property has been truncated is given the name of the truncated property with `IsTruncated` appended. So in this example, the name of the special property is `shippingAddressesIsTruncated`.

The external system that the webhook sends its payload to can take appropriate action when it receives indication that a property has been truncated. For example, the system could then make a call to the `getProfile` endpoint to get the addresses.

The following tables list the event webhooks and function webhooks that have properties whose values can be truncated:

Event Webhook	Truncatable Properties
Shopper Profile Create	• secondaryAddresses • shippingAddresses
Order Submit	• commerceItems
Order Submit Without Payment Details	• commerceItems
Remorse Period Start	• commerceItems
Remorse Period Start Without Payment Details	• commerceItems
Shopper Profile Update	• secondaryAddresses • shippingAddresses
Cart Idle	• items • childItems
Return Request Update	• commerceItems • returnItemList • childReturnItems
Return Request Update Without Payment Details	• commerceItems • returnItemList • childReturnItems

Event Webhook	Truncatable Properties
Account Create	• allSecondaryAddresses • secondaryAddresses
Account Update	• allSecondaryAddresses • secondaryAddresses
Request Quote	• commerceItems
Account Request	• secondaryAddresses
Picked Up Items	• commerceItems
Order Cancel	• commerceItems
Order Cancel Without Payment Details	• commerceItems

Function Webhook	Truncatable Properties
Shipping Calculator	• items • childItems
Credit Card Payment	• commerceItems
External Price Validation	• items • childItems
Generic Payment	• commerceItems
External Tax Calculation	• items • childItems
Order Approvals	• items • childItems
Catalog and Price Group Assignment	• items • childItems • allSecondaryAddresses • secondaryAddresses • shippingAddresses
Custom Currency Payment	• commerceItems
Return Request Validation	• commerceItems • returnItemList • childReturnItems
Return Request Validation Without Payment Details	• commerceItems • returnItemList • childReturnItems
Order Qualification	• items • childItems
Order Validation	• items • childItems
External Promotions	• items • childItems
Cancel Order Update	• commerceItems

Send changes rather than complete data

Another option for reducing the size of webhook payloads is to include only the items that have been changed. This option is available for two event webhooks, Shopper Profile Update and Account Update, and applies to the properties listed in the table below:

Webhook	Properties
Shopper Profile Update	• secondaryAddresses • shippingAddresses • loyaltyPrograms • secondaryOrganizations • siteProperties • roles • abandonedOrders • secondaryAddresses.types • shippingAddresses.types
Account Update	• secondaryAddresses • siteOrganizationProperties • members • relativeRoles • ancestorOrganizations • secondaryAddresses.types • siteOrganizationProperties.shippingMethods

To enable this option for the Shopper Profile Update or Account Update webhook, you set the webhook's `includeChangesOnly` value to `true`. (The default is `false`.) For example:

```
PUT /ccadmin/v1/webhooks/production-updateProfile  HTTP/1.1

{
    "includeChangesOnly": true
}
```

If this option is enabled for one of these webhooks, the properties listed above will include the records that have been modified. Each modified object includes a special `_actionCode` property whose value indicates how the object has changed. Valid values are `CREATE`, `UPDATE`, and `DELETE`. In the following example, the `shippingAddresses` property includes only the addresses that have been changed:

The following is an example of the `shippingAddresses` property in the request:

```
"shippingAddresses": [
  {
    "country": "US",
    "lastName": "Anderson",
    "address3": "",
    "city": "Syracuse",
    "address2": "",
    "prefix": "",
    "address1": "21 Cedar Ave",
    "postalCode": "13202",
    "companyName": "",
    "county": "",
    "suffix": "",
    "firstName": "Kim",
    "externalAddressId": null,
```

```
    "phoneNumber": "212-555-1977",
    "item-id": null,
    "_actionCode": "UPDATE",
    "repositoryId": "se-980031",
    "faxNumber": "",
    "middleName": "",
    "state": "NY"
  },
  {
    "_actionCode": "DELETE",
    "repositoryId": "140010"
  }
],
```

Manage failed webhook calls

This section discusses how to manage webhook calls when they fail.

Oracle Commerce provides two mechanisms for resending failed calls, one for event webhooks and one for function webhooks.

Queue event webhooks for resending

As discussed in Understand event webhooks, an event webhook sends a `POST` request to specified URLs each time a specific event occurs (for example, when an order is submitted). The body of the request contains the data associated with the event. An external system that receives the message returns a 200-level HTTP status code if the data is received successfully.

If the message is not received successfully by one of the URLs (for example, due to a network issue or an external system being down), Oracle Commerce sends the `POST` request again to that URL after a specified interval, and continues resending it until it succeeds or until the specified limit on the number of attempts is reached. By default, the interval is one hour, and the maximum number of attempts is 5, but you can change these values using either the `updateWebHook` or `updateWebHooks` endpoint.

Messages that are not delivered successfully after the maximum number of attempts are saved to a failed message log for later retrieval. Commerce includes a mechanism for managing failed messages automatically. You can also manage these failed messages manually using endpoints in the Admin REST API, or using the administration interface.

Manage failed messages automatically

To manage failed messages, Commerce monitors each target URL that a webhook is configured to send messages to, and if a URL is unresponsive, disables it as a target. For example, if the Order Submit webhook sends messages to three different URLs, and Commerce detects that calls to one of the URLs are failing consistently (returning non-200-level status codes, or not returning any response), it stops sending messages to this URL, while continuing to send messages to the other two URLs. The messages for the disabled URL are instead added directly to the failed message log.

Commerce continues monitoring the disabled target. When it detects that the URL is responding again, it resumes sending messages to it. Messages to the URL that failed previously (either reached the maximum number of retries, or were sent directly to the failed message log after Commerce disabled the target) are queued for resending. Note that it may take a while for all failed messages to be resent.

Manage failed messages using the REST API

The Admin REST API has several endpoints for viewing, deleting, and resending failed event webhook messages.

You can use the `getFailedMessage` endpoint to view a failed message that has been stored. You specify the ID of the message in a URL path parameter.

You can use the `getFailedMessages` endpoint to view all of the failed messages that have been stored. However, there may be a large number of messages, so you may find it desirable to return only a subset of the failed messages.

You can use the `q` query parameter with the `getFailedMessages` endpoint to filter the set of messages to return, based on values of the message properties. Typically you would filter based on `serverType` (`production` or `publishing`) or `messageType`. For example, the following call returns only those failed messages whose `messageType` is `atg.commerce.fulfillment.SubmitOrder`:

```
GET /ccadmin/v1/webhookFailedMessages?
q=messageType="atg.commerce.fulfillment.SubmitOrder"  HTTP/1.1
Authorization: Bearer <access_token>
```

You can also filter messages by when they were saved. For example, to return messages that were saved after a specific time:

```
GET /ccadmin/v1/webhookFailedMessages?q=savedTime >
datetime("2018-9-22  12:05:54 GMT")  HTTP/1.1
Authorization: Bearer <access_token>
```

To resend failed messages, you can either specify them individually using the `updateFailedMessage` endpoint, or use the `updateFailedMessages` endpoint to queue all of the stored messages for resending.

To resend a single failed webhook message, use the `updateFailedMessage` endpoint. The body of the request should set the `resend` property of the failed message to `true`. For example:

```
PUT /ccadmin/v1/webhookFailedMessages/200001  HTTP/1.1
Authorization: Bearer <access_token>

{
   "resend": true
}
```

Setting `resend` to `true` causes the message to be added to a queue for resending. If the message was originally sent to multiple URLs, the service that manages the queue ensures that the message is resent to only those URLs for which the webhook failed originally.

You can use the `updateFailedMessages` endpoint to queue all of the stored messages for resending, or use this endpoint with the `q` parameter to specify a subset of the stored messages for resending. Note, however, the format of filter expressions for this parameter is different from the format used for the `getFailedMessages` endpoint. With `getFailedMessages`, the `q` parameter accepts expressions in RQL format by default

(although it can optionally accept SCIM format instead). With `updateFailedMessages`, the `q` parameter accepts expressions in SCIM format only. See REST API query parameters for more information.

For example, the following call adds the failed production messages to the queue for resending:

```
PUT /ccadmin/v1/webhookFailedMessages?q=serverType eq "production"  HTTP/1.1
Authorization: Bearer <access_token>

{
   "resend": true
}
```

The following call adds only the production messages that were saved after a specific time:

```
PUT /ccadmin/v1/webhookFailedMessages?q=serverType eq "production" and
savedTime gt "2019-04-11T02:41:00.000Z"  HTTP/1.1
Authorization: Bearer <access_token>

{
   "resend": true
}
```

As an alternative to the `updateFailedMessages` endpoint, you can use the `requeueFailedMessages` endpoint, which allows you to specify the set of messages to resend using criteria specified in the endpoint request body.

Manage failed event webhooks in the administration interface

In addition to using Admin API endpoints to retrieve and resend failed event webhook messages, you can also perform these tasks in the Commerce administration interface.

To view a list of failed event webhook messages in the Commerce administration interface:

1. Click the **Service Operations** icon.
   Commerce displays a list of failed event webhook messages.
2. Use the options at the top of the page to sort and filter the list of failed webhook messages.
   For example, you can sort them from oldest to newest, and filter the list so that it displays only Order Submit messages in your production environment that failed in the last 24 hours.
3. Click a message's **Information** icon to see details about why the message failed.

Once you have filtered the list of failed webhook messages, you can resend or delete some or all of them.

- To resend a single webhook message, click its **Resend** icon. To resend all the webhook messages in the filtered list, click the Resend All icon at the top of the page.
  Commerce adds these messages to a queue for resending. If the message was originally sent to multiple URLs, the service that manages the queue ensures that the message is resent to only those URLs for which the webhook failed originally.
- To delete all the webhook messages in the list, click the **Delete All** icon at the top of the page. You cannot delete a message that is queued for retry.

Changes you make on the Service Operations page take effect as soon as you save them. You do not need to publish the changes.

Retry function webhooks

As discussed in Queue event webhooks for resending, Oracle Commerce includes a mechanism for managing failed event webhook calls. Because event webhooks are asynchronous, this mechanism supports queueing the failed messages for periodic retry.

Function webhooks, however, are synchronous, so the queueing mechanism used for event webhooks is not suitable for managing failed function webhook calls. Instead, Commerce provides a synchronous retry mechanism for certain function webhooks. If a webhook call using this mechanism does not initially succeed, it is immediately retried several times until it either succeeds or reaches the maximum number of retries, at which point it fails.

A call succeeds only if it returns an HTTP status code in the 2xx range. If any other status code is returned, or if nothing is returned due to a timeout or network error, the call fails.

Retry is supported for the following function webhooks:

- Shipping Calculator
- External Tax Calculation
- Catalog and Price Group Assignment
- Order Approvals
- Return Request Validation

Enable retry

Retry is controlled by two JSON properties, `supportsSynchronousRetry` and `synchronousRetries`. The `supportsSynchronousRetry` property is a read-only property that specifies whether the webhook supports the use of retry. It is set to `true` for the webhooks listed above, and is set to `false` for all other function webhooks. You cannot change the value of this property on any function webhook.

If a webhook's `supportsSynchronousRetry` property is `true`, you can enable retry for that webhook by setting its `synchronousRetries` property to an integer greater than zero (0). The value of `synchronousRetries` specifies the maximum number of times to retry the call. Note that if the value of `synchronousRetries` is 0, no retry will take place, even if `supportsSynchronousRetry` is `true`.

The following example sets the value of `synchronousRetries` for the Order Approvals webhook:

```
PUT /ccadmin/v1/functionWebhooks/production-checkOrderApprovalWebhook
HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
    "synchronousRetries": 5
}
```

The retry mechanism has a one minute limit. If the calls fail with an HTTP error immediately, the retries will be executed in rapid succession until the limit is reached or a call succeeds. But if the calls time out, the mechanism may reach the one minute limit before the maximum number of retries is reached.

Note that the Shipping Calculator and External Tax Calculation webhooks support a fallback mechanism that returns preconfigured default values if calls to the associated shipping calculator or tax calculator fail. If both fallback and retry are enabled for one of these webhooks, in some cases the fallback values may be returned even if the maximum number of retries has not been reached.

To see a list of all of the available function webhooks, including information about which ones support retry, use the `getFunctionWebHooks` endpoint in the Admin API. The response includes the values of the `supportsSynchronousRetry` and `synchronousRetries` properties for each function webhook.

Return webhook errors in an endpoint response

The External Tax Calculation webhook and Shipping Calculator webhook can be invoked by order-handling endpoints in the Store and Agent APIs. When one of these webhooks returns an error, the error is passed to the endpoint that called the webhook so it can be included in the endpoint's response.

The `description` and `errorCode` properties for each error in the webhook response are returned to the specific endpoint that invoked the webhook. This information is then included in the `errors` property in the endpoint response to make it available to the storefront or Agent Console.

The information returned by the endpoint varies depending on the type of error returned by the webhook. There are three main situations where errors may be returned:

- The webhook response is valid, but the response body includes errors.
- The webhook response is invalid. Depending on why the response is invalid, it may or may not include errors.
- The webhook does not return a response. This is typically due to network issues.

If the webhook response is valid, but the response body includes errors, the webhook returns a status code of 200. The endpoint then returns a status code of 400, and the `errors` property in the endpoint response includes the errors present in the webhook response body. For example:

```
{
  "errorCode": "40001"
  "message": "Please check the shipping address and try again.",
  "errors": [
    {
      "errorCode": "12345",
      "message": "First error in response"
    },
    {
      "errorCode": "67890",
      "message": "Second error in response"
    }
  ],
```

```
  "status": "400"
}
```

If the webhook response is invalid, the webhook returns a 400 or 500 level status code. The status code returned by the endpoint is the same as the webhook response status code. The `errors` property in the endpoint response includes an error message indicating the webhook response is invalid, as well as the errors present in the webhook response body. For example:

```
{
  "errorCode": "28088",
  "message": "Please check the selected shipping method. The
previously selected shipping method is no longer available.",
  "errors": [

    {
      "errorCode": "402",
      "message": "Invalid response code of 402 received for webhook
calculateShipping"
    },
    {
      "errorCode": "11111",
      "message": "First error in response"
    },
    {
      "errorCode": "22222",
      "message": "Second error in response"
    }
  ],
  "status": "402"
}
```

If there is no webhook response, the endpoint response status code is 502. The `errors` property in the endpoint response includes an error message with the actual exception.

Notes about the Shipping Calculator webhook

The error-handling behavior described above applies to the Shipping Calculator webhook when either an invalid shipping method or a valid external shipping method is used. If a valid internal shipping method is selected, however, no error is produced even if the webhook fails, returns an invalid response, or returns a valid response with errors.

In addition to the order endpoints, the Store and Agent shipping method endpoints can invoke the Shipping Calculator webhook, but only to retrieve the list of external shipping methods. If any errors are present in the webhook response body, the external shipping methods are not included in the endpoint response, but the endpoint does not return an error.

Summary of webhooks involved in order submission

Depending on how your sites are configured, there are a number of different webhooks that may fire before, during, and after order submission.

This section lists the webhooks associated with the order submission process, organized by the stage in the process when they may fire. Note that within a given stage, the set of webhooks and the order they fire may vary depending on the situation.

Prior to final order repricing

Order Qualification

External Price Validation

During final order repricing

External Promotions

Shipping Calculator

External Tax Calculation

After final order repricing

Order Approvals

Order Validation

During payment processing

Custom Currency Payment

Credit Card Payment

Generic Payment

After order submission

Order Submit

Order Submit Without Payment Details

4

Develop Server-Side Extensions

In addition to providing REST APIs and webhooks for integrating with external systems, and widgets for extending your storefront, Commerce also includes support for developing server-side extensions written in JavaScript.

Server-side extensions (SSEs) are applications built using the Express web framework and executed in the Node.js runtime environment. These extensions implement custom REST endpoints. For example, you could create a custom shipping calculator whose path is `/ccstorex/custom/v1/calculateShipping`.

SSEs provide the ability to execute custom logic on the server. This capability allows you to:

- Run critical code in a secure environment that cannot be easily modified by an end user.
- Run integration code on Oracle Commerce servers in the Commerce PCI zone, making it easier to achieve PCI compliance with custom extensions.
- Develop complex flows that integrate Commerce data and logic with external systems, and present a single endpoint to the storefront or to those systems.

Depending on the logic you implement, the endpoints you create can be called by a Commerce component such as a widget or webhook, or by an external system. Note that widgets and webhooks should call these endpoints on the storefront server, not the administration server.

A common pattern is to implement an endpoint that is designed to receive a POST request from a Commerce webhook. The SSE then converts the POST data (if necessary) and sends it to an external system. When the endpoint receives a response from the external system, the SSE converts the response (if necessary) and passes it to the webhook, which sends its response to the storefront.

SSEs perform and scale well. Note, however, that the server-side extension environment is not designed for large application development. It is intended for implementing integration code or small amounts of custom logic. An SSE application is limited to 1 GB of memory at runtime.

Access Node.js servers

Customer and partner developers do not have direct access to the Commerce Node.js servers. The storefront, admin, and agent server extension requests are routed to the Node.js servers, and the responses are sent back to the user.

The custom server applications can be accessed using the following URL routes:

- Storefront request route prefix: `/ccstorex/custom/*`
- Admin request route prefix: `/ccadminx/custom/*`
- Agent request route prefix: `/ccagentx/custom/*`

For example, if you develop a storefront endpoint with the route `/v1/calculateShipping`, it will be accessed at:

`https://<storefront-hostname>:<storefront-port>/ccstorex/custom/v1/calculateShipping`

Note that the `/v1` portion is recommended for versioning but not required. This matches the versioning scheme used for standard Commerce endpoints.

For an admin or agent endpoint, the route prefix includes `ccadminx` or `ccagentx`, and the URL includes the hostname and port for the administration server. Admin and agent endpoints require authentication using bearer tokens or user credentials, as described in REST API authentication.

Configure an SSE

A server-side extension consists of a directory structure containing code and configuration files.

The extension must be packaged as a single ZIP file that can be uploaded through the administration interface. The ZIP file should not be larger than 25 MB. The filename consists of the name of the application plus `.zip`. If an extension by the same name has already been uploaded, uploading this file will overwrite the existing extension.

The following is an example of the directory structure of an SSE ZIP file:

```
<extension-name>
   package.json
   config/
      config.json
   lib/
      *.js
   node_modules/
      <modules>
   locales/
      <locale>.json
```

This section describes how to configure an SSE, using settings available in `.json` files and in the server extension framework. For more information about SSE configuration management, see Server-Side Extension Configuration Management.

package.json

The top level of an SSE must include a metadata file named `package.json` containing predefined properties that provide information about the extension. For example:

```
{
   "name": "shippingCalculator",
   "version": "0.0.1",
   "description": "SSE that calls an external shipping calculator
service.",
   "main": "/app/index.js",
   "author": "Fred Smith",
   "dependencies" : { "config": "latest" },
```

```
   "devDependencies" : { "express" : "latest" },
   "authenticatedUrls": [],
   "publicUrls": [
      "/ccstorex/custom/v1/calculateShipping",
   ],
...
}
```

The following are key properties whose values you specify in the file:

- `main`: Identifies the JavaScript file that is the entry point into the application. This is executed and loads the extension to be run. Required.
- `publicUrls`: Lists all the routes for the extension that do not require authentication.
- `authenticatedUrls`: Lists the routes that only logged-in users can access. This is typically used for SSEs that implement Admin or Agent endpoints, but it can also be used to require shoppers accessing Store endpoints to be logged in.
- `dependencies`: Specifies the application's runtime dependencies. Ensure any modules you list here are also packaged with your application and uploaded to Commerce.
- `devDependencies`: Specifies the application's development-only dependencies. These modules should not be included in the uploaded ZIP file.

At least one route must be listed in either `publicUrls` or `authenticatedUrls`.

In addition, if your server-side extension needs to call out to any external domains, you must use the `package.json` file's `allowedUrls` property to specify an array of these URLs. For example:

```
"allowedUrls": [
   "https://www.example.com",
   "https://www.example2.com"
  ]
```

The URLs you specify are added to the list maintained by your Oracle Commerce environment. Calls to these URLs (including any subpaths) are permitted. Calls to domains that are not on this list are blocked. Note that calls from SSEs must use HTTPS and be sent over port 443 or 8443. For sending email, ports 465 (legacy SMTPS) and 587 (STARTTLS) are allowed; you can specify an SMTP server by just its domain name, such as `smtp.example.com`. SSEs can use a library such as Nodemailer that is capable of sending email through a proxy.

Note: The `allowedUrls` property replaces the `whitelistUrls` property that was formerly used for specifying external domains. The `whitelistUrls` property has been deprecated. If you currently have an SSE that has the `whitelistUrls` property, you should rename the property to `allowedUrls`. If you do not rename the property, Commerce will continue to permit access to the domains specified in `whitelistUrls`. However, if you add the `allowedUrls` property without removing `whitelistUrls`, Commerce will ignore `whilelistUrls` and permit access only to the domains specified in `allowedUrls`.

Development dependencies

The Oracle Commerce server-side extension framework includes a number of libraries that you can declare as `devDependencies` for use in developing your application. These libraries include:

- `Express`: Node.js web application framework.
- `Body-parser`: Middleware that parses incoming request bodies.
- `Jasmine`: Development framework for testing JavaScript code.
- `Moment`: Tool that parses, validates, manipulates, and displays dates and times in JavaScript.
- `Nconf`: Simple key-value store with support for both local and remote storage.
- `Winston`: Simple and universal logging library with support for multiple transports.

You should not include any of these libraries in the SSE ZIP file that you upload to Commerce, because they may override the versions of those libraries in the framework.

Environment variables

The extension server framework defines environment variables using the `Nconf` library. These variables are available to all SSEs running in your environment. Note that you should not include the `Nconf` library in an SSE's `node_modules` folder, because this will override the version in the framework and may prevent the SSE from accessing the environment variables.

The following table describes the predefined SSE environment variables:

Variable Name	Description
`atg.application.credentials`	Container for token, ID, and name values.
`atg.application.token`	3rd-party application key used to access the Commerce Admin API; child of `atg.application.credentials`.
`atg.application.id`	Application ID; child of `atg.application.credentials`.
`atg.application.name`	Application name; child of `atg.application.credentials`.
`atg.server.url`	URL of the storefront server.
`atg.server.admin.url`	URL of the administration server.
`general:proxy-server`	Commerce environment's proxy server.

The following is an example of how to use nconf to access environment variables in your code:

```
var nconf = require('nconf');
var storeServerUrl = nconf.get('atg.server.url');
var adminServerUrl = nconf.get('atg.server.admin.url');
var adminToken = nconf.get('atg.application.credentials:
atg.application.token');
```

You can create and manage additional environment variables for your extension server using the Extension Server Environment Variables endpoints in the Admin API. For example, you could create an environment variable called `my.custom.variable` and access it in your code like this:

```
var customVar = nconf.get('my.custom.variable');
```

config.json

You can use the `config.json` file in the SSE's top-level `config` directory to create and initialize properties specific to the SSE. You can use these properties to define additional configuration or to specify data to be passed to endpoints and webhooks that the SSE invokes. For example, the `config.json` file for an SSE that integrates with a payment gateway might be similar to this:

```
{
    "authenticationType": "http_signature",
    "enableLog": true,
    "logFilename": "logging.txt",
    "cardTypes": {
        "VISA": "001",
        "MASTERCARD": "002",
        "AMEX": "003"
        }
}
```

Implement an SSE

You implement an SSE as a Node.js application that creates custom endpoints to use as targets for Commerce webhook requests.

This section describes some key considerations to be aware of when you implement an SSE.

Export the application

To make the SSE application available in Commerce, you export the Express subapplication object from the `/app/index.js` module. For example:

```
// Export the subapplication to be embedded in the server-side extension
var express = require('express');
var subApplication = express.Router();
module.exports = subApplication;
```

Implement logging

The SSE framework includes a system logger that uses Winston with the necessary transports preconfigured. You can access the system logger in your code by including the `global.occ.logger` variable. (Inside the route definition, you can use either the `global.occ.logger` or `res.locals.logger` variable; both variables refer to the same logger.) You can download the logs by using the `GET /ccadminx/custom/v1/logs` endpoint.

The following example illustrates using the system logger in SSE code:

```
if(global.occ) {
  global.occ.logger.debug('Loading sample application.');
}
const response = 'hello system logger!!';
app.get('/v1/helloSystemLogger', function(req, res){

if(global.occ) {
```

```
  global.occ.logger.debug('Using global occ variable: ' + response);
  res.locals.logger.debug('Using res.locals.logger within the route
definition');
  res.status(200).send(response);
} else {
   res.status(500).send('error');
  }
});
```

Use supported MIME and file types

Server-side extensions support the following MIME types for inbound communication:

- `application/json`
- `application/xml`
- `text/xml`
- `application/x-www-form-urlencoded`

Template files, images, and other formats are not supported. You can use any MIME type for outbound communication.

An SSE's ZIP file should contain the following file types only:

- `.json`
- `.js`
- `.pem`
- `.txt`
- `.properties`

SSL certificate files must be in PEM format and be stored in the top-level `ssl/` folder of the extension. Each certificate must be in a separate file.

Make outbound calls

The extension server runs behind a proxy, and all outbound calls from the extension server must include the proxy details directly or indirectly. The Commerce HTTPS module indirectly includes the proxy details, so you typically should not need to pass them in.

If, however, you are using any other HTTP client libraries (for example, node-fetch or Axios), check the corresponding library documentation to determine whether you need to provide the proxy details. If so, they can be accessed using the following:

```
const nconf = require ('nconf');
const proxyServer = nconf.get ("general:proxy-server");
```

Note that if your code includes references to a Node.js proxy module, they should be to the `https-proxy-agent` module rather than `http-proxy-agent`.

Upload an SSE

Before you upload a server-side extension to Commerce, be sure to remove any unneeded modules from the `node_modules` folder. You should include only the modules that the application requires.

To upload an extension to Commerce, you must first obtain an application key and use it to log into the Admin REST API. For example:

```
POST /ccadmin/v1/login  HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Authorization: Bearer <application_key>

grant_type=client_credentials
```

Commerce returns a bearer token, which you supply with subsequent requests.

Now use the `POST /ccadmin/v1/serverExtensions` endpoint to upload the extension:

```
POST /ccadmin/v1/serverExtensions  HTTP/1.1
Content-Type: multipart/form-data
Authorization: Bearer <access_token>

filename: <extension_name>.zip
uploadType: extensions
force: true
fileUpload: <open_handle_to_extension_file>
```

Validate SSEs

The server-side extension framework performs validation checks on SSEs to avoid conflicts and errors.

By default, the SSE framework checks the `authenticatedUrls` and `publicUrls` property in each SSE's `package.json` file to determine whether the SSE has at least one URL that is registered with the main Express application. If it does, the SSE is considered valid. If at least one SSE running on the server is valid, network traffic can be routed to the server.

You can disable this checking for an individual SSE by setting the `validateModule` property in the `package.json` file to `false`:

```
{
"validateModule" : false,
...
}
```

Commerce also checks whether your SSE ZIP file includes libraries that are part of the server-side extension framework, such as Express, Winston, Nconf, or Moment. If these modules are included in your `node_modules` folder, they may cause unpredictable behavior, because the SSE framework will use the local `node_modules` copy instead of the global copy.

The SSE framework produces a warning if you upload an SSE that includes one or more of these modules. The framework also warns if an SSE's proxy configuration is invalid.

5

Manage Shopper Profiles

This section describes how to use the Commerce REST web services APIs to add custom properties to shopper profiles.

Understand shopper profiles and shopper types

Shopper profiles include a predefined set of properties that store information about shoppers at your store.

Profile properties include common shopper data, such as `firstName` and `phoneNumber`, plus data used internally by Oracle Commerce.

A shopper type defines the set of properties that exist for each shopper profile of that type. Shopper types are similar to product types, in that a shopper type is a template for a shopper profile rather than a profile itself. However, there is a currently only one shopper type available. The ID of this shopper type is `user`.

You cannot create additional shopper types, but you can add custom properties to the `user` shopper type. For example, if your store carries books, you might want to add a `favorite_author` property to shopper profiles. You can do this by using the Oracle Commerce Admin API to modify the `user` shopper type.

The Shopper Types resource in the Admin API includes endpoints for creating and working with custom properties of the `user` shopper type. The Profiles resource in the Admin API includes endpoints that you can use to set the values of properties of individual shopper profiles, including custom properties that have been added to the `user` shopper type.

When you add a custom property to the user shopper type, the property is added to all shopper profiles and preview profiles, including any new profiles you create and any profiles that already exist.

Shopper profiles and preview profiles

Oracle Commerce maintains two separate sets of shopper profiles. The main set represents actual shoppers who register on your storefront. You can access and modify existing shopper profiles, and create new shopper profiles, using the Store endpoints on your storefront server.

The other set of profiles is for preview users. These are fictional shoppers that you can use to log into your store on the Admin server to preview changes before they are published. You can access and modify existing preview profiles, and create new preview profiles, using the Store endpoints on your administration server. See Access preview through the APIs for information.

View a shopper profile

You can view a shopper profile using the REST API.

To view an existing shopper profile, first log into the Admin API on the administration server using an account that has the Administrator role.

For example:

```
POST /ccadmin/v1/mfalogin HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=admin1@example.com&password=A3ddj3w2&totp_
code=365214
```

Then issue a `GET` request to the `/ccadmin/v1/profiles/{id}` endpoint, providing the ID of the profile you want to view, and including the access token that was returned by `/ccadmin/v1/mfalogin`. For example:

```
GET /ccadmin/v1/profiles/se-570031 HTTP/1.1
Authorization: Bearer <access_token>
```

The following is an example of the response returned:

```
{
    "receiveEmail": "yes",
    "shippingSurchargePriceList": null,
    "lastName": "Anderson",
    "locale": "en_US",
    "priceListGroup": null,
    "links": [
        {
          "rel": "self",
          "href": "http://myserver.example.com:7002/ccadmin/v1/
profiles/se-570031"
        }
    ],
    "repositoryId": "se-570031",
    "id": "se-570031",
    "email": "kim@example.com",
    "shippingAddresses": [
        {
            "lastName": "Anderson",
            "postalCode": "13202",
            "phoneNumber": "212-555-1977",
            "county": null,
            "state": "NY",
            "address1": "21 Cedar Ave",
            "address2": null,
            "firstName": "Kim",
            "repositoryId": "se-980031",
            "city": "Syracuse",
            "country": "US"
        }
    ],
    "translations": {},
    "daytimeTelephoneNumber": null,
    "firstName": "Kim",
    "shippingAddress": {
        "lastName": "Anderson",
```

```
        "postalCode": "13202",
        "phoneNumber": "212-555-1977",
        "county": null,
        "state": "NY",
        "address1": "21 Cedar Ave",
        "address2": null,
        "firstName": "Kim",
        "repositoryId": "se-980031",
        "city": "Syracuse",
        "country": "US"
    }
}
```

The response shows the predefined profile properties that are exposed by Oracle Commerce. You can set the values of these properties for an existing profile using the `PUT /ccadmin/v1/profiles/{id}` endpoint on the administration server.

Create a shopper profile

To create a new shopper profile, issue a `POST` request to the `/ccadmin/v1/profiles` endpoint on the administration server.

Specify the values of the profile properties as a JSON map in the body of the request. For example:

```
POST /ccadmin/v1/profiles  HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
    "receiveEmail": "yes",
    "lastName": "Wilson",
    "locale": "en_US",
    "email": "fredW@example.com",
    "firstName": "Fred"
}
```

If the profile is created successfully, the response body returned includes the ID for the new profile and a link to the URL used in the request:

```
{
    "id": "120000",
    "links": [
        {
            "rel": "self",
            "href": "http://myserver.example.com:7002/ccadmin/v1/profiles"
        }
    ]
}
```

Specify the email address and login

The value of the `login` property on the profile is used as the shopper's username. Each username must be unique. On many storefronts, the username is also the shopper's email address, in which case the value of the `login` property is the same as the value of the `email` property.

When you create a profile through an API call, the request must explicitly set the `email` property, but can omit the `login` property. If the request does not specify a value for the `login` property, it is set to the same value as the `email` property. For example, the following request creates a profile with `bobW@example.com` as both the username and the email address:

```
POST /ccadmin/v1/profiles HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
    "receiveEmail": "yes",
    "lastName": "Wilson",
    "email": "bobW@example.com",
    "firstName": "Bob"
}
```

If you want the username to be different from the email address, you can set separate values for the `email` and `login` properties in the request. For example, the following request creates a profile with `fwilson` as the username and `fredW@example.com` as the email address:

```
POST /ccadmin/v1/profiles HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
    "login": "fwilson",
    "receiveEmail": "yes",
    "lastName": "Wilson",
    "email": "fredW@example.com",
    "firstName": "Fred"
}
```

Allow profiles to share an email address

One reason you may want the `login` and `email` values to differ is to allow multiple profiles to share an email address. By default, Commerce requires each profile to have a unique email address, but your business needs may make this restriction undesirable. If so, you can use the following call to allow multiple profiles to share the same email address:

```
PUT /ccadmin/v1/merchant/shopperProfileConfiguration  HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>
```

```
{
  "duplicateEmailsAllowed": true
}
```

Note that once your Commerce instance has profiles that share an email address, you cannot set `duplicateEmailsAllowed` back to `false`. If you try to do this, the call will return an error.

Block weak passwords

As discussed in the Configure Shopper Settings, you can configure password policies through the Commerce administration interface. The purpose of these settings is to ensure that shoppers do not use passwords that are easy to guess.

The properties set through the administration interface can also be set using the `savePolicies` endpoint in the Admin API. In addition to these properties, this endpoint can set the `blockCommonPasswords` property, which has no equivalent setting in the administration interface. If `blockCommonPasswords` is set to `true`, Commerce rejects weak passwords, regardless of whether they meet the criteria specified in the other properties.

The properties set using this endpoint are site-specific, so the call must specify the site using the `x-ccsite` header. For example, the following call specifies values for a site's password policy settings, including `blockCommonPasswords`, which it sets to `true`:

```
PUT /ccadmin/v1/merchant/profilePolicies  HTTP/1.1
Authorization: Bearer <access_token>
x-ccsite: 100002

{
  "guestCheckoutEnabled": true,
  "numberOfPreviousPasswords": 3,
  "numberOfPreviousPasswordsMinVal": 1,
  "passwordExpirationEnabled": false,
  "passwordExpirationLengthMinVal": 1,
  "sessionTimeoutLength": 15,
  "cannotUsePreviousPasswords": false,
  "passwordExpirationLength": 90,
  "minPasswordLengthMinVal": 4,
  "sessionTimeoutEnabled": true,
  "minPasswordLengthMaxVal": 64,
  "useNumber": true,
  "cannotUseUsername": false,
  "useMinPasswordLength": true,
  "minPasswordLength": 8,
  "numberOfPreviousPasswordsMaxVal": 6,
  "useMixedCase": false,
  "sessionTimeoutLengthMinVal": 1,
  "sessionTimeoutLengthMaxVal": 120,
  "useSymbol": false,
  "blockCommonPasswords": true
}
```

If `blockCommonPasswords` is `true`, Commerce rejects any password that appears in its dictionary of weak passwords. When the shopper specifies a new password, it is compared against all of the entries in the dictionary, and if it matches one of those entries, it is rejected.

In addition to the passwords listed in the dictionary, you can specify your own list of passwords to block using the `updateRestrictedWords` endpoint. For example:

```
POST /ccadmin/v1/merchant/profilePolicies/updateRestrictedWords
HTTP/1.1
Authorization: Bearer <access_token>

{
     "add": ["frog", "cow", "pig"]
}
```

The response includes an `items` array that lists your blocked entries:

```
{
  "links": [
    {
      "rel": "self",
      "href": "http://myserver.example.com:7002/ccadmin/v1/merchant/
profilePolicies/updateRestrictedWords"
    }
  ],
  "items": [
    "frog",
    "cow",
    "pig"
  ]
}
```

You can also display your current list using the `getRestrictedWords` endpoint (`GET /ccadmin/v1/merchant/profilePolicies/restrictedWords`).

Note that your list of blocked passwords is not site-specific. The entries you specify apply to all sites whose `blockCommonPasswords` property is `true`.

The `updateRestrictedWords` endpoint can also take a `delete` array for specifying entries to remove from your list. For example:

```
POST /ccadmin/v1/merchant/profilePolicies/updateRestrictedWords
HTTP/1.1
Authorization: Bearer <access_token>

{
     "delete": ["frog", "cow"]
}
```

Deleting entries affects only your own list of blocked passwords. You cannot modify the dictionary that Commerce uses. For example, if you add an entry to your list that matches a value already in the dictionary, and subsequently delete that entry from your list, it does not affect the entry for that value in the dictionary.

Note that changing settings in the password policy does not invalidate existing passwords. The policy change is applied only when a shopper attempts to set a new password.

Disable soft login

As discussed in Configure Shopper Settings, a logged-out shopper can still see personalized content based on their profile. This is referred to as soft login. Soft login is enabled by default. You can use the `/ccadmin/v1/merchant/profilePolicies` REST API endpoint to disable or enable it.

If `softLoginEnabled` is set to `false`, Commerce disables soft login. The following call sets `softLoginEnabled` to `false` for the specified site:

```
PUT /ccadmin/v1/merchant/profilePolicies  HTTP/1.1
Authorization: Bearer <access_token>
x-ccsite: 100002

{
  "softLoginEnabled": false
}
```

Create a shopper profile on an instance running multiple sites

If you are running multiple sites on your Commerce instance, shopper profiles are shared by all of these sites. If a shopper registers on one site running on the instance, the shopper's profile is automatically available on all sites running on the instance.

However, the values of certain properties in the profile can be site-specific. For example, the values of the `receiveEmail` property are site-specific.

When you create a shopper profile using a `POST` request, you can specify a site using the `x-ccsite` header, as described in Use the APIs on instances running multiple sites. The resulting profile applies to all of the sites on your Commerce instance, but the value you provide for a site-specific property applies only to the site you specify. (If you do not specify a site, the value is applied to the default site only.) On all other sites, the value of this property is set to its default. You can modify the property on a specific site using a `PUT` request. For example:

```
PUT /ccadmin/v1/profiles/120000  HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>
x-ccsite: 100001

{
    "receiveEmail": "yes"
}
```

Similarly, if you use a `GET` request to view a profile, the value returned for a site-specific property reflects the site specified in the `x-ccsite` header, or the default site if no site is specified.

Set a site-specific profile property for multiple sites

If your Commerce instance is running multiple sites, shoppers can set site-specific profile properties for each site individually. For example, the storefront for each site can provide a checkbox for setting the `receiveEmail` property, with the setting applying only to the current site.

A drawback of this approach is that in order to configure settings on all sites, a shopper must access each site separately. To simplify profile configuration, you may instead want to provide a way for the shopper to configure multiple sites in one place.

To enable this, you can modify your storefront to use the `updateSiteProperties` endpoint in the Store API. This endpoint sets the values of site-specific properties of the current profile (the profile in the current calling context). Note that the current profile must be for a registered shopper, which means the shopper must be logged in.

For example, the following call sets the value of the `receiveEmail` property of the current profile for two different sites:

```
PUT /ccstore/v1/profiles/current/siteProperties  HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
  "siteProperties": [
    {
      "site": {
        "id": "siteUS"
      },
      "properties": {
        "receiveEmail": "yes"
      }
    },
    {
      "site": {
        "id": "siteUK"
      },
      "properties": {
        "receiveEmail": "no"
      }
    }
  ]
}
```

The response shows the values of the `receiveEmail` property for all sites on which it has been set. For example, the following is a portion of the response to the above request:

```
{
  ...
    "items": [
        {
            "site": {
                "id": "siteUK"
            },
            "properties": {
                "receiveEmail": "no"
            }
        },
        {
            "site": {
                "id": "siteUS"
```

```
            },
            "properties": {
                "receiveEmail": "yes"
            }
        }
    ],
    "totalNumberOfItems": 2
}
```

You can also display the current values of site-specific properties using the `listSiteProperties` endpoint. For example:

```
GET /ccstore/v1/profiles/current/siteProperties  HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>
```

The response contains the same data as the `updateSiteProperties` endpoint response.

Some things to note about using the `updateSiteProperties` and `listSiteProperties` endpoints:

- Before you can set the value of a site-specific property for an individual site, the site object's `enabled` property must be set to `true`, and the site object must be published.
- You can use the `updateSiteProperties` endpoint to update the value of a site-specific property for all of the sites running on the instance, or for a subset of the sites. In the latter case, you include only the sites you want to update in the body of the request.
- Setting the values of a site-specific property affects only sites that have already been created. If you subsequently create an additional site, the property's value for that site is set to the property's default value on all shopper profiles.
- If the value of a site-specific property has not been set explicitly for a site, the value for that site is set to the property's default value, but no entry for the site is included in the `updateSiteProperties` or `listSiteProperties` response.

Send site-specific properties in webhooks

Several webhooks include profile data for the current shopper in their request bodies. This data includes a `sitePropertiesList` array that lists the values of site-specific properties such as `receiveEmail` for each site. These webhooks are:

- Cart Idle
- Shopper Registration
- Shopper Account Update
- External Price Validation
- External Tax Calculation

The following is an example of a `sitePropertiesList` array that lists site-specific property values in a webhook request:

```
"sitePropertiesList": [
      {
         "site": {"id": "siteDE"},
         "properties": {"receiveEmail": "no"}
```

```
      },
      {
         "site": {"id": "siteUS"},
         "properties": {"receiveEmail": "yes"}
      }
   ]
```

View a shopper type

To view a shopper type, issue a `GET` request to the `/ccadmin/v1/shopperTypes/{id}` endpoint on the administration server.

The following example illustrates calling this endpoint with `user` (the only type currently available) specified as the value for `id`:

```
GET /ccadmin/v1/shopperTypes/user HTTP/1.1
Authorization: Bearer <access_token>
```

The following example shows a portion of the response returned:

```
{
    "id": "user",
    "links": [
        {
           "rel": "self",
           "href": "http://myserver.example.com:7002/ccadmin/v1/
shopperTypes/user"
        }
    ],
    "properties": {
         ...
        "lastName": {
            "writable": true,
            "localizable": false,
            "label": "Last name",
            "type": "shortText",
            "uiEditorType": "shortText",
            "textSearchable": false,
            "multiSelect": null,
            "dimension": false,
            "internalOnly": null,
            "default": null,
            "audienceVisibility": null,
            "editableAttributes": [
                "textSearchable",
                "multiSelect",
                "dimension",
                "internalOnly",
                "default",
                "label",
                "required",
                "audienceVisibility",
                "searchable"
```

```
            ],
            "length": 254,
            "required": false,
            "searchable": false
        },
    },
    ...
}
```

This example shows a portion of the response corresponding to one of the predefined profile properties appearing in the previous example. The property has a group of attributes that control the behavior associated with the property. To modify the `user` shopper type, you can create custom properties or modify existing properties by setting the values of these attributes.

Settable attributes of shopper type properties

The following table describes all of the attributes of shopper type properties that you can set through the Shopper Types endpoints of the Admin API:

Attribute	Description
`label`	String containing the display name of the property on the storefront. This attribute is localizable. If a value is not supplied, the attribute value is set to the name of the property.
`type`	Data type of the property. Valid values are `shortText`, `richText`, `number`, `date`, and `checkbox`. This attribute must be set explicitly, and it cannot be modified after it is set.
`uiEditorType`	Data type for determining the type of user interface control for editing the value of the property. This attribute must be set to the same value as the `type` attribute for the property, and cannot be modified after it is set.
`internalOnly`	Boolean that specifies whether the property can be displayed on the storefront. If `true`, the property cannot be displayed. Defaults to `false` if not specified explicitly.
`default`	Value to use for the property if a value is not specified. Defaults to null if not set explicitly. Must be set explicitly if the required attribute is `true`.
`required`	Boolean that specifies whether the property value must be set. If this attribute is `true`, the property must have a default value set through the `default` attribute. Defaults to `false` if not set explicitly.
`audienceVisibility`	String that determines whether the property appears as a choice in the Attributes field of the audience interface. For shopper profile properties, this value should be set to all. See Define Audiences.

Note that in addition to the attributes listed in the table above, there are several more attributes whose values are returned when you issue a `GET` request to the `/ccadmin/v1/`

`shopperTypes/{id}` endpoint. These attributes either cannot be set through the API, or if they are set, have no effect.

Add custom properties to a shopper type

To add custom properties to a shopper type, issue a `PUT` request to the `/ccadmin/v1/shopperTypes/{id}` endpoint on the administration server.

Use the following format:

- The request header must specify the `x-ccasset-language` value.
- The request body is a map where each key is the ID of a new property, and each value is an object that specifies the values of the attributes of the property.
- Each object is also a map, with each key being the name of an attribute and each value being the corresponding attribute value.

Note that the ID of a custom property must include the underscore character (_). This ensures that the ID will not conflict with any properties that Commerce adds to shopper types in the future. The endpoint produces an error if you attempt to create a custom property without an underscore in its ID.

The following example shows a sample request for adding two custom properties to the user shopper type:

```
PUT /ccadmin/v1/shopperTypes/user HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en
Content-Type: application/json

{
 "properties": {
     "loyalty_program_member": {
         "label": "Member of loyalty program?",
         "type": "checkbox",
         "uiEditorType": "checkbox",
         "internalOnly": true,
         "default": false,
         "required": true,
         "audienceVisibility": "all"
     },
     "favorite_website": {
         "label": "Favorite Web Site",
         "type": "shortText",
         "uiEditorType": "shortText",
         "internalOnly": false,
         "default": null,
         "required": false,
         "audienceVisibility": "all"
}}}
```

See Settable attributes of shopper type properties for information about specifying the attribute values.

The following is a portion of the response that shows the new properties:


```
{
  ...
"properties": {
     ...
    "loyalty_program_member": {
         "writable": true,
         "localizable": false,
         "label": "Member of loyalty program?",
         "type": "checkbox",
         "uiEditorType": "checkbox",
         "textSearchable": false,
         "multiSelect": null,
         "dimension": false,
         "internalOnly": true,
         "default": false,
         "audienceVisibility": "all"
         "editableAttributes": [
             "textSearchable",
             "multiSelect",
             "dimension",
             "internalOnly",
             "default",
             "label",
             "required",
             "audienceVisibility",
             "searchable"
         ],
         "length": 19,
         "required": true,
         "searchable": false
         },
    "favorite_website": {
         "writable": true,
         "localizable": false,
         "label": "Favorite Web Site",
         "type": "shortText",
         "uiEditorType": "shortText",
         "textSearchable": false,
         "multiSelect": null,
         "dimension": false,
         "internalOnly": false,
         "default": null,
         "audienceVisibility": "all"
         "editableAttributes": [
             "textSearchable",
             "multiSelect",
             "dimension",
             "internalOnly",
             "default",
             "label",
             "required",
             "audienceVisibility",
             "searchable"
```

```
            ],
            "length": 254,
            "required": false,
            "searchable": false
            },
            ...
        }
        ...
}
```

Set custom properties on a shopper profile

After adding new custom properties to the `user` shopper type, you can use the Admin API to set the values of these properties on shopper profiles.

You can issue a `PUT` request to the `/ccadmin/v1/profiles/{id}` endpoint on the administration server to set the values of custom properties on an existing shopper profile, or issue a `POST` request to the `/ccadmin/v1/profiles` endpoint to set these and other properties when you create a new shopper profile. Custom properties you create on the user shopper type are automatically exposed to these endpoints.

The following example shows a sample request body for setting the two custom properties created in the previous section on an existing shopper profile:

```
{
   "loyalty_program_member": true,
   "favorite_website": www.oracle.com
}
```

The following shows the response body returned:

```
{
    "receiveEmail": "yes",
    "shippingSurchargePriceList": null,
    "lastName": "Anderson",
    "locale": "en_US",
    "priceListGroup": null,
    "links": [
        {
          "rel": "self",
          "href": "http://myserver.example.com:7002/ccadmin/v1/
profiles/se-570031"
        }
    ],
    "repositoryId": "se-570031",
    "id": "se-570031",
    "loyalty_program_member": true,
    "email": "kim@example.com",
    "shippingAddresses": [
        {
            "lastName": "Anderson",
            "postalCode": "13202",
            "phoneNumber": "212-555-1977",
```

```
            "county": null,
            "state": "NY",
            "address1": "21 Cedar Ave",
            "address2": null,
            "firstName": "Kim",
            "repositoryId": "se-980031",
            "city": "Syracuse",
            "country": "US"
        }
    ],
    "translations": {},
    "daytimeTelephoneNumber": null,
    "favorite_website": "www.oracle.com",
    "firstName": "Kim",
    "shippingAddress": {
        "lastName": "Anderson",
        "postalCode": "13202",
        "phoneNumber": "212-555-1977",
        "county": null,
        "state": "NY",
        "address1": "21 Cedar Ave",
        "address2": null,
        "firstName": "Kim",
        "repositoryId": "se-980031",
        "city": "Syracuse",
        "country": "US"
    }
}
```


Note that the values of custom profile properties can be set on the storefront using a custom widget that accesses the `UserViewModel`. The view model can then call the `updateProfile` REST endpoint to update the data on the server. For details, see Access custom properties using the UserViewModel.

Create custom properties for addresses

You can use the Oracle Commerce REST web services APIs to add custom properties to shopper and account addresses.

See Use the REST APIs for information you need to know before using the services. Note that the view models for addresses support custom properties, but widgets included with Oracle Commerce require customization to access these properties.

View the `contactInfo` item type

Addresses are stored internally as instances of the `contactInfo` item type. You can view this item type with the following call:

```
GET /ccadmin/v1/itemTypes/contactInfo  HTTP/1.1
Authorization: Bearer <access_token>
```

The following example shows a portion of the response representing one of the `contactInfo` properties. Each property has a group of attributes whose values control the behavior associated with the property:

```
...
{
   "length": 254,
   "label": "City",
   "type": "shortText",
   "required": false,
   "searchable": false,
   "writable": true,
   "internalOnly": false,
   "uiEditorType": "shortText",
   "default": null,
   "audienceVisibility": "all",
   "localizable": false,
   "textSearchable": false,
   "id": "city",
   "dimension": false,
   "editableAttributes": [
      "internalOnly",
      "default",
      "audienceVisibility",
      "textSearchable",
      "label",
      "dimension",
      "required",
      "searchable",
      "multiSelect"
   ],
   "multiSelect": null
}
...
```

You can use the `updateItemType` endpoint to modify the `contactInfo` item type:

- Modify existing properties by changing the values of their attributes.
- Create custom properties by specifying their attributes.

See Settable attributes of shopper type properties for descriptions of these attributes.

Add custom properties to the `contactInfo` item type

You can use the `updateItemType` endpoint in the Commerce Admin API to add custom properties to the `contactInfo` item type. When you add a custom property to the `contactInfo` item type, the property is added to addresses in profiles, as well as in other data objects that include addresses, such as accounts and orders. The property is not added to inventory locations, however.

The ID of a custom property must include the underscore character (_). This ensures that the ID will not conflict with any properties that Commerce adds to addresses in the future. The `updateItemType` endpoint produces an error if you attempt to create a custom property without an underscore in its ID.

The following example illustrates using the `updateItemType` endpoint to add a custom property to addresses. Note that the request header must specify the `x-ccasset-language` value:

```
PUT /ccadmin/v1/itemTypes/contactInfo  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "id": "contactInfo",
  "specifications": [
    {
       "id": "sales_region",
       "label": "Sales Region",
       "type": "shortText",
       "uiEditorType": "shortText",
       "internalOnly": false,
       "required": false,
       "default": null
    }
  ]
}
```

The response includes the custom property you added:

```
...
  {
            "length": 254,
            "label": "Sales Region",
            "type": "shortText",
            "required": false,
            "searchable": false,
            "writable": true,
            "internalOnly": false,
            "uiEditorType": "shortText",
            "default": null,
            "audienceVisibility": null,
            "localizable": false,
            "textSearchable": false,
            "id": "sales_region",
            "dimension": false,
            "editableAttributes": [
                "internalOnly",
                "default",
                "audienceVisibility",
                "textSearchable",
                "label",
                "dimension",
                "required",
                "searchable",
                "multiSelect"
            ],
            "multiSelect": null
```

```
        }
...
```

Access custom properties using the `UserViewModel`

The `UserViewModel`, which is the global view model that contains a shopper's profile information, provides access to any custom profile properties you have created via the `dynamicProperties` observable array.

You can write custom widgets to retrieve the values of custom profile properties from this array, and also set the values of any custom properties you have created.

Get a custom profile property via the `UserViewModel`

To access a custom profile property from within a widget, you first create a widget-level observable in the widget's JavaScript file and then assign a value to that observable after retrieving it from the `dynamicProperties` array. In the following example, we assume that two custom profile properties have been created, `age` and `nickname`.

```
// Create the widget-level observables
age : ko.observable(),
nickname : ko.observable(),

// Iterate over the dynamicProperties array and assign the value of
the property
// with id = age to the age observable. Repeat for id = nickname.
for (var i=0; i< widget.user().dynamicProperties().length; i++){
        if (widget.user().dynamicProperties()[i].id() == 'age') {
          widget.age(widget.user().dynamicProperties()[i].value());
        } else if (widget.user().dynamicProperties()[i].id() ==
'nickname') {
          widget.nickname(widget.user().dynamicProperties()
[i].value());
        }
}
```

At this point, you can bind the widget-level observables to UI components defined in the widget's template. For example, this code snippet binds the `age` and `nickname` observables to text boxes in the widget's UI.

```
<div id="dyn-prop">
    <b>Age:</b><input type="text" name="age" id="CC-dyn-prop-age"
        aria-required="true" data-bind="value: age" ><br>
    <b>Nickname:</b><input type="text"  name="nickname" id="CC-dyn-
prop-nickname"
        aria-required="true" data-bind="value: nickname" ><br>
</div>
```

This code results in a UI that displays two text boxes that have the labels Age and Nickname and are populated with the current values of the `age` and `nickname` observables.

Set a custom profile property via the `UserViewModel`

To set the value of a custom property, your widget must update the `dynamicProperties` array in the `UserViewModel` using the current value of the widget-level observable. For example, the code below updates the dynamic property with `id=age` to the value of the `age` observable.

```
for (var i=0; i< widget.user().dynamicProperties().length; i++){
    if (widget.user().dynamicProperties()[i].id() == 'age') {
        widget.user().dynamicProperties()[i].value(widget.age());
            break;
    }
}
```

To propagate the change in the view model to the server side, the `handleUpdateProfile` function of the `UserViewModel` must be called. This function detects modifications in the observables of the `UserViewModel` and triggers a call to the `updateProfile` REST endpoint to update the data on the server. Typically, the process of making this call is triggered via clicking a Save button on the page. In the code sample below, taken from the `customerProfileDetails.template` for the Customer Profile widget, the Save button has a click binding that calls the `widget.handleUpdateProfile()` method. This method publishes a PubSub event to the `USER_PROFILE_UPDATE_SUBMIT` topic which, in turn, triggers a call to the `handleUpdateProfile()` method in the `UserViewModel`.

```
<!-- Define the Save and Cancel buttons -->
<button class="cc-button-primary col-sm-2 col-xs-12 pull-right
        cc-customer-profile-button" id="CC-customerProfile-save"
        data-bind="click: handleUpdateProfile,
        event: { mousedown: handleMouseDown, mouseup: handleMouseUp}">
  <span data-bind="widgetLocaleText: 'buttonSave'"></span>
</button>
<button class="cc-button-secondary col-sm-2 col-xs-12 pull-right
        cc-customer-profile-button" id="CC-customerProfile-cancel"
        data-bind="click: handleCancelUpdate,
        event: { mousedown: handleMouseDown, mouseup: handleMouseUp}">
  <span data-bind="widgetLocaleText: 'buttonCancel'"></span>
</button>
```

Note that the `widget.handleUpdateProfile()` method is defined in the Customer Profile widget's `customerProfile.js` file and it looks like this:

```
// Handles User profile update
widget.handleUpdateProfile = function () {

  if(widget.isUserProfilePasswordEdited()) {
    widget.user().isPasswordValid();
  }

  // Sends a PubSub message for the update
  $.Topic(PubSub.topicNames.USER_PROFILE_UPDATE_SUBMIT).publishWith(
            widget.user(),
            [{message: "success"}]
```

```
    );
};
```

6

Access SKU Properties through Widgets

In addition to SKU properties that are defined out the box, Commerce provides the ability to add custom properties to SKUs, for example, a UPC code.

Custom properties for SKUs are defined in the administration interface. They can be defined at the Base Product level, in which case all SKUs will have them, or as part of a custom product type, in which case only SKUs that are children of that product type will have them. The values for the custom properties can then be set for any given SKU. You can display out-of-the-box and custom SKU properties on your product details pages or on any page where products/SKUs are displayed, using the APIs described in this section.

Understand APIs for accessing SKU properties

SKU properties, both custom and out-of-the-box, are returned as part of the `ProductViewModel`.

Any widget that has access to the `ProductViewModel` can access SKU properties from it. Widgets that do not have access to the `ProductViewModel` can access SKU properties via the `viewModels/SkuPropertiesHandler` module.

A call to either of these APIs returns an array of SKU properties. Each item in the array consists of a property definition for a SKU property, including ID, label, type, and so on. Note that the values for the properties are returned as part of the child SKU objects, so you must make a call to one of the APIs to determine what SKU properties exist and then call the SKU objects themselves to get the values for those properties. For example, if a custom SKU property called `upcCode` exists, your widget can retrieve the value for the property for each child SKU object from the `ProductViewModel` as shown in the example below (assuming your widget has access to the `ProductViewModel`):

```
<div><span data-bind="text: product().childSKUs()[0].upcCode"></span></div>
```

As with other product properties, SKU properties can be any of five types:

- Short text
- Rich text
- Number
- Date
- Checkbox

To correctly display a SKU property's value, you must take into account the type of property being handled.

Access SKU properties from the `ProductViewModel`

Any widget that has access to the `ProductViewModel` can access SKU properties from it, for example:

```
widget.product().skuProperties()[0].label
```

This code snippet returns the label for each SKU property, both custom and out-of-the-box, defined in the SKU properties array.

Access SKU properties using `SkuPropertiesHandler`

The `viewModels/SkuPropertiesHandler` module provides access to SKU properties for any widgets that do not have access to the `ProductViewModel`. The `viewModels/SkuPropertiesHandler` module must be imported by a widget in order for the widget to call its methods, which are described below.

`SkuPropertiesHandler.getBase`

This method returns property definitions for both custom and out-of-the-box SKU properties created for the Base Product, for example:

```
viewModels/SkuPropertiesHandler.getBase(targetArray,
successCallbackFunction,errorCallbackFunction)
```

Required arguments include:

- `targetArray`: An observable array in the UI element that is populated with the properties to be displayed.
- `successCallbackFunction`: (Optional) The function that is called on success.
- `errorCallbackFunction`: (Optional) The function that is called on error.

`SkuPropertiesHandler.getCustom`

This method returns property definitions for any custom SKU properties created for the specified `productType`, for example:

```
viewModels/SkuPropertiesHandler.getCustom(targetArray,
productType,successCallbackFunction, errorCallbackFunction)
```

If there are no custom SKU properties defined, this method does not populate `targetArray`.

Required arguments include:

- `targetArray`: An observable array in the UI element that is populated with the properties to be displayed.
- `productType`: The name of the product type (for Base Product, this value is product.)
- `successCallbackFunction`: (Optional) The function that is called on success.
- `errorCallbackFunction`: (Optional) The function that is called on error.

SkuPropertiesHandler.getAll

This method returns property definitions for both custom and out-of-the-box SKU properties created for both the Base Product as well as the specified `productType`, for example:

```
viewModels/SkuPropertiesHandler.getAll(targetArray,
productType,successCallbackFunction, errorCallbackFunction)
```

Required arguments include:

- `targetArray`: An observable array in the UI element that is populated with the properties to be displayed.
- `productType`: The name of the product type (for Base Product, this value is product.)
- `successCallbackFunction`: (Optional) The function that is called on success.
- `errorCallbackFunction`: (Optional) The function that is called on error.

Create an element to display SKU properties

You can create an element that renders SKU properties, either custom or out-of-the-box, for a selected SKU. That element can then be used in a widget such as the Product Details widget on the Product Layout.

This section provides code samples for creating a sample element named `sku-properties`.

Note: Elements must be included in an extension and uploaded to Commerce before they can be used in a widget. For details on this process, see Understand widgets.

To create an element for displaying SKU properties, you need three files, `element.js`, `template.txt`, and `element.json`. The `element.js` file in this example defines a `sku-properties` element along with an `onLoad` function that, when the `sku-properties` element is loaded by a parent widget, creates an array of SKU properties for the selected SKU:

```
define(
['knockout'],
function (ko) {
  "use strict";
    return {
      // Name of the element
      elementName: 'sku-properties',

      // When the element is loaded by the widget, execute this function
      onLoad : function(widget) {
        var self = this;
        // Custom array named 'mySkuProps' that can be used within the
        // template.txt file
        self.mySkuProps = ko.computed(function() {
          var currentArray = [];
          // If and when the widget has a SKU selected
          if (widget.selectedSku()) {
            var currentSku = widget.selectedSku();
            // Access skuProperties from the product view model and
            // iterate over it to get the value for each SKU property
            for (var i=0; i < widget.product().skuProperties().length; i++) {
```

```
              var currentProperty = widget.product().skuProperties()
[i];
              if (currentSku[currentProperty.id]) {
                // Add the property ID, label, and value to the array
so
                // that they can be rendered by the template.txt
                currentArray.push({
                  'mylabel': currentProperty.label,
                  'myid': currentProperty.id,
                  'myvalue': currentSku[currentProperty.id]
                });
              }
            }
          }
          return currentArray;
        });
    }
  }
});
```

The `template.txt` file provides the HTML rendering code for the element. In this example, `template.txt` iterates over each entry in the custom SKU properties array and renders labels and values for them.

```
<div>
  <!--  ko if: initialized() && $data['sku-properties'] -->
      <!-- // Iterate over each entry in the mySkuProps array -->
      <!-- ko foreach: $data['sku-properties'].mySkuProps() -->
          <div>
              <b>
                 <!-- // Display labels, IDs, and values as needed -->
                 <span data-bind="text: mylabel"></span> (<span data-
bind="text:
                    myid"></span>) : <span data-bind="text: myvalue"></
span>
              </b>
          </div>
      <!-- /ko -->
  <!-- /ko -->
</div>
```

In this sample `element.json` meta-data file, the `sku-properties` element is made available for use by the Product Details widget:

```
{
 "inline" : true,
 "supportedWidgetType" : ["productDetails"],
 "translations" : [
 {
  "language" : "en_EN",
  "title" : "Sku Properties",
  "description" : "Displaying Sku Properties in the product details
widget"
```

```
 }
 ]
}
```

In order to use the new element in a widget, you need to add some additional tags to the widget's `display.template` and `widget.template` files that enable the element to be rendered as part of the output page and to be managed on the Design page. If the widget has already been broken into elements, you will, at a minimum, need to add an `oc` section tag for the new element:

```
<!-- oc section: sku-properties -->
    <div data-bind="element: 'sku-properties'"></div>
<!-- /oc -->
```

You may need to add other tags if the widget has not already been broken into elements. See Understand widgets for more information on the `display.template` and `widget.template` files and adding elements to them.

`SkuPropertiesHandler` example

In order to access SKU properties for a given SKU object using the `viewModels/SkuPropertiesHandler` module, you must determine the product type the SKU belongs to so it may be passed to the `viewModels/SkuPropertiesHandler` method that retrieves the property definitions.

The following widget code snippet iterates over the SKUs in a cart, determines the product type for each SKU, retrieves the SKU property definitions for that product type, and then populates an observable array with the label, ID, and values for each SKU property for each SKU in the cart.

```
// Make sure to import 'viewModels/skuPropertiesHandler' and alias
// as SkuPropertiesHandler

// Iterate over a given set of cart items (SKUs) and populate the properties
of
// each SKU
ko.utils.arrayForEach(widget.cart().items(), function(item) {
    widget.populateSkuProperties(item.productData());
});
// Sample function that populates the SKU properties
populateSkuProperties: function(item) {
    var self = this;
    // Property definition array
    var skuPropDefinition = ko.observableArray([]);
    // Variable in the item object to store properties and their values
    item.mySkuPropertyArray = ko.observableArray([]);
    // Get the product type of the SKU
    var productType = 'product';
    if (item.type) {
      productType = item.type;
    }
    // API call to get the property definition of the SKU
    SkuPropertiesHandler.getAll(skuPropDefinition, productType,
```

```
      function(skuPropDefinition) {
    // Iterate over each property of the SKU and populate the array,
which can
    // then be used in the widget template
    for (var i=0; i < skuPropDefinition().length; i++) {
      var currentProperty = skuPropDefinition()[i];
      var currentSku = item.childSKUs[0];
      if (currentSku[currentProperty.id]) {
        // Add the label, ID, value, etc to a variable that can be
accessed
        // by the widget template
        item.mySkuPropertyArray.push({
          'mylabel':currentProperty.label,
          'myid':currentProperty.id,
          'myvalue': currentSku[currentProperty.id]});
      }
    }
  });
}
```

After creating the observable array, you can use it in the widget's template to render the SKU properties on the page:

```
<!-- // Iterate over the array that was created by the
'populateSkuProperties' in widget.js -->
<!-- ko foreach: mySkuPropertyArray -->
    <div>
        <span data-bind="text: mylabel"></span> (<span data-bind="text:
          myid"></span>) : <span data-bind="text: myvalue"></span>
    </div>
<!-- /ko -->
```

7

Create Custom Promotions

You can use the Oracle Commerce Admin API to programmatically create and manage promotions, including promotion types that are not available in the UI.

This section describes how to use the Admin API to work with promotions, including creating and managing custom promotions and associating multiple promotions to a coupon. See Manage Promotions for information about working with promotions on the Marketing tab, as well as overview information about promotions. See Use the REST APIs for information you need to know before you start creating promotions.

You can also find more custom promotions information and examples in the post Custom Promotions API How-To with Examples on Oracle Cloud Customer Connect.

Understand PMDL discount rules

The Price Model Definition Language (PMDL) describes Oracle Commerce promotions internally.

PMDL describes the rules for when a promotion may apply (the condition), the rules for what may be discounted (the offer), and how to apply the discount (for example, 10% off).

This section describes the XML used for constructing discount rules that represent promotions in Oracle Commerce.

PMDL XML structure

The PMDL is used to describe promotions discount rules. The DTD defines the following to use in your PMDL rules:

- Iterators such as `next`, `up-to-and-including` and `every`.
- Quantifiers that are used in `WHEN` conditions, such as `at-least`, `at-most`, `exactly` and `all`.
- Operators such as `and`, `or` and `not`.
- Comparators such as `starts-with`, `ends-with`, `contains`, `greater-than`, `less-than and equals`, which compare values and/or arrays.
- Operations such as `union` and `anded-union`.
- Value types such as `constant` or null.
- Array types such as `constant` or `value`.

The PMDL DTD contains the following elements:

Pricing-Model element

The `pricing-model` element is the root tag for the PMDL. The `pricing-model` determines what to discount, how to discount it, and by how much.

Offer element

Every `pricing-model` requires one `offer` element. The `offer` includes one or more `discount-structure` elements, which contain detailed information about the discount and its target.

You can include more than one `discount-structure` element in an offer; this allows you to wrap multiple discounts in a single promotion.

If you have multiple discount structures within a single item promotion, you can specify the `filter-collection-name` attribute of the `offer`; this ensures that once a given item has been marked to receive a discount, it cannot receive a discount from any other `discount-structures`. If `filter-collection-name` is not set, filtering does not take place, and a given commerce item can be the target for more than one discount. The `filter-collection-name` should match the `iterator` element's `collection-name` attribute, which is normally set to `items`. Filtering is not required for single discount structures, or for non-item-based promotions.

Qualifier Element

Every `pricing-model` requires one `qualifier` element. The `qualifier` is the root tag for the promotion's `buy` condition.

Target Element

The `target` element is used in item discounts to specify the rule for selecting which items are discounted. Order and shipping discounts do not have a `target` element in the `discount-structure`.

Discount-Structure element

The `discount-structure` element has the following attributes:

- `calculator-type`: A calculator service configured in the pricing engine. For all Oracle Commerce promotions, the value for `calculator-type` is `standard`.
- `discount-type`: The calculators use this value to determine how to calculate an adjustment. Valid values are: `percentOff`, `amountOff`, `free`, and `fixedPrice`.
- `adjuster`: This optional attribute specifies the price adjustment to make for this discount. For example, the following discount structure element specifies that the promotion should discount the target by 50%:

```
<discount-structure calculator-type="standard" discount-
type="percentOff" adjuster="50"></discount-structure>
```

Attribute element

The `attribute` element allows you to add generic name/value pairs to parent tags. During PMDL parsing, the attributes and their values are placed in an attribute Map. There are also predefined iterators: `next`, `up-to-and-including` and `every`.

Iterator element

An `iterator` sorts a collection of items, then evaluate each item against one or more sub-expressions. It returns those items that match the sub-expressions.

The `iterator` element allows you to create custom iterators. Your new `iterator` element must include a `name` attribute that is unique across the PMDL.

An `iterator` element can have the following attributes and sub-elements:

- `name` attribute (required)
- `sort-by` attribute (required)
- `sort-order` attribute (required)
- `collection-name` (required)
- `element-name` (required)
- `element-quantity-property`

Quantifier element

Quantifiers evaluate a collection of items against one or more sub-expressions. Quantifiers are used in `WHEN` conditions, such as `at-least`, `at-most`, `exactly` and `all`.

The `quantifier` element allows you to create custom quantifiers. Your new `quantifier` element must include a name attribute that is unique across the PMDL.

A `quantifier` element can have the following attributes and sub-elements:

- `name` attribute (required)
- `number` attribute
- `collection-name` (required)
- `element-name` (required)
- `element-quantity-property`

Operator element

Operators return `and`, `or`, `not`, `true` or `false` based on the Boolean results from their sub-expressions.

The `operator` element allows you to create custom operators. Your new `operator` element must include a name attribute that is unique across the PMDL.

An `operator` element can specify any number of attribute sub-elements and operates on at least one comparator, operator or quantifier.

Comparator element

Comparators return `true` or `false` depending on the values of their sub-expressions. Comparators such as `starts-with`, `ends-with`, `contains`, `greater-than`, `less-than` and `equals`, compare values and/or arrays.

The `comparator` element allows you to create custom comparators. Your new `comparator` element must include a `name` attribute that is unique across the PMDL. A `comparator` element can specify any number of attribute sub-elements and must specify at least one value or array name.

Comparators evaluate using one or more sub-expressions. For example:

```
<comparator name="includes-any">
```

Comparators can also compare two value elements, and custom comparators could include any number of `value` or `constant` elements.

Value element

The `value` element returns the value of a property of the item the promotion is evaluating. You must include the `value` element for Buy One Get One promotions as follows:

```
<value>item.auxiliaryData.productRef.ancestorCategoryIds</value>
```

Constant element

The `constant` element returns a constant value against which other values can be compared. For example:

```
<constant>
 <data-type>java.lang.String</data-type>
 <string-value>xprod2147</string-value>
</constant>
```

Create a promotion

You create a `rawPmdlTemplate` promotion by using the `POST /ccadmin/v1/promotions` endpoint on the administration server.

The following table describes the body of the endpoint request. See Sample promotions for examples of requests.

Property	Description
`displayName` (required)	A string that identifies the promotion.
`description`	A string that describes the promotion. This does not appear on your site.

`priceListGroups`	An array of strings that specifies the price groups this promotion applies to. For example:
	
	`"priceListGroups": [`
	`"defaultPriceGroup",`
	`"CanadianDollar"`
	`]`
	
	If you do not include `priceListGroups` in the request, the promotion applies to all price groups. See Manage Promotions for more information.

`enabled`	A Boolean that specifies whether this promotion can be used with a qualifying order. The default value is `true`, but a promotion's availability also depends on any start and end dates you set. If `enabled` is set to `false`, the promotion cannot be used regardless of the start and end dates.

Property	Description
`priority` (required)	An integer that specifies the priority of the promotion. Promotions are applied in order of priority, with low priority numbers applied first. Oracle Commerce sorts the promotions by the value of this property. A promotion's priority is evaluated against other promotions of the same type. For example, item discounts are evaluated only against other item discounts, not against order discounts. If an order qualifies for multiple promotion types, item discounts are applied first, followed by order discounts, then shipping discounts. Promotions that are of the same type and have the same priority have no guaranteed sequence, so the order in which they are evaluated is undefined.

`startDate`	A string that specifies the date and time the promotion becomes available, formatted in ISO-8601 format as follows:
	`YYYY-MM-DD T LOCALTIME OFFSET`
	For example:
	`"startDate":"2016-03-10T00:00:00.000-05:00"`

`endDate`	A string that specifies the date and time the promotion is no longer available, formatted in ISO-8601 format as follows:
	`YYYY-MM-DD T LOCALTIME OFFSET`
	For example:
	`"endDate":"2116-03-30T00:00:00.000-05:00"`

`templateName` (required)	A string that specifies the name of the promotion template to use. For a custom promotion created with raw PMDL, the `templateName` value is `rawPmdlTemplate`.
`templatePath` (required)	A string that specifies the path to the promotion template. Supported `templatePath` values are `order`, `item`, and `shipping`.
`templateValues`	A string that specifies the template values that are used as part of the promotion to control its behavior. For `rawPmdlTemplate` promotions, the string is the XML that describes the promotion. See Understand PMDL discount rules and Sample promotions for more information.

Property	Description

`shippingMethods`	For shipping promotions, an array of strings that specifies which shipping methods can be used with the promotion. For example:
	
	`"shippingMethods": [`
	`"priorityShippingMethod",`
	`"groundShippingMethod"`
	`]`

`cardIINRanges`	A numeric range that indicates an international Issuer Identification Number (IIN) for credit cards. The number consists of the first six digits of a credit card, identifying the type of card used. By setting this range, you can present and apply promotions based on payment type to customers. You can use wildcards in this numeric property. Note that IIN ranges are not applicable when working with CyberSource and PayPal payment gateways.

If the promotion is created successfully, the response body returned includes the ID for the new promotion. For example:

```
{
 "id": "promo20014",
 "enabled": true,
 "type": 9,
 "displayName": "Spend $20 in Fan Favorites Get Order Discount"
}
```

For information on creating custom promotions with an external promotion system, refer to the Use promotions from an external system.

View promotions created with the REST API

The Promotions list on the Marketing tab in the administration interface displays all your store's promotions, including promotions created with the Admin API.

Unlike promotions created with the UI, merchandisers can view and edit only basic details for promotions created with the API.

The Marketing tab displays the following information about a promotion created with the API:

- A message that explains this promotion was created with the API and that not all promotion details are displayed here.
- The promotion's name, description, start date, end date, priority, and whether the promotion is enabled. These are the only details that you can edit on the Marketing tab.
- The type of the promotion (item discount, order discount, or shipping discount).

- Details about coupon codes and the Add Coupon Code button.

You can use the Marketing tab to add and edit coupon codes for promotions created with the API. See Manage Promotions for more information.

You cannot use the Copy button on the Marketing tab to copy a promotion created with the API.

Create custom promotions for shared carts

When creating promotions, you may want to create customized promotions for shoppers who use shared carts.

You can allow your shoppers to access multiple sites while using a single cart. Creating shared carts allows your customers to shop from any site that you add to a cart sharing group, which lets them add items to their shopping cart without having to log out of and log into different sites or manage multiple orders. For information on working with Shared Carts, refer to Create Shared Carts.

Sometimes, you may want to identify promotions that apply only to a single site that is part of a cart sharing group. Or, you may want to define site-specific promotions for multiple sites within a cart sharing group. To do this, use the Admin API to create a promotion with custom Pricing Model Definition Language (PMDL).

Create a site-specific promotion

When you create a promotion, you use PMDL to describe the promotions. PMDL provide the rules for conditions that trigger the promotion. They also indicate what may be discounted and how to apply the discount. You create a `rawPmdlTemplate` promotion by issuing a `POST` command to the `/ccadmin/v1/promotions` endpoint. For detailed information on working with PMDL, refer to Understand PMDL discount rules.

The following PMDL example shows two different pricing models. The first site-specific promotion offers a 20% discount for items that are added to the shopper's cart from the USStore site. The second site-specific promotion offers a 40% discount for items that are added to the shopper's cart from the UKStore site.

```
<pricing-model>
  <qualifier/>
  <offer>
    <discount-structure calculator-type="standard" discount-
type="percentOff" adjuster="20">
      <target>
        <iterator name="up-to-and-including" number="-1" sort-
by="priceInfo.listPrice" sort-order="ascending">
          <collection-name>items</collection-name>
          <element-name>item</element-name>
          <element-quantity-property>quantity</element-quantity-property>
          <comparator name="equals">
            <value>item.siteId</value>
            <constant>
              <data-type>java.lang.String</data-type>
              <string-value>USStore</string-value>
            </constant>
          </comparator>
        </iterator>
```

```
      </target>
    </discount-structure>
  </offer>
</pricing-model>

<pricing-model>
  <qualifier/>
  <offer>
    <discount-structure calculator-type="standard" discount-
type="percentOff" adjuster="40">
      <target>
        <iterator name="up-to-and-including" number="-1" sort-
by="priceInfo.listPrice" sort-order="ascending">
          <collection-name>items</collection-name>
          <element-name>item</element-name>
          <element-quantity-property>quantity</element-quantity-
property>
          <comparator name="equals">
            <value>item.siteId</value>
            <constant>
              <data-type>java.lang.String</data-type>
              <string-value>UKStore</string-value>
            </constant>
          </comparator>
        </iterator>
      </target>
    </discount-structure>
  </offer>
</pricing-model>
```

The request payload may be something similar to the following:

```
{
  "displayName": "20% off All US Store items",
  "description": "20% off All US Store items",
  "filterForQualifierActedAsQualifier": false,
  "filterForTargetActedAsQualifier": false,
  "priceListGroups": [
    "defaultPriceGroup"
  ],
  "enabled": true,
  "priority": "1",
  "templatePath": "item",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier/><offer><discount-structure
calculator-type=\"standard\"
         discount-type=\"percentOff\" adjuster=\"20\">
      <target><iterator name=\"up-to-and-including\" number=\"-1\"
sort-by=\"priceInfo.listPrice\"
         sort-order=\"ascending\">
      <collection-name>items</collection-name><element-name>item</
element-name>
```

```
         <element-quantity-property>quantity</element-quantity-property>
      <comparator name=\"equals\"><value>item.siteId</value><constant>
         <data-type>java.lang.String</data-type><string-value>USStore</
string-value>
      </constant></comparator></iterator></target></discount-structure></
offer>
         </pricing-model>"
    }
}
```

Or the following request:

```
{
  "displayName": "40% off All UK Store items",
  "description": "40% off All UK Store items",
  "filterForQualifierActedAsQualifier": false,
  "filterForTargetActedAsQualifier": false,
  "priceListGroups": [
    "defaultPriceGroup"
  ],
  "enabled": true,
  "priority": "1",
  "templatePath": "item",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier/><offer><discount-structure
calculator-type=\"standard\"
         discount-type=\"percentOff\" adjuster=\"40\">
      <target><iterator name=\"up-to-and-including\" number=\"-1\" sort-
by=\"priceInfo.listPrice\"
         sort-order=\"ascending\">
      <collection-name>items</collection-name><element-name>item</element-
name>
         <element-quantity-property>quantity</element-quantity-property>
      <comparator name=\"equals\"><value>item.siteId</value><constant>
         <data-type>java.lang.String</data-type><string-value>UKStore</
string-value>
      </constant></comparator></iterator></target></discount-structure></
offer>
         </pricing-model>"
    }
  }
}
```

For additional information on working with PMDL, refer to Understand PMDL discount rules.

Understand promotion filters

Promotions can be filtered, which allows you to perform a number of actions, such as exclude or include qualified items, or include items that are on sale. Filters, which are boolean properties, are based on qualifying conditions or offers presented by the promotion:

```
"displayName": "40% off All UK Store items",
  "description": "40% off All US Store items",
  "filterForQualifierActedAsQualifier": false,
  "filterForTargetActedAsQualifier": false,
```

The following filters are available for promotions:

Filter Property	Description	Used on promotions
`filterForQualifierActedAsQualifier`	This conditional filter allows you to identify if a qualifier can be reused in a promotion.	Item, Order or Shipping promotions.
`filterForQualifierDiscountedByAny`	This conditional filter allows you to determine if items discounted by any promotion can act as a qualifier for other promotions.	Item, Order or Shipping promotions.
`filterForQualifierDiscountedByCurrent`	This conditional filter determines if items that are discounted by the current promotion can act as qualifiers for other promotions.	Item promotions.
`filterForQualifierOnSale`	This conditional filter identifies if items that have a sales price can act as a qualifier.	Item, Order or Shipping promotions.
`filterForQualifierZeroPrices`	This conditional filter determines if items that have a zero price can act as a qualifier.	Item, Order or Shipping promotions.
`filterForQualifierNegativePrices`	This conditional filter identifies if items with negative prices can act as qualifiers.	Item, Order or Shipping promotions.
`filterForTargetActedAsQualifier`	This offer filter determines if items that have acted as a qualifier for any discount can be applied to the current discount.	Item promotions only.
`filterForTargetDiscountedByAny`	This offer filter determines if items that have been discounted by any promotion can receive another discount.	Item promotions only.
`filterForTargetDiscountedByCurrent`	This offer filter determines if items that have already been discounted by the current promotion can receive the discount again.	Item promotions only.

Filter Property	Description	Used on promotions
`filterForTargetOnsale`	This offer filter identifies if items that were priced with a sales price are allowed to receive the current discount.	Item promotions only.
`filterForTargetZeroPrices`	This offer filter determines if items with zero prices can receive the discount.	Item promotions only.
`filterForTargetNegativePrices`	This offer filter determines if items with negative prices can receive the discount.	Item promotions only.
`filterForTargetPriceLTOETPromoPrice`	This offer filter determines if it should exclude items with prices lower than the promotion price.	Item promotions only.

For detailed information on working with these filters, refer to the REST API documentation for Promotions.

Sample promotions

This section includes sample promotions you can create with the Admin API.

The following promotions include sample request bodies for the `POST /ccadmin/v1/promotions` endpoint.

You can find more sample promotions in Custom Promotions API How-To with Examples on Oracle Cloud Customer Connect.

Get Order Discount

This promotion automatically discounts an entire order with no spend requirements.

The following example creates a promotion that discounts an order by 10%.

```
{
  "displayName":"Get 10% off your order",
  "description":"A get order discount promotion",
  "priceListGroups":["defaultPriceGroup"],
  "enabled":true,
  "priority":"1",
  "startDate":"2016-03-10T00:00:00.000-05:00",
  "endDate":"2116-03-30T00:00:00.000-05:00",
  "templatePath": "order",
  "templateName": "rawPmdlTemplate",
  "templateValues":{"pmdl": {"xml": "<pricing-model><qualifier/><offer>
    <discount-structure calculator-type="standard" discount-type="percentOff"
    adjuster="10"/></offer></pricing-model>"}
}
}
```

Spend Y in X Get Order Discount

This promotion discounts an entire order when the shopper spends the specified amount in the specified collections.

The following example creates a promotion that discounts an order by 10% when the shopper spends $20 in the Summer Favorites collection, whose ID is cat60036.

```
{
  "displayName":"Spend $20 in Summer Favorites Get Order Discount",
  "description":"A sample spend y in x get order discount promotion",
  "priceListGroups":["defaultPriceGroup"],
  "enabled":true,
  "priority":"1",
  "startDate":"2016-03-10T00:00:00.000-05:00",
  "endDate":"2116-03-30T00:00:00.000-05:00",
  "templatePath": "order",
  "templateName": "rawPmdlTemplate",
  "templateValues":{"pmdl": {"xml": "<pricing-model><qualifier>
    <quantifier name="at-least" number="20"><collection-name>items</
collection-
    name><element-name>item</element-name><aggregator
name="spendAmount"
    operation="total"/><comparator name="includes-any">
    <value>item.auxiliaryData.productRef.ancestorCategoryIds</value>
    <constant><data-type>java.util.Set</data-type><string-
value>cat60036</string-
    value></constant></comparator></quantifier></qualifier><offer>
    <discount-structure calculator-type="standard" discount-
type="percentOff"
    adjuster="10"></discount-structure></offer></pricing-model>"}
}
}
```

Spend Y in X Get Item Discount

This promotion discounts one or more items when a shopper spends the specified amount in the specified collections. Unlike the Spend Y Get Order Discount promotion, which looks only at the total amount spent on the order to determine whether the customer qualifies for the promotion, this promotion examines the individual items in the shopping cart.

The following example creates a promotion that discounts the product Beach Umbrella by 50% when the shopper spends $10 in the Summer Favorites collection, whose ID is cat60036.

```
{
  "displayName":"Spend $10 in Summer Favorites, get 50% off a beach
umbrella",
  "description":"A sample spend y in x get item discount promotion",
  "priceListGroups":["defaultPriceGroup"],
  "enabled":true,
  "priority":"1",
  "startDate":"2016-03-10T00:00:00.000-05:00",
  "endDate":"2116-03-30T00:00:00.000-05:00",
```

```
  "templatePath": "item",
  "templateName": "rawPmdlTemplate",
  "templateValues":{"pmdl": { "xml": {"<pricing-model><qualifier><quantifier
    name="at-least" number="10"><collection-name>items</collection-
name><element-
    name>item</element-name><aggregator name="spendAmount"
    operation="total"/><comparator name="includes-any">
    <value>item.auxiliaryData.productRef.ancestorCategoryIds</value>
    <constant><data-type>java.util.Set</data-type><string-
    value>cat60036</string-value></constant></comparator>
    </quantifier></qualifier><offer><discount-structure
    calculator-type="standard" discount-type="percentOff"
    adjuster="50"><target><iterator name="up-to-and-including" number="-1"
    sort-by="priceInfo.listPrice" sort-order="ascending">
    <collection-name>items</collection-name><element-name>item</element-name>
    <aggregator name="quantity" operation="total"/>
    <comparator name="includes-any">
    <value>item.auxiliaryData.productRef.ancestorCategoryIds</value>
    <constant><data-type>java.util.Set</data-type>
    <string-value>cat60036</string-value></constant></comparator>
    </iterator></target></discount-structure></offer></pricing-model>"}
}
}
```

Spend Y in X Get Shipping Discount

This promotion offers free shipping when the shopper purchases items from a specified collection.

The following example creates a promotion that offers free shipping when a shopper buys anything from the Summer Favorites collection, whose ID is cat60036.

```
{
  "displayName":"Spend $10 in Summer Favorites, Get Free Shipping",
  "description":A sample spend y in x get shipping discount promotion,
  "priceListGroups":["defaultPriceGroup"],
  "enabled":true,
  "priority":"1",
  "startDate":"2016-03-10T00:00:00.000-05:00",
  "endDate":"2016-03-30T00:00:00.000-05:00",
  "templatePath": "shipping",
  "templateName": "rawPmdlTemplate",
  "templateValues":{"pmdl": {"xml": "<pricing-model><qualifier><quantifier
    name=\"at-least\" number=\"10\"><collection-name>items</collection-name>
    <element-name>item</element-name><aggregator name=\"spendAmount\"
    operation=\"total\"/><comparator name=\"includes-any\">
    <value>item.auxiliaryData.productRef.ancestorCategoryIds</value>
    <constant><data-type>java.util.Set</data-type>
    <string-value>cat60036</string-value></constant></comparator>
    </quantifier></qualifier> <offer><discount-structure calculator-
type=\"standard\"
    discount-type=\"fixedPrice\" adjuster=\"0\"></discount-structure>
    </offer></pricing-model>"}
}
}
```

Apply shipping discounts to individual shipping groups

Shipping discount promotions you create from templates in the administration interface apply to a shopper's entire order. However, you can use the Admin API to create promotions that discount shipping for individual, qualifying shipping groups. You can create promotions that discount shipping when a shipping group reaches a value threshold ("Spend $100, Get Free Ground Shipping") or when it contains specific items ("All Outerwear Ships Free").

The following sample request body creates a promotion that offers free ground shipping for a shipping group when it contains anything from the Summer Favorites collection, whose ID is cat60036.

```
{
  "displayName":"Free Shipping on All Your Summer Favorites",
  "description":"A sample shipping discount promotion for individual
shipping groups",
  "priceListGroups":["defaultPriceGroup"],
  "enabled":true,
  "priority":"1",
  "startDate":"2018-03-10T00:00:00.000-05:00",
  "endDate":"2018-03-30T00:00:00.000-05:00",
  "shippingMethods":"US48Ground",
  "templatePath": "shipping",
  "templateName": "rawPmdlTemplate",
  "templateValues":{"pmdl": {"xml":
    "<pricing-model>
      <qualifier>
        <quantifier name=\"at-least\" number=\"1\">
          <collection-name>shippingGroup.commerceItemRelationships</
collection-name>
          <element-name>item</element-name>
          <aggregator name=\"spendAmount\" operation=\"total\"/>
          <comparator name=\"includes-any\">
            <value>item.auxiliaryData.productRef.ancestorCategoryIds</
value>
            <constant>
              <data-type>java.util.Set</data-type>
              <string-value>cat60036</string-value>
            </constant>
          </comparator>
        </quantifier>
      </qualifier>
      <offer>
        <discount-structure calculator-type=\"standard\" discount-
type=\"fixedPrice\" adjuster=\"0.0\">
        </discount-structure>
      </offer>
    </pricing-model>"}}
}
```

The following sample request body creates a promotion that offers free two-day shipping when a shipping group contains $100 in merchandise.

```
{
  "displayName":"Spend $100, Get Free 2 Day Shipping",
  "description":"A sample shipping discount promotion for individual
shipping groups",
  "priceListGroups":["defaultPriceGroup"],
  "enabled":true,
  "priority":"1",
  "startDate":"2018-03-10T00:00:00.000-05:00",
  "endDate":"2018-03-30T00:00:00.000-05:00",
  "shippingMethods":"US48TwoDay",
  "templatePath": "shipping",
  "templateName": "rawPmdlTemplate",
  "templateValues":{"pmdl": {"xml":
    "<pricing-model>
      <qualifier>
        <greater-than>
          <value>shippingGroup.priceInfo.itemSubtotal</value>
          <constant>
            <data-type>java.lang.Double</data-type>
            <string-value>100.0</string-value>
          </constant>
        </greater-than>
      </qualifier>
      <offer>
        <discount-structure calculator-type=\"standard\" discount-
type=\"fixedPrice\" adjuster=\"0.0\">
        </discount-structure>
      </offer>
    </pricing-model>"}}
}
```

The following examples show how you could extend the PDML to make customized promotions

Get 15% off order when you buy both product X and Y

This sample request body creates a promotion that takes 15% off of an order when the customer purchase both of the specified products:

```
{
  "displayName": "Spend $100 in Category X and must buy both product Y and Z
from category X get 15% Off Order",
  "description": "A sample 15% off order promotion",
  "enabled":true,
  "priority":"1",
  "templatePath": "order",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier><anded-union><iterator name=\"next\"
number=\"1\"
         sort-by=\"priceInfo.listPrice\" sort-
```

```
order=\"descending\"><collection-name>items</collection-name>
         <element-name>item</element-name><aggregator
name=\"quantity\" operation=\"total\"/><equals>
         <value>item.auxiliaryData.productId</value><constant><data-
type>java.lang.String</data-type>
         <string-value>Product_36Exy</string-value></constant></
equals></iterator>
         <iterator name=\"next\" number=\"1\" sort-
by=\"priceInfo.listPrice\" sort-order=\"descending\">
         <collection-name>items</collection-name><element-name>item</
element-name>
         <aggregator name=\"quantity\" operation=\"total\"/><equals>
         <value>item.auxiliaryData.productId</value><constant><data-
type>java.lang.String</data-type>
         <string-value>Product_27Fxyzi</string-value></constant></
equals></iterator>
         <iterator name=\"next\" number=\"100\" sort-
by=\"priceInfo.listPrice\" sort-order=\"ascending\">
         <collection-name>items</collection-name><element-name>item</
element-name>
         <aggregator name=\"spendAmount\" operation=\"total\" /
><includes>
         <value>item.auxiliaryData.productRef.ancestorCategoryIds</
value><constant>
         <data-type>java.lang.String</data-type><string-
value>Category_14Bxi</string-value></constant>
         </includes></iterator></anded-union></qualifier><offer>
         <discount-structure calculator-type=\"standard\" discount-
type=\"percentOff\" adjuster=\"15\"/>
         </offer></pricing-model>"
    }
  }
}
```

Spend $100 in Category X get 15% off order

This sample request body creates a promotion where, once the shopper has spent $100 in Category X and have bought both product Y and Z from Category X, they receive 15% off their order:

```
{
  "displayName": "Buy both product Y and Z from category X. Get 15%
Off Order",
  "description": "A sample promotion for 15% off an order",
  "enabled":true,
  "priority":"1",
  "templatePath": "order",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier><anded-union><iterator
name=\"next\" number=\"1\"
         sort-by=\"priceInfo.listPrice\" sort-
order=\"descending\"><collection-name>items</collection-name>
         <element-name>item</element-name><aggregator
```

```
name=\"quantity\" operation=\"total\"/><equals>
         <value>item.auxiliaryData.productId</value><constant><data-
type>java.lang.String</data-type>
         <string-value>Product_36Exy</string-value></constant></equals>
         </iterator><iterator name=\"next\" number=\"1\" sort-
by=\"priceInfo.listPrice\"
         sort-order=\"descending\"><collection-name>items</collection-
name><element-name>item</element-name>
         <aggregator name=\"quantity\" operation=\"total\"/><equals>
         <value>item.auxiliaryData.productId</value><constant><data-
type>java.lang.String</data-type>
         <string-value>Product_27Fxyzi</string-value></constant></equals></
iterator></anded-union>
         </qualifier><offer><discount-structure calculator-type=\"standard\"
discount-type=\"percentOff\"
         adjuster=\"15\"/></offer></pricing-model>"
    }
  }
}
```

Spend $200 in Category X or Category XX and get 15% off order

This sample request body creates a promotion where the promotion is extended after the shopper has spent $200 in either Category X or Y and has purchased both products Y and Z from Category X.

```
{
  "displayName": "Spend $200 in Category X or Category XX and must buy both
product Y and Z from category X. Get 15% Off Order",
  "description": "An example promotion for 15% off order",
  "enabled":true,
  "priority":"1",
  "templatePath": "order",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier><and><at-least
number=\"200\"><collection-name>items</collection-name>
         <element-name>item</element-name><aggregator name=\"spendAmount\"
operation=\"total\" />
         <comparator name=\"includes-
any\"><value>item.auxiliaryData.productRef.ancestorCategoryIds</value>
         <constant><data-type>java.util.Collection</data-type><string-
value>Category_14Bxi</string-value>
         <string-value>Category_35Exy</string-value></constant></
comparator></at-least><at-least number=\"1\">
         <collection-name>items</collection-name><element-name>item</element-
name><aggregator name=\"quantity\"
         operation=\"total\"/><equals><value>item.auxiliaryData.productId</
value><constant>
         <data-type>java.lang.String</data-type><string-value>Product_36Exy</
string-value></constant></equals>
         </at-least><at-least number=\"1\"><collection-name>items</
collection-name>
         <element-name>item</element-name><aggregator name=\"quantity\"
```

```
operation=\"total\"/><equals>
         <value>item.auxiliaryData.productId</value><constant><data-
type>java.lang.String</data-type>
         <string-value>Product_27Fxyzi</string-value></constant></
equals></at-least></and></qualifier><offer>
         <discount-structure calculator-type=\"standard\" discount-
type=\"percentOff\" adjuster=\"15\"/>
         </offer></pricing-model>"
    }
  }
}
```

Group item discount

The following sample request body allows you to offer a group of products at a discount or fixed price. This type of promotion applies only to item discount promotions, where the customer receives a discount on an individual item or multiple items. This example extends 30% off of the item when you buy something from Category_14Nxi and Category_35Exy:

```
{
  "displayName": "Group Discount - Buy something from Category_14Bxi
and Category_35Exy and get 30% off",
  "description": "A sample promotion for 30% off for group discount",
  "enabled": true,
  "priority": "1",
  "templatePath": "item",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier /><offer><discount-structure
calculator-type=\"group\"
         discount-type=\"percentOff\" adjuster=\"30\"><target><group-
iterator name=\"every-group\">
         <anded-union><iterator name=\"next\" number=\"1\" sort-
by=\"priceInfo.listPrice\"
         sort-order=\"ascending\"><collection-name>items</collection-
name><element-name>item</element-name>
         <aggregator name=\"quantity\" operation=\"total\"/
><comparator name=\"includes-any\">
         <value>item.auxiliaryData.productRef.ancestorCategoryIds</
value><constant>
         <data-type>java.util.Set</data-type><string-
value>Category_14Bxi</string-value></constant>
         </comparator></iterator><iterator name=\"next\" number=\"1\"
sort-by=\"priceInfo.listPrice\"
         sort-order=\"descending\"><collection-name>items</collection-
name><element-name>item</element-name>
         <aggregator name=\"quantity\" operation=\"total\"/
><comparator name=\"includes-any\">
         <value>item.auxiliaryData.productRef.ancestorCategoryIds</
value><constant>
         <data-type>java.util.Set</data-type><string-
value>Category_35Exy</string-value></constant>
         </comparator></iterator></anded-union></group-iterator></
```

```
target></discount-structure></offer>
         </pricing-model>"
    }
  }
}
```

BOGO Gift With Purchase

The following sample request body allows you to create a promotion that combines both BOGO and Gift With Purchase promotions, which gives a shopper who buys a camera a free gift card SKU. This promotion adds a specific SKU as the gift, and automatically adds the SKU to the cart. This promotion loops, so every time the shopper meets the conditions, they receive the gift. Note that BOGO promotions loop by default, but Gift With Purchase promotions do not.:

```
{
  "displayName": "BOGO GWP",
  "description": "1 gift for 1",
  "priceListGroups": [
    "defaultPriceGroup"
  ],
  "enabled": true,
  "priority": "1",
  "templatePath": "item",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier><iterator name=\"next\" number=\"1\"
sort-by=\"priceInfo.listPrice\"
         sort-order=\"ascending\"><collection-name>items</collection-
name><element-name>item</element-name>
         <aggregator name=\"quantity\" operation=\"total\"/><comparator
name=\"includes-any\">
         <value>item.auxiliaryData.productRef.ancestorCategoryIds</
value><constant>
         <data-type>java.util.Set</data-type><string-value>cameracat</string-
value></constant></comparator>
         </iterator></qualifier><offer><discount-structure calculator-
type=\"gwp\" discount-type=\"fixedPrice\"
         adjuster=\"0\" process-zero-targets=\"true\"><discount-detail
name=\"gift\">
         <attribute name=\"quantity\" value=\"1\"/><attribute
name=\"autoRemove\" value=\"true\"/>
         <attribute name=\"giftDetail\" value=\"giftCardsku_1\"/><attribute
name=\"giftType\" value=\"sku\"/>
         </discount-detail> <target><iterator name=\"up-to-and-including\"
number=\"1\"
         sort-by=\"priceInfo.listPrice\" sort-
order=\"ascending\"><collection-name>items</collection-name>
         <element-name>item</element-name><element-quantity-
property>quantity</element-quantity-property>
         <comparator
name=\"equals\"><value>item.auxiliaryData.catalogRef.id</value><constant>
         <data-type>java.lang.String</data-type><string-value>giftCardsku_1</
string-value></constant>
```

```
        </comparator></iterator></target></discount-structure></
offer></pricing-model>"
    }
  }
}
```

Pick two items from a predefined selection for a fixed price

The following example displays a promotion that is triggered when two items are selected. Note that this promotion iterate, allowing the shopper to by 2 of the items for $15, 4 of the items for $30, etc.:

```
{
  "displayName": "$15 for every 2 soaps",
  "description": "$15 for every 2 soaps",
  "priceListGroups": [
    "defaultPriceGroup"
  ],
  "enabled": true,
  "priority": "1",
  "templatePath": "item",
  "templateName": "rawPmdlTemplate",
  "templateValues": {
    "pmdl": {
      "xml": "<pricing-model><qualifier/><offer><discount-structure
adjuster=\"7.5\"
      calculator-type=\"standard\" discount-
type=\"fixedPrice\"><target><every-group><union>
      <iterator name=\"next\" number=\"2\" sort-
by=\"priceInfo.listPrice\" sort-order=\"ascending\">
      <collection-name>items</collection-name><element-name>item</
element-name>
      <aggregator name=\"quantity\" operation=\"total\"/><comparator
name=\"includes-any\">
      <value>item.auxiliaryData.productRef.ancestorCategoryIds</
value><constant><data-type>java.util.Set</data-type>
      <string-value>SOAPS</string-value></constant></comparator></
iterator></union></every-group></target>
      </discount-structure></offer></pricing-model>"
    }
  }
}
```

PMDL Example

The following is an example of the PMDL DTD file used by Commerce.

This is an example of the DTD file that defines the promotion attributes:

```
<!--
This DTD defines the circumstances under which a pricing model should
be applied

Categories of expression elements:
```

```
o iterators   = next, up-to-and-including, every, iterator
o operators   = and, or, not, operator
o operations  = anded-union
o comparators = starts-with, ends-with, contains, greater-than, less-than,
equals, comparator
o quantifiers = at-least, at-most, exactly, all, quantifier
o value-names = value, constant, null
o array-names = array, value
-->

<!ENTITY % iterators "next | up-to-and-including | every | iterator">
<!ENTITY % group-iterators "every-group | up-to-and-including-group | group-
iterator">
<!ENTITY % operators "and | or | not | operator">
<!ENTITY % operations "union | anded-union">
<!ENTITY % comparators "starts-with | ends-with | contains | equals |   not-
equals | less-than | less-than-or-equals |
  greater-than | greater-than-or-equals | includes | includes-any | includes-
all | isoneof | isnotoneof | comparator">
<!ENTITY % quantifiers "exactly | at-least | at-most | all | quantifier">

<!ENTITY % value-types "value | constant | null">
<!ENTITY % array-types "value | constant | array">



<!-- Root element of the pricing-model. -->
<!ELEMENT pricing-model (qualifier, (target?|offer))>

<!ELEMENT offer (attribute*, discount-structure+)>
<!ATTLIST offer
  filter-collection-name CDATA #IMPLIED
  quantity-property-name CDATA "quantity">

<!ELEMENT qualifier (attribute*, ((%quantifiers;) | (%operators;) |
(%operations;) | (%comparators;) | (%iterators;))?) >

<!ELEMENT target (attribute*, ((%operations;) | (%iterators;) | (%group-
iterators;))?) >

<!ELEMENT discount-structure (attribute*, discount-detail*, target?)>
<!ATTLIST discount-structure
  calculator-type CDATA #REQUIRED
  discount-type CDATA #REQUIRED
  adjuster CDATA #IMPLIED
  process-zero-targets CDATA "false">

<!ELEMENT discount-detail (attribute*)>
<!ATTLIST discount-detail
  name CDATA #IMPLIED>

<!-- generic name/value pair used to add custom attributes and values to
many elements -->
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
```

```
  name CDATA #REQUIRED
  value CDATA #REQUIRED>

<!-- used by iterators and quantifiers -->
<!ELEMENT element-name (#PCDATA)>

<!-- used to fully qualify a property within the environment of
objects.  Use dot-notation: item.sku.tracks.songTitle
or in the case of attributes just a value -->
<!ELEMENT value (#PCDATA)>

<!-- used in quantifiers and iterators to specify over which
collection the element is iterating -->
<!ELEMENT collection-name (#PCDATA)>

<!-- specifies the property which is the 'quantity' of an iterator or
quantifier element -->
<!ELEMENT element-quantity-property (#PCDATA)>

<!ELEMENT aggregator (attribute*)>
<!ATTLIST aggregator
  name CDATA #REQUIRED
  operation CDATA #REQUIRED>

<!--
Iterators
Evaluate expression elements
Built-in iterators:
o    next: evaluate next n expression elements
o    up-to-and-including: evaluate up to n expression elements
(inclusive)
o    every: evaluate all expression elements
-->

<!-- returns up to and including <number> items. It does this process
exactly once the sort is performed before the items are selected -->
<!ELEMENT up-to-and-including (
  attribute*,
  collection-name,
  element-name,
  (element-quantity-property | aggregator)?,
  (%comparators; | %operators; | %quantifiers;)?
)>
<!ATTLIST up-to-and-including
  number     CDATA #REQUIRED
  sort-by    CDATA #REQUIRED
  sort-order CDATA #REQUIRED>

<!-- consumes every item that matches its rule. -->
<!ELEMENT every (
  attribute*,
  collection-name,
  element-name,
  (%comparators; | %operators; | %quantifiers;)?
)>
```

```
<!ATTLIST every
  sort-by    CDATA #IMPLIED
  sort-order CDATA #IMPLIED>

<!-- consumes the next N things that match its rule.  If it doesn't find N
things, it -->
<!-- fails.  the sort is performed before the items are selected -->
<!ELEMENT next (
  attribute*,
  collection-name,
  element-name,
  (element-quantity-property | aggregator)?,
  (%comparators; | %operators; | %quantifiers;)?
)>
<!ATTLIST next
  number     CDATA #REQUIRED
  sort-by    CDATA #REQUIRED
  sort-order CDATA #REQUIRED>

  <!-- custom iterator can be used for extending the set of built-ins -->
<!ELEMENT iterator (
  attribute*,
  collection-name,
  element-name,
  (element-quantity-property | aggregator)?,
  (%comparators; | %operators; | %quantifiers;)?
)>
<!ATTLIST iterator
  name       CDATA #REQUIRED
  number     CDATA #IMPLIED
  sort-by    CDATA #REQUIRED
  sort-order CDATA #REQUIRED>

<!--
  Group Iterators
  Evaluate expression elements
  Built-in iterators:
  o    every-group: evaluate all expression elements
  o    up-to-and-including-group: evaluate up to n expression elements
(inclusive)
  -->

  <!-- consumes every group returned by the sub-expression until no more are
found -->
<!ELEMENT every-group (
  attribute*,
  (%operations;)
)>

  <!-- consumes the next N group groups returned by the sub-expression -->
<!ELEMENT up-to-and-including-group (
  attribute*,
  (%operations;)
)>
<!ATTLIST up-to-and-including-group
```

```
  number     CDATA #REQUIRED>

  <!-- custom group iterator can be used for extending the set of
built-ins -->
<!ELEMENT group-iterator (
  attribute*,
  (%operations;)
)>
<!ATTLIST group-iterator
  name       CDATA #REQUIRED
  number     CDATA #IMPLIED>


<!--
Quantifiers
Evaluate if a certain quantity of expression elements evaluate to true.
Built-in quantifiers:
o    exactly: evaluates true if exactly n expression elements evaluate
to true
o    at-least: evaluates true if at least n expression elements
evaluate to true
o    at-most: evaluates true if at most n expression elements evaluate
to true
o    all: evaluates true if all expression elements evaluate to true
-->
<!ELEMENT exactly (
  attribute*,
  collection-name,
  element-name,
  (element-quantity-property | aggregator)?,
  (%comparators; | %operators; | %quantifiers;)?
)>
<!ATTLIST exactly
  number CDATA #REQUIRED>

<!ELEMENT at-least (
  attribute*,
  collection-name,
  element-name,
  (element-quantity-property | aggregator)?,
  (%comparators; | %operators; | %quantifiers;)?
)>
<!ATTLIST at-least
  number CDATA #REQUIRED>

<!ELEMENT at-most (
  attribute*,
  collection-name,
  element-name,
  (element-quantity-property | aggregator)?,
  (%comparators; | %operators; | %quantifiers;)?
)>
<!ATTLIST at-most
  number CDATA #REQUIRED>
```

```
<!ELEMENT all (
  attribute*,
  collection-name,
  element-name,
  (%comparators; | %operators; | %quantifiers;)?
)>

<!-- custom quantifier can be used for extending the set of built-ins -->
<!ELEMENT quantifier (
  attribute*,
  collection-name,
  element-name,
  (element-quantity-property | aggregator)?,
  (%comparators; | %operators; | %quantifiers;)?
)>
<!ATTLIST quantifier
  name   CDATA #REQUIRED
  number CDATA #IMPLIED>

<!--
Operators
Perform boolean logic operations on expression elements
Built-in operators:
o    and: conjunction operator; evaluates true if both expression elements
evaluate to true
o    or: disjunction operator; evaluates true if either or both expression
elements evaluate to true
o    not: complement operator; negates the value of the expression element
-->
<!ELEMENT and (
  attribute*,
  (%comparators; | %operators; | %quantifiers;),
  (%comparators; | %operators; | %quantifiers;)*
)>

<!ELEMENT or (
  attribute*,
  (%comparators; | %operators; | %quantifiers;),
  (%comparators; | %operators; | %quantifiers;)*
)>

<!ELEMENT not (
  attribute*,
  (%comparators; | %operators; | %quantifiers;)
)>

<!-- custom operator can be used for extending the set of built-ins -->
<!ELEMENT operator (
  attribute*,
  (%comparators; | %operators; | %quantifiers;)+
)>
<!ATTLIST operator
  name CDATA #REQUIRED>

<!--
```

```
Operations
Perform mathematical operations on expression elements
Built-in operators:
o    union: a collection of items that are returned from each
expression
-->
<!ELEMENT union (
  attribute*,
  (%iterators;),
  (%iterators;)*
)>

<!-- anded-union: a collection of items that are returned from each
expression if all iterators match -->
<!ELEMENT anded-union (
  attribute*,
  (%iterators;),
  (%iterators;)*
)>

<!--
Comparators
Compare values and/or array with each other
Built-in comparators:
o    starts-with: evaluates true if a value starts with another value
o    ends-with: evaluates true if a value ends with another value
o    contains: evaluates true if a value contains another value

o    equals: evaluates true if a value ==  another value
o    not-equals: evaluates true if a value != another value
o    less-than: evaluates true if a value is < another value
o    less-than-or-equals: evaluates true if a value <= another value
o    greater-than: evaluates true if a value > another value
o    greater-than-or-equals: evaluates true if a value >= another value

o    includes: evaluates true if an array/value includes a value
o    includes-any: evaluates true if an array/value includes any value
in another array/value
o    includes-all: evaluates true if an array/value includes all
values in another array/value
o    isoneof: evaluates true if a value is a member of an array/value
o    isnotoneof: evaluates true if a value is not a member of an array/
value
-->
<!ELEMENT starts-with
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT ends-with
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT contains
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT equals
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT not-equals
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT less-than
```

```
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT less-than-or-equals
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT greater-than
  ( attribute*, (%value-types;), (%value-types;) )>
<!ELEMENT greater-than-or-equals
  ( attribute*, (%value-types;), (%value-types;) )>

<!ELEMENT isoneof
  ( attribute*, (%value-types;), (%array-types;) )>
<!ELEMENT isnotoneof
  ( attribute*, (%value-types;), (%array-types;) )>
<!ELEMENT includes
  ( attribute*, (%array-types;), (%value-types;) )>
<!ELEMENT includes-any
  ( attribute*, (%array-types;), (%array-types;) )>
<!ELEMENT includes-all
  ( attribute*, (%array-types;), (%array-types;) )>

<!-- custom comparator can be used for extending the set of built-ins -->
<!ELEMENT comparator ( attribute*, (%value-types; | %array-types;)+ )>
<!ATTLIST comparator
  name CDATA #REQUIRED>

<!--
Value
Represents data (or empty)
o    value: parsed character data
o    constant: a data-type/string-value pair
o    null: an empty value
-->
<!ELEMENT constant (data-type, string-value+)>
<!ELEMENT array (%value-types;)+>
<!ELEMENT null EMPTY>

<!ELEMENT data-type (#PCDATA)>
<!ELEMENT string-value (#PCDATA)>
```

Create custom properties for promotions

This section describes how to use the Oracle Commerce REST web services APIs to add custom properties to promotions.

The Use the REST APIs section contains information you should be familiar with before creating custom properties.

Understand item types

Like shopper profiles, commerce items and orders, promotions include a predefined set of properties. Promotion properties are determined by an item type.

Promotions use the `/itemTypes/{id}` administration endpoint, where the ID is a repository item type that supports customizable properties. When you use `PUT` with the endpoint, the endpoint uses attributes whose names are configured for each item type. In the case of custom properties for promotions, the `itemType` is promotion.

You cannot create additional item types, but you can add custom properties to the `promotion` item type. For example, you could add a custom property that an administrator uses to indicate additional promotion details.

View a promotion

Promotions can be created with the Oracle Commerce REST Admin APIs by using the `createPromotion` endpoint, or updated with the `updatePromotion` endpoint. In both cases, custom properties appear in the request body alongside the other predefined properties.

You can use the Oracle Commerce Admin API to retrieve promotion information. Issue a `GET` request to `/ccadmin/v1/promotions` to display all promotions. For example:

```
GET /ccadmin/v1/promotions HTTP/1.1
Authorization: Bearer <access_token>
```

To view a specific promotion, issue a `GET` request to the `/ccadmin/v1/promotions/{id}` endpoint, providing the ID of the promotion you want to view.

The response displays a list of promotions and a subset of properties for each promotion. You can modify the values of these properties using the `PUT /ccadmin/v1/promotions/{id}` endpoint on the administration server.

Create a promotion custom property

To add custom properties to a promotion, issue a `PUT` request to the `/ccadmin/v1/itemTypes/promotion` endpoint on the administration server.

The Item Types resource in the administration API includes endpoints for creating and working with custom properties of the item type. The items resource in the Admin API includes endpoints that you can use to set the values of properties of individual promotions, including custom properties that have been added to the promotion item type.

The `promotion` item type uses some of the attributes used by all non-product item types, which are `type`, `uiEditorType`, `label`, `default`, `internalOnly`, `required`, and `localizable`. Note that the `localizable` attribute is rejected by other item types. An attribute that is specific to the `promotion` item type is the `includeInDiscountInfoJson` attribute. When you define a custom property for a promotion, the `includeInDiscountInfoJson` attribute, which is set by default to `true`, indicates if the custom property will be added automatically along with the `promotionId`, `promotionDesc`, and `promotionLevel` values that are included in `discountInfo` for each promotion. This attribute, which is a boolean attribute, can be set when creating or updating the custom property.

When you add a custom property to the `promotion` item type, the property is added to all promotions, including any new promotions. Note that changes to an individual promotion must be selected for publication and then published. Custom property definitions are automatically included in the publishing process. For additional information on publishing, refer to Publish Changes.

The ID of a custom property must include the underscore character (_). This ensures that the ID will not conflict with any properties that Commerce adds to promotions in the future. The endpoint produces an error if you attempt to create a custom property without an underscore in its ID.

The following is a sample request to create a custom property named `promotion_campaign`. Note that `shortText` is the only value supported for the `type` attribute of a promotion property:

```
{
  "specifications": [
    {
      "id": "promotion_campaign",
      "label": "Campaign",
      "type": "shortText",
      "required": false,
      "uiEditorType": "shortText",
      "localizable": false,
      "includeInDiscountInfoJson": true
    }
  ]
}
```

The following is an example response with the new `promotion_campaign` custom property:

```
{
  "propertiesOrder": [
    "upsell",
    "global",
    "displayName",
    "isSiteRestricted",
    "filterForQualifierActedAsQualifier",
    "filterForQualifierDiscountedByAny",
    "template",
    "giveToAnonymousProfiles",
    "filterForQualifierOnSale",
    "beginUsable",
    "startDate",
    "filterForQualifierZeroPrices",
    "endDate",
    "priority",
    "endUsable",
    "relativeExpiration",
    "description",
    "timeUntilExpire",
    "filterForQualifierNegativePrices",
    "allowMultiple",
    "uses",
    "enabled",
    "pmdlVersion",
    "evaluationLimit",
    "pmdlRule",
    "promotion_campaign"
  ],
  "displayName": "Promotion",
  "links": [
    {
      "rel": "self",
      "href": "http://my.website.com:9080/ccadmin/v1/itemTypes/promotion"
    }
```

```
],
  ...
  {
    "internalOnly": false,
    "uiEditorType": "shortText",
    "default": null,
    "length": 254,
    "includeInDiscountInfoJson": true,
    "localizable": false,
    "label": "Campaign",
    "id": "promotion_campaign",
    "type": "shortText",
    "editableAttributes": [
      "internalOnly",
      "default",
      "includeInDiscountInfoJson",
      "label",
      "required"
    ],
    "required": false,
    "writable": true
  },
  {
    "internalOnly": false,
    "uiEditorType": "number",
    "default": 1,
    "length": 10,
    "includeInDiscountInfoJson": false,
    "localizable": false,
    "label": "Max uses per customer",
    "id": "uses",
    "type": "number",
    "editableAttributes": [
      "internalOnly",
      "default",
      "includeInDiscountInfoJson",
      "label",
      "required"
    ],
    "required": false,
    "writable": true
  },
  {
    "internalOnly": false,
    "uiEditorType": "date",
    "default": null,
    "length": 7,
    "includeInDiscountInfoJson": false,
    "localizable": false,
    "label": "Distribute starting",
    "id": "startDate",
    "type": "date",
    "editableAttributes": [
      "internalOnly",
      "default",
```

```
        "includeInDiscountInfoJson",
        "label",
        "required"
      ],
      "required": false,
      "writable": true
    }
  ]
}
```

The following is an example of a request for creating a promotion with the `promotion_campaign` custom property:

```
{
  "priceListGroups" : [ "defaultPriceGroup" ],
  "endDate" : null,
  "templateName" : "tieredOrderDiscount",
  "displayName" : "10% off over $100",
  "templateValues" : {
    "discountStructure" : {
      "discount_details" : [ {
        "spend_value" : "100.00",
        "discount_value" : "10.00"
      } ],
      "discount_type_value" : "percentOff"
    }
  },
  "promotion_campaign" : "onlineOnly",
  "sites" : [ ],
  "priority" : 1,
  "promotionId" : null,
  "startDate" : null,
  "excludedPromotions" : [ ],
  "templatePath" : "order"
}
```

The following is the request response:

```
{
  "template": "/order/tieredOrderDiscount.pmdt",
  "dynamicPropertyMapLong": {},
  "endDate": null,
  "displayName": "10% off over $100",
  "templateValues": {
    "discountStructure": {
      "discount_details": [
        {
          "spend_value": "100.00",
          "discount_value": "10.00"
        }
      ],
      "discount_type_value": "percentOff"
    }
  },
```

```
  "description": null,
  "global": true,
  "sites": [],
  "type": 9,
  "enabled": true,
  "parentFolder": null,
  "priceListGroups": [
    "defaultPriceGroup"
  ],
  "includedPromotions": [],
  "dynamicPropertyMapString": {
    "promotion_campaign": "onlineOnly"
  },
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:9080/ccadmin/v1/promotions/promo10001"
    }
  ],
  "paymentTypes": [],
  "id": "promo10001",
  "filterForQualifierActedAsQualifier": null,
  "evaluationLimit": -1,
  "shippingMethods": [],
  "dynamicPropertyMapBigString": {},
  "priority": 1,
  "excludedPromotions": [],
  "repositoryId": "promo10001",
  "promotion_campaign": "onlineOnly",
  "stackingRule": null,
  "dynamicPropertyMapDouble": {},
  "startDate": null
}
```


Update an existing custom property

To update a custom property issue a `PUT` request to the `/ccadmin/v1/itemTypes/promotion` endpoint on the administration server with the ID of the existing custom property. The following example changes the label displayed for the `promotion_campaign` property:

```
{
"id": "promotion",   //this is the ID of the item-type item
"specifications": [
    {
      "id": "promotion_campaign",  //this is the ID of the property
      "label": "Discount Source"
    }
]
}
```

Assign and manage coupons

The Commerce UI allows merchandisers to assign only a single coupon or coupon batch to each promotion. However, you can use the Oracle Commerce Admin API to associate multiple promotions with a coupon or coupon batch.

This section describes how to use the Oracle Commerce Admin API to associate multiple promotions with a coupon or coupon batch.

Assign multiple promotions to a coupon or coupon batch

You must use the Oracle Commerce Admin API to assign multiple promotions to a coupon or coupon batch. You cannot perform these tasks on the Marketing tab in the administration interface.

You can assign promotions to a new coupon or batch when you create it or you can update an existing coupon or batch so that it applies to multiple promotions:

- To assign multiple promotions to a coupon when you create it, issue a `POST` request to `/ccadmin/v1/claimables`.
- To assign multiple promotions to a coupon batch when you create it, issue a `POST` request to `/ccadmin/v1/couponBatches`.
- To associate promotions with an existing coupon, issue a `PUT` request to `/ccadmin/v1/claimables/{id}`.
- To associate promotions with an existing coupon batch, issue a `PUT` request to `/ccadmin/v1/couponBatches/{id}`.

Keep the following points in mind when assigning promotions to coupons and coupon batches:

- The promotions you want to assign to a coupon or coupon batch must already exist; simply specifying a promotion ID when you create or update a coupon or coupon batch does not automatically create the promotion.
- To assign additional promotions to an existing coupon or coupon batch that already has promotions assigned to it, include both the new promotions and the existing promotions in the `PUT` request. To replace the existing promotions with new promotions, include only the new promotions in the `PUT` request.
- Each coupon or coupon batch has a `startDate` property that specifies the date and time the coupon or batch is available for use, for example, to be associated with shopper profiles. You can specify a start date when you create a coupon or coupon batch. By default, if a newly-created coupon or batch is associated with only one promotion, its `startDate` value is the same as the `startDate` value of the promotion. If a coupon or batch is associated with multiple promotions, the value of its `startDate` property is null.

You can change the value of the `startDate` for a coupon or coupon batch with a `PUT` request to `/ccadmin/v1/claimables/{id}` or `/ccadmin/v1/couponBatches/{id}`.

Note: If you update a promotion, either through the Marketing tab in the administration interface or with the Admin API, the value of the `startDate` property of any coupons or batches associated with the updated promotion is automatically reset to the `startDate` value of the updated promotion. If you update a promotion whose child coupons have a different `startDate` value than the promotion itself, remember to reset the `startDate` value of each child coupon or batch to avoid unexpected behavior.

The following example creates a new coupon that applies to two promotions:

```
POST /ccadmin/v1/claimables HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
   "promotions":[{"repositoryId":"promo10001"},
{"repositoryId":"promo10003"}],
   "id": "MYSALE",
   "maxUses": "3"
}
```

If the coupon is created successfully, the response body returned includes the ID for the new coupon:

```
{
  "repositoryId": "MYSALE"
}
```

The following example shows a sample request that adds four promotions to an existing coupon batch.

```
PUT /ccadmin/v1/couponBatches/{SAVE2a462ON50} HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
"promotions": [
   {"repositoryId": "promo10001"},
   {"repositoryId": "promo10002"},
   {"repositoryId": "promo10003"},
   {"repositoryId": "promo10006"}]
}
```

The following shows the response body returned:

```
{
  "id": "SAVE2a462ON50",
  "numberOfCoupons": 100,
  "maxUses": 8,
  "numberClaimed": 0,
  "prefix": "Summer Sale",
  "promotions": [
    {"repositoryId": "promo10001"},
    {"repositoryId": "promo10002"},
    {"repositoryId": "promo10003"},
    {"repositoryId": "promo10006"}
  ],
  "uses": 0,
  "repositoryId": "SAVE2a462ON50"
}
```

Remove a coupon or coupon batch from a promotion

You remove a coupon or coupon batch from one or more promotions by issuing a `DELETE` request:

- To remove a coupon from one or more promotions, issue a `DELETE` request to `/ccadmin/v1/claimables/{id}`.
- To remove a coupon batch from one or more promotions, issue a `DELETE` request to `/ccadmin/v1/couponBatches/{id}`.

The body of the request must include the IDs for the promotions to remove the coupon from. If the coupon is still associated with one or more promotions after you issue the `DELETE` request, it is not deleted but is no longer associated with the promotions you specified in the request body. If the coupon is no longer associated with any promotions, it is deleted. If you remove all a promotion's coupons and do not add another, shoppers will never be able to claim the promotion; it is automatically disabled until you associate it with at least one coupon or coupon batch and then re-enable it in one of the following ways:

- With the API: Issue a `PUT` request to either `/ccadmin/v1/promotions/{id}` whose body includes `"enabled": true`.
- In the UI: On the Marketing tab, click the name of the promotion and on its details page, select the Enabled checkbox.

The following example removes the coupon `FreeSummer` from the promotion whose ID is `promo10001`:

```
DELETE /ccadmin/v1/claimables/FreeSummer?promotionId=promo10001
```

The following example removes the coupon batch `SAVE4e49eON50` from the promotion whose ID is `promo10007`:

```
DELETE /ccadmin/v1/couponBatches/SAVE4e49eON50?promotionId=promo10007
```

Set up promotion upsell messages

When working with promotions, you can configure promotion upsell messages for your shoppers.

Promotion upsell messages let your shoppers know if they are close to qualifying or have successfully qualified for a promotion. For example, if you have a promotion that says "Buy Two Dog Bowls Get a Free Collar", when a shopper has put one dog bowl in their shopping cart, the upsell message might say "Buy another dog bowl and you'll get a free collar!" Once they have put two dog bowls in their cart, the message changes to "Congratulations! You're getting a free collar!"

Merchandisers can use the administration interface to select upsell messages to associate with promotions. To use these upsell messages, they must be associated with widgets so that they can be displayed in the appropriate locations.

This section describes how to create message tags and promotional upsell widgets using the REST API. For information on working with promotion upsell messages using the administration interface, refer to Manage Promotions.

The following are the steps to create promotion upsell messages. Some of these steps are, or can be, performed using the administration interface. You may not need to perform all of these steps, they are provided for context:

- The merchandiser creates the promotions. Refer to Understand promotions for information on working with and creating promotions.
- Create tags for the promotions, as described in Create message tags. Tags can also be created using the administration interface, as described in Manage upsell messages.
- Create the messages, as described in Manage upsell messages.
- Publish the new tags so that they can be accessed by widgets and promotions. For additional information, refer to Publish Changes.
- Create a promotion upsell widget as described in Create promotion upsell widgets.
- Upload the widget as described in Create and load the extension bundle.

Understand promotion upsell messages

This section describes promotion upsell messages that are created using the REST API. For information on using the administrative interface, refer to Manage upsell messages. Upsell messages are displayed using messaging widgets that read tags indicating which and where messages appear. For example, promotion upsell messages can be displayed on the shopper's checkout page, the site's home page and the product description page.

There are three types of promotion upsell messages:

- `Not Qualified` – This message indicates that the shopper has not yet qualified to receive the promotion.
- `Partially Qualified` – This message indicates that the shopper is close to obtaining a promotion. The message may include specific actions that the shopper can take to obtain the promotion. You can provide placeholder variables that allows you to specify the values needed for a shopper's cart qualification. This type of message also allows you to implement closeness qualifiers that you use to identify criteria that trigger messages.
- `Success` – This message indicates that the shopper has reached the criteria needed to receive the promotion.

Note that you can create only one of each type of upsell message per promotion.

Promotions are associated to an order once a pricing operation is initiated. During the pricing operation, each promotion is evaluated. Once a promotion qualifies, it is removed from further evaluation. Upsell messages are specific to individual promotions and cannot be used across promotions.

Should the shopper no longer qualify for a promotion, the partially qualified message, or the unqualified message if the cart does not meet the qualifying criteria, replaces the success message.

A threshold level can be set for partially qualified messages. This allows you to set a specific threshold which, when the shopper reaches the level, displays the partially qualified message. This threshold is read by the pricing engine that uses the threshold as a boundary condition. For example, if you set a threshold for $30, once a shopper's cart reaches $30, the promotion becomes qualified and a qualified upsell message is presented.

If the promotion has the same tag for all of the upsell messages, the partially qualified message will replace the not qualified message when the closeness qualifier fires. The success message will replace the partially qualified message when the promotion is applied, and the partially qualified message will replace the success message if the shopper no longer qualifies for the promotion. The not qualified message will replace the partially qualified message if the closeness qualifier no longer fires.

For additional information on promotion upsell messages, refer to Manage upsell messages.

Use the promotion upsell API

You can set up promotion upsell messages using the administration interface, as described in Manage upsell messages, or using the REST API with the endpoints listed in this section.

Create message tags

Messages use tags, in conjunction with promotion upsell messaging widgets, to indicate where an upsell message should appear on the storefront. A merchandiser uses the administration interface to select the tags associated with each message. A single upsell message can be associated with multiple tags.

To create or update message tags, use the `messageTags` API by issuing a POST command in `/ccadmin/v1/messageTags`. Use the message tag endpoints to work with tag item types.

Message tag endpoints allow you to manage tags, and include the following:

- `createMessageTag`
- `getMessageTag`
- `updateMessageTag`
- `listMessageTags`
- `deleteMessageTag`

Refer to the API documentation for additional information on using these endpoints.

Use the `createMessageTag` endpoint to create and name a tag. The following example creates a tag named FREE_SHIPPING_HOMEPAGE_BANNER, indicating that this tag displays a message on the banner of the home page. When you create tags, name them with a term or phrase that will help you identify their use. For example, a tag named “FREE_SHIPPING_HOMEPAGE_BANNER” indicates that the message appears in the banner on the home page, or “CART_UPSELL” indicates that the message is displayed on the shopper’s cart:

```
// createMessageTag:
{
  "name": "FREE_SHIPPING_HOMEPAGE_BANNER"
}
// returns:
{
  "repositoryId" : "mt200006",
  "name" : "FREE_SHIPPING_HOMEPAGE_BANNER"
}
```

Note: Message tags are case sensitive. Also, when naming your message tags, do not use the following characters: < >, { }, {{ }} or " ".

You must remember to publish your new tags so that promotions and widgets can reference them. For information on publishing, refer to Understand publishing.

Work with promotion upsell messages

When you create a new message, provide a `message` and a `tag` value. The `tag` value is the name of the tag that you created and the `message` value is the text of the message that is displayed to the shopper. This text can be localized.

For example:

```
{
  "unqualifiedMessages": [
   { "message": "Spend $100 and get free shipping!",
     "tags": [
   { "name": "CART_UPSELL" },
   ] } ]
 }
```

The process of creating or updating a message differs only that you provide an existing message ID if updating.

Note: If you update a promotion without providing an ID, the system will create a new message, overwriting the message you are trying to update.

```
The following example updates an unqualified message with new text and
creates an additional tag:
// Change any of the contents EXCEPT IDs.
// NOTE: The presence of the message ID indicates this is an existing
message.
// This example changes the message, and adds a tag named
"LOYALTY_CART".
{
  "unqualifiedMessages": [
    { "repositoryId": "tm100003"
      "message": " Spend $100 or 10 loyalty points and get free
shipping!",
    "tags": [
      { "repositoryId: "tm100003"
        "name": "FREE_SHIPPING_HOMEPAGE_BANNER" },
      { "name": "LOYALTY_CART" } ] } ]
 }
```

Work with promotion endpoints

Use the following promotion endpoints to set up and configure upsell messages with closeness qualifiers and promotions:

- `getPromotion` – This endpoint displays the upsell messages.
- `updatePromotion` – This endpoint allows you to update both new and existing messages, as well as apply tags. There are three properties available on promotions that are relevant to upsell messages: `unqualifiedMessages`, `qualifiedMessages`, and `closenessQualifier`.

Use the `updatePromotion` endpoint to manage promotion messages by issuing a PUT command to `/ccadmin/v1/promotions/{ID}` to initiate a call for a specific promotion.

Use placeholder variables

Partially qualified messages can use placeholder variables to indicate where a value should be inserted. Placeholder variables are identified with double brackets (`{{ }}`). For example, if the shopper has put two dog bowls in the cart, you could present a message like “Buy **2** more dog bowls and get a free collar!” To do this, you would create the following message: `"Buy <b>{{QuantityStillNeeded}}</b> more dog bowls and get a free collar!"` Note that formatting should be put outside of the place holder brackets. Putting formatting tags within the double brackets will result in an error.

Be careful when crafting your message and avoid plural or singular tenses. You cannot create messages that change based on tense. For example, if you say “Buy 1 more bowl!” when you work with more than one object, the message would be “Buy 2 more bowl!” You messages should be more generic, for example, “You’ve almost got all of your bowls! You need X more to get free shipping!”

These variables, which are computed during the pricing operation, are as follows:

- `{{AmountSpent}}` – The shopper’s current qualifying amount spent.
- `{{AmountStillNeeded}}` – Identifies the amount still needed by the shopper to qualify for the promotion.
- `{{QuantityBought}}` – Identifies the number of qualifying items in the shopper’s cart.
- `{{QuantityStillNeeded}}` – Indicates the minimum number of items needed in the shopper’s cart to qualify for the promotion.

Note that this list contains the accepted place holders. Any other placeholder variables that you create, even if you use the curly braces (`{{ }}`), is treated as literal text and will be seen by your shoppers.

When you create a message, you can indicate where you want a dynamic variable by using the dynamic variable-specific syntax. When a widget queries for messages, these numbers are calculated dynamically and then returned in the message. For example, to create the partially qualified message “You’re almost there! Spend X and you’ll get free shipping!” you would provide the message ID, and the necessary tags.

```
{
  "closenessMessages":
   { "repositoryId": "tm100003"
     "message" : "You’re almost there! Spend {{AmountStillNeeded}}, and
you’ll get free shipping!",
   "tags": [
     { "name": "FREE_SHIPPING_HOMEPAGE_BANNER" }]}
}
```

Understand closeness qualifiers

Closeness qualifiers set the boundaries on how close a shopper is to qualifying for a promotion by holding a condition value and the corresponding message for that condition. Use the `updatePromotion` endpoint to create a closeness qualifier. The following example

creates a closeness qualifier that displays the partially qualified message, also known as a closeness message, when the shopper's cart hits $35:

```
{
  "closenessQualifiers" : [
    {
      "closenessMessages" : [
        {
          "message" : "<b> You're almost there! Spend
{{AmountStillNeeded}}, and you'll
               get free shipping!"</b>",
          "tags" : [{"name" : "FREE_SHIPPING_HOMEPAGE_BANNER"}]
        } ] } ],
}
```

The process of creating or updating a closeness qualifier differs only that you provide an existing ID if updating.

To delete a qualifying condition, pass an empty array for the `closenessQualifiers` property value.

The following is an example `getPromotion` call:

```
...
{
  "unqualifiedMessages": [
    {
      "repositoryId": "tm100001",
      "text": "Spend $100 and get free shipping!",
      "tags": []
    }
  ],
  "qualifiedMessages": [
    {
      "repositoryId": "tm100003",
      "text": "Congratulations! This order is shipping for free!",
      "tags": [
        { "name": "CART_UPSELL" },
        { "name": "FREE_SHIPPING_HOMEPAGE_BANNER" }
      ]
    }
  ],
 "closenessQualifiers": [
   {
     "repositoryId": "2015",
     "closenessMessages": [
       {
         "text": "You're almost there! Spend {{AmountStillNeeded}} and
you'll
            get free shipping!",
         "tags": [
           { "name": "FREE_SHIPPING_HOMEPAGE_BANNER" }
         ]
        }
       ]
```

```
      }
     ],
      "templateValues" : [
     {"customer_alert_spend_value" : 35 } ]
   }
 ...
```

The following is a list of the promotion templates that support closeness qualifiers and their threshold value names in the `templateValues` property. For information on working with promotion templates, refer to Create a promotion. Threshold value names are required when providing input for closeness qualifiers. The following table describes the threshold value names used by each promotion template:

Promotion Template	Threshold Value Name
item/bogo	closeness_value
item/bogoSortBy	closeness_value
item/buyItemXGetGWP	closeness_value
item/spendYGetGWP	customer_alert_spend_value
item/spendYInXGetItemDiscount	customer_alert_spend_value
item/spendYInXGetItemDiscountSortBy	customer_alert_spend_value
order/buyXGetOrderDiscount	closeness_value
order/spendYGetOrderDiscount	spend_close_value
order/spendYInXGetOrderDiscount	customer_alert_spend_value
order/tieredOrderDiscount	spend_close_value
shipping/buyXGetShippingDiscount	closeness_value
shipping/spendYGetShippingDiscount	spend_close_value
shipping/spendYInXGetShippingDiscount	customer_alert_spend_value

Work with order store endpoints

The order store endpoint payloads contain the upsell messages. This includes `priceOrder`, `updateCurrentProfileOrder`, `updateOrder` and `createOrder` endpoints.

Each pricing endpoint response contains a list of promotion upsell messages that contain a text and a tags property. The text property sets the message value, while the tags property indicates the name or names of the associated tags. For example:

```
},
  "pricingMessages": {
    "promotionUpsellMessages":[
    {
      "text":"Spend $45 get free shipping!",
      "tags":["BANNER_UPSELL", "FOOTER_UPSELL"]
    },
    {
      "text":"You're almost there! Spend $100 and you'll get free shipping!",
      "tags":["CART_UPSELL"]
    },
    {
      "text":"Add one more dog bowl and get a free collar!",
      "tags":["CART_UPSELL"]
    },
```

```
    {
      "text":"Buy any 2 All Natural Chew Toys and get 50%!",
      "tags":["CART_UPSELL"]
    }
  ]},
```

You can use the `/ccstore/v1/orders/getUpsellMessages` call to return unqualified or success messages. For example:

```
{
  "promotionUpsellMessages":[
    {
      "text":"Spend $45 and you'll get free shipping!",
      "tags":["CART_UPSELL"]
    },
    {
      "text":"Congratulations! You have qualified for a free collar!",
      "tags":["FREE_SHIPPING_HOMEPAGE_BANNER","CART_UPSELL"]
     }]
 }
```

Create promotion upsell widgets

To display your upsell messages, you create promotion upsell widgets that are used on page layouts. The promotion upsell widget can be added to any layout or page where messages are needed. You can also use multiple instances of the widget on different or the same pages. Messages are available to promotion messaging widgets using tags that are associated with the messages. This allows the relevant promotion message to be displayed in the correct location.

A messaging widget allows you to display particular upsell messages. Messages use the tags that you created earlier in Create message tags to communicate with the widget. Using widgets allows you to select the available tags, as well as specify a threshold for how many messages to display.

The widget uses the `tags` field to identify the message that will be displayed by the widget. The field can contain a single tag name, or a comma-separated list of tag names. The maximum number of messages that can be displayed is configured using the `messageLimit` field.

If there are widgets configured to display promotion messages with the same tag on more than one location on storefront layout, the system cannot display a different message from the sequenced list in each location. You must ensure that your messages use different tags.

The promotion upsell widget uses the promotion upsell container view model, which holds all of the upsell messages obtained from processing and `getUpsellMessages` endpoint calls. The widget instances interact with the view model to get messages with matching tags. This view model also makes endpoint calls to get non-qualified messages from the `getUpsellMessage` endpoint. The cart view model also populates the promotion upsell container view model with promotion messages that come from pricing calls.

A `promotionUpsell` widget could be created in a way similar to the following example. The `widget.json` file might contain the following:

```
{
  "config": {
    "tags":"CART_UPSELL",
    "messageLimit":"1"
  },
  "availableToAllPages": true,
   "global": false,
   "globalEnabled": false,
   "i18nresources": "promotionUpsell",
   "imports": [],
   "javascript": "promotion-upsell",
   "jsEditable": false,
   "name": "Promotion Upsell Widget",
   "version": 1
}
```

The `config.json` file might contain the following:

```
{
  "widgetDescriptorName": "promotionUpsell",
  "properties": [
    {
      "id": "tags",
      "type": "stringType",
      "name": "tags",
      "helpTextResourceId": "tagsHelpText",
      "labelResourceId": "tagsLabel",
      "defaultValue": ""
    },
    {
      "id": "messageLimit",
      "type": "stringType",
      "name": "messageLimit",
      "helpTextResourceId": "messageLimitHelpText",
      "labelResourceId": "messageLimitLabel",
      "defaultValue": "1",
      "required": true
    }
  ]
}
```

The widget's JavaScript might be similar to the following, which you could store in a file named `promotion-upsell.js`:

```
define(
//-------------------------------------------------------------------
// DEPENDENCIES
//-------------------------------------------------------------------
  ['knockout', 'pubsub', 'viewModels/promotionUpsellContainer'],
//-------------------------------------------------------------------
```

```
// MODULE DEFINITION
//--------------------------------------------------------------------
  function(ko, pubSub, promotionUpsellContainer) {
  "use strict";
  return {
   /** Widget root element ID */
   WIDGET_ID: 'promotionUpsell',
   onLoad : function(widget) {
     widget.messageLimit = widget.messageLimit && null !=
widget.messageLimit()
       && widget.messageLimit != "" ? parseInt(widget.messageLimit()) :
       parseInt("1");
     widget.promotionUpsellContainer =
promotionUpsellContainer.getInstance();
     widget.widgetTags = widget.tags && null != widget.tags() &&
widget.tags()
       != "" ? widget.tags().split(","):[];
     widget.promotionUpsellMessages = ko.pureComputed(function() {
       var promoMessages = widget.promotionUpsellContainer.
           promotionUpsellMessages().filter(function(message, index) {
         var widget = this;
         var displayMessage = false;
         widget.widgetTags.forEach(function(widgetTags) {
           for (var i=0; i<message.tags.length; i++) {
             if(message.tags[i] == widgetTags) {
               displayMessage = true;
               break;
             }
           }
         });
       return displayMessage;
       }, widget);
      return promoMessages.slice(0,widget.messageLimit);
     }).extend({ rateLimit: 500 });
     widget.items = ko.computed(function(){
       return widget.cart().allItems();
     }).extend({ rateLimit: 1000 });
     widget.getNonQualifiedMessages = function () {
       if (widget.cart().items().length == 0) {
         widget.promotionUpsellContainer.getNonQualifiedMessages();
       }
     };
     widget.getNonQualifiedMessagesSubsciption = function () {
     widget.items.subscribe(function(newVal) {
       if (newVal.length == 0) {
         widget.promotionUpsellContainer.getNonQualifiedMessages();
       }
      });
     };
     if(!widget.promotionUpsellContainer.
         isNonQualifiedMessagesSubscribedToQuantity) {

widget.promotionUpsellContainer.isNonQualifiedMessagesSubscribedToQuant
ity
         = true;
```

```
        widget.getNonQualifiedMessages();
        $.Topic(pubSub.topicNames.PAGE_LAYOUT_UPDATED).subscribe(widget.
          getNonQualifiedMessagesSubsciption);
        $.Topic(pubSub.topicNames.USER_LOGIN_SUCCESSFUL).subscribe(widget.
          getNonQualifiedMessages);
        $.Topic(pubSub.topicNames.USER_LOGOUT_SUCCESSFUL).subscribe(widget.
          getNonQualifiedMessages);
        $.Topic(pubSub.topicNames.USER_AUTO_LOGIN_SUCCESSFUL).subscribe(widget
.
          getNonQualifiedMessages);
        }
      }
    }
  }
 );
```

The locale resources, which are part of the configuration and provide text, might be similar to this:

```
{
  "resources": {
    "tagsHelpText":"Add the list of tags that this widget uses to display
promotions.",
    "tagsLabel":"List of tags this this widget uses to display promotions.",
    "messageLimitLabel":"Message Limit Label",
    "messageLimitHelpText":"The HelpText message limit."
  }
}
```

The `display.template` file might contain the following:

```
<div id="CC-promotionUpsell">
  <!-- ko foreach : {data: $data.promotionUpsellMessages(), as: 'message'}-->
    <div class="cc-rich-text" data-bind="html: message.text"/>
  <!-- /ko -->
</div>
```

If you have multiple widgets you must query whatever mechanism you use for storing the results of the API call for that specific widget.

8

Manage Multiple Inventory Locations

By default, Oracle Commerce maintains one set of inventory values for each product or SKU. You may want to maintain multiple inventory locations so that you can provide inventory information to shoppers.

This section describes how to use the Admin API to create locations and manage location-specific inventory data. For example, you may currently have multiple international sites that are supplied from a centralized inventory warehouse, but now you would like to have multiple inventory locations that service specific sites so that your Scottish site accesses a Scottish distribution warehouse and your German-based site accesses a German-based distribution warehouse.

An inventory record uses a location ID (`locationId`) that associates it with a location with the same location ID (`locationId`) value. A site is associated with the location when its location inventory ID (`inventoryLocationId`) matches the location's location ID (`locationId`).

Note that the inventory's location ID does not need to match a location's location ID. You can create an ID that allows you to manage inventory data. Multiple inventories allow you to provide different inventory values per site, rather than a single inventory, and help you to split inventory data.

Note that any changes you make to inventory are reflected on the storefront immediately. However, when you create or modify a location, you must publish these changes before they appear on the storefront. Also, note that the Store API has endpoints for retrieving locations and inventory data, but not for creating or modifying these resources.

Access inventory data

You can use the `getInventory` endpoint to retrieve inventory information for a specific product or SKU.

The `getInventory` endpoint takes a `type` query parameter to specify the item type. The value of this parameter must be `product` (for a product) or `variant` (for a SKU). The default is `variant`, so if you omit the parameter, Oracle Commerce assumes that the item is a SKU.

For example:

```
GET /ccadmin/v1/inventories/xprod1004?type=product HTTP 1.1
Authorization: Bearer <access_token>
```

The response body includes inventory information for the product as a whole, plus information about each individual SKU:

```
{
    "id": "xprod1004",
    "stockStatus": "partialAvailability",
    "totalStockLevel": 210,
    "links": [
```

```
        {
            "rel": "self",
            "href":"https://myserver.example.com:7002/ccadmin/v1/
               inventories/xprod1004?type=product"
        }
    ],
    "childSKUs": [
        {
            "preorderThreshold": 0,
            "stockThreshold": 0,
            "availabilityStatus": 1000,
            "backorderThreshold": 0,
            "availabilityStatusMsg": "inStock",
            "backorderLevel": 0,
            "locationId": null,
            "preorderLevel": 0,
            "skuNumber": "xsku5014",
            "availableToPromise": null,
            "translations": null,
            "skuId": "xsku5014",
            "availabilityDate": null,
            "inventoryId": null,
            "displayName": "Titanium Analog Watch",
            "stockLevel": 100
        },
        {
            "preorderThreshold": 0,
            "stockThreshold": 0,
            "availabilityStatus": 1000,
            "backorderThreshold": 0,
            "availabilityStatusMsg": "inStock",
            "backorderLevel": 0,
            "locationId": null,
            "preorderLevel": 0,
            "skuNumber": "xsku5015",
            "availableToPromise": null,
            "translations": null,
            "skuId": "xsku5015",
            "availabilityDate": null,
            "inventoryId": null,
            "displayName": "Silver Plated Analog Watch",
            "stockLevel": 100
        },
        {
            "preorderThreshold": 0,
            "stockThreshold": 20,
            "availabilityStatus": 1001,
            "backorderThreshold": 0,
            "availabilityStatusMsg": "outOfStock",
            "backorderLevel": 0,
            "locationId": null,
            "preorderLevel": 0,
            "skuNumber": "xsku5016",
            "availableToPromise": null,
            "translations": null,
```

```
            "skuId": "xsku5016",
            "availabilityDate": null,
            "inventoryId": null,
            "displayName": "Brushed Steel Analog Watch",
            "stockLevel": 10
        }
    ],
    "displayName": "Analog Watch"
}
```

If you specify a SKU, you can omit the `type` parameter. For example:

```
GET /ccadmin/v1/inventories/xsku5014 HTTP 1.1
Authorization: Bearer <access_token>
```

The response contains inventory information about the SKU only:

```
{
    "preorderThreshold": 0,
    "stockThreshold": 0,
    "availabilityStatus": 1000,
    "backorderThreshold": 0,
    "availabilityStatusMsg": "inStock",
    "backorderLevel": 0,
    "locationId": null,
    "links": [
        {
       "rel": "self",
       "href": "https://myserver.example.com:7002/ccadmin/v1/inventories/
xsku5014"
        }
    ],
    "preorderLevel": 0,
    "skuNumber": "xsku5014",
    "availableToPromise": null,
    "translations": null,
    "skuId": "xsku5014",
    "availabilityDate": null,
    "inventoryId": null,
    "displayName": "Titanium Analog Watch",
    "stockLevel": 100
}
```

Update inventory

You can use the `updateInventory` endpoint to modify the inventory of a specific SKU. In the request body, specify new values for the properties you want to update. For example:

```
PUT /ccadmin/v1/inventories/xsku5014 HTTP 1.1
Authorization: Bearer <access_token>

{
  "stockThreshold": 15,
```

```
  "stockLevel": 200
}
```

Create locations

To maintain inventory for individual physical stores or web sites, you must represent them in Oracle Commerce by creating new locations.

Create physical locations

Use the `createLocation` endpoint to create a new physical location. You specify information about the location in the body of the request. The following example creates a physical location for a warehouse:

```
POST /ccadmin/v1/locations HTTP/1.1
Authorization: Bearer <access_token>

{
  "externalLocationId": "107",
  "locationId": "Warehouse13",
  "address1": "221 Third Street",
  "country": "USA",
  "city": "Cambridge",
  "faxNumber": "(617) 386-1200",
  "postalCode": "02141",
  "phoneNumber": "(617) 386-1200",
  "email": "wh13@example.com",
  "stateAddress": "MA",
  "county": "Middlesex",
  "name": "Warehouse 13 -- 02141",
  "longitude": -71.0901,
  "latitude": 42.3629
}
```

You must supply a value for the name property. If you omit `locationId` (the property used to identify the location in REST API calls), a value is automatically supplied. If you omit other properties, their values will be null.

The endpoint returns the location information in the response body:

```
{
  "country": "USA",
  "distance": null,
  "city": "Cambridge",
  "endDate": null,
  "postalCode": "02141",
  "latitude": 42.3629,
  "county": "Middlesex",
  "stateAddress": "MA",
  "pickUp": false,
  "sites": [],
  "type": "location",
  "inventory": false,
```

```
  "locationId": "Warehouse13",
  "email": "wh13@example.com",
  "longitude": -71.0901,
  "address3": null,
  "address2": null,
  "address1": "221 Third Street",
  "externalLocationId": "107",
  "phoneNumber": "(617) 386-1200",
  "siteGroups": [],
  "repositoryId": "Warehouse13",
  "name": "Warehouse 13 -- 02141",
  "faxNumber": "(617) 386-1200",
  "startDate": null
}
```

Understand site inventory locations

When a shopper accesses a Product Display page, the page displays the item's stock level at the site's specified inventory location. An inventory check is also made when a shopper adds to or decreases the amount of an item in a cart.

Location-based inventory allows each of your web sites to be served by its own inventory. Once you have mapped a specific web site to an inventory, transactions will display the relevant inventory levels. Oracle Commerce identifies a site's inventory location and uses that location throughout the shopper's session. This means that any inventory checks made during the session are location-aware. For example, when an order is submitted, the inventory decrement occurs on the inventory linked to the site.

A site can be mapped to a default inventory location. If no inventory mapping for a site is detected, the system uses the default inventory location.

The `shipFrom` location based on the storefront's inventory location is used to calculate shipping costs. A hard goods shipping group is created when a cart item is identified as being shipped to a physical address.

Create a web site inventory location

Before you can create an inventory location for your site, you must create or update a site to use the `inventoryLocationId` property. Use the site endpoint to set a default inventory location ID for a custom site that contains the `inventoryLocationId`. For example:

```
POST /ccadmin/v1/sites HTTP/1.1
Authorization: Bearer <access_token>

{
  "properties": {
    "repositoryId": "CustomSite2",
    "name": "Custom Site 2",
    "defaultCatalog": {
      "repositoryId": "ClassicalMoviesCatalog"
    },
    "inventoryLocationId": "WH12",
    "enabled": true
  }
}
```

Once you have the site configured with the inventory location ID, you can create a location. Set the site's location ID and include the custom site in the sites array:

```
POST /ccadmin/v1/locations HTTP/1.1
Authorization: Bearer <access_token>

{
  "country": "USA",
  "address3": null,
  "endDate": "2017-04-25",
  "address2": "Building 4",
  "city": "Glen Allen",
  "address1": "4870 Sadler Rd.",
  "latitude": 37.6659833,
  "postalCode": "23060",
  "county": "Henrico",
  "stateAddress": "VA",
  "externalLocationId": "187",
  "phoneNumber": "(617) 637-8687",
  "locationId": "Warehouse12",
  "name": "Warehouse 12 --23060",
  "faxNumber": "(617) 386-1200",
  "startDate": "2016-04-25",
  "email": "wh12@example.com",
  "longitude": -77.5063697
}
```

You can determine when and if inventory levels are decremented by identifying the specific inventory location for each SKU or product. The default inventory is decremented if you do not configure an inventory location.

Note the following:

- A site that contains a null `inventoryLocationId` returns records from a null inventory
- A site that contains an `inventoryLocationId` that is mapped to a location but not mapped to an inventory record empties inventory records with a status of `IN_STOCK`.

Once the locations are defined, you must associate an inventory record with a location as described in the Create inventory data for locations section.

List locations

Use the `listLocations` endpoint to retrieve a listing of all locations:

```
GET /ccadmin/v1/locations HTTP/1.1
Authorization: Bearer <access_token>
```

You can use query parameters to restrict the set of locations returned.

Modify a location

Use the `updateLocation` endpoint to modify a location:

```
PUT /ccadmin/v1/locations/Warehouse13 HTTP/1.1
Authorization: Bearer <access_token>

{"postalCode": "02141"}
```

Delete a location

Use the `deleteLocation` endpoint to delete a location:

```
DELETE /ccadmin/v1/locations/Warehouse13 HTTP/1.1
Authorization: Bearer <access_token>
```

Delete a location currently in use

If you remove active locations from a custom site, the following may occur:

- Returns will no longer be possible for removed locations
- The location ID property is not removed; references to it will remain on your site / inventory
- The Ship From address defaults to the site's default
- Historic orders will reference the location used when the order was placed

Create inventory data for locations

You can use the `createInventory` endpoint to set inventory values for a specific SKU at a location.

When you create a new inventory location, it initially has no inventory data associated with it. You can set inventory values for a SKU at the location. For example, the following request specifies inventory data for the location created in the previous section:

```
POST /ccadmin/v1/inventories HTTP/1.1
Authorization: Bearer <access_token>

{
  "locationId": "Warehouse13",
  "id": "xsku5014",
  "stockThreshold": 10,
  "stockLevel": 75
}
```

The endpoint includes the inventory data in the response body:

```
{
    "locationInventoryInfo": [
        {
            "preorderThreshold": 0,
```

```
            "stockThreshold": 10,
            "availabilityStatus": 1000,
            "backorderThreshold": 0,
            "availabilityStatusMsg": "inStock",
            "backorderLevel": 0,
            "locationId": "Warehouse13",
            "preorderLevel": 0,
            "skuNumber": null,
            "availableToPromise": null,
            "translations": null,
            "availabilityDate": null,
            "inventoryId": null,
            "displayName": null,
            "stockLevel": 75
        }
    ],
    "links": [
        {
            "rel": "self",
            "href": "https://myserver.example.com:7002/ccadmin/v1/
inventories"
        }
    ],
    "skuNumber": "xsku5014",
    "skuId": "xsku5014",
    "displayName": "Titanium Analog Watch"
}
```


Notice that the response encapsulates the returned inventory data in a `locationInventoryInfo` array object. This object type is used whenever inventory is returned for a non-default location, and enables returning inventory for multiple locations. (See Retrieve inventory data for locations for an example of returning inventory for multiple locations.)

Update inventory for a location

You can use the `updateInventory` endpoint to modify the inventory of a specific SKU at a specific location. You specify the location and the updated inventory data in the request body. For example:

```
PUT /ccadmin/v1/inventories/xsku5014 HTTP/1.1
Authorization: Bearer <access_token>

{
  "locationId": "Warehouse13",
  "stockThreshold": 20,
  "stockLevel": 200
}
```

Delete inventory for a location

You can use the `deleteInventory` endpoint to delete the inventory of a specific SKU at a specific location. This endpoint takes two query parameters:

- A `type` query parameter to specify the item type. The value of this parameter must be `product` (for a product) or `variant` (for a SKU). The `default` is variant, so if you omit the parameter, Oracle Commerce assumes that the item is a SKU.
- An optional `locationId` query parameter to specify the location by its ID.

For example:

```
DELETE /ccadmin/v1/inventories/xsku5014?locationId=Warehouse13 HTTP/1.1
Authorization: Bearer <access_token>
```

Retrieve inventory data for locations

To retrieve inventory at specific locations, you use the `getInventory` endpoint with the `locationIds` query parameter.

The `locationIds` query parameter allows you to specify one or more location IDs as a comma-separated list. For example:

```
GET /ccadmin/v1/inventories/xsku5014?locationIds=Warehouse13,Warehouse11
HTTP/1.1
Authorization: Bearer <access_token>
```

The response includes a `locationInventoryInfo` array in which each entry is the inventory for one of the locations specified in the URL. For example:


```
{
    "locationInventoryInfo": [
        {
            "preorderThreshold": 0,
            "stockThreshold": 20,
            "availabilityStatus": 1000,
            "backorderThreshold": 0,
            "availabilityStatusMsg": "inStock",
            "backorderLevel": 0,
            "locationId": "Warehouse13",
            "preorderLevel": 0,
            "skuNumber": null,
            "availableToPromise": null,
            "translations": null,
            "availabilityDate": null,
            "inventoryId": null,
            "displayName": null,
            "stockLevel": 200
        },
        {
            "preorderThreshold": 0,
            "stockThreshold": 5,
            "availabilityStatus": 1000,
            "backorderThreshold": 0,
            "availabilityStatusMsg": "inStock",
            "backorderLevel": 0,
            "locationId": "Warehouse11",
            "preorderLevel": 0,
```

```
            "skuNumber": null,
            "availableToPromise": null,
            "translations": null,
            "availabilityDate": null,
            "inventoryId": null,
            "displayName": null,
            "stockLevel": 22
        }
    ],
    "links": [
        {
       "rel": "self",
       "href": "https://myserver.example.com:7002/ccadmin/v1/
inventories/xsku5014?
         locationIds=Warehouse13,Warehouse11"
        }
    ],
    "skuNumber": "xsku5014",
    "skuId": "xsku5014",
    "displayName": "Titanium Analog Watch"
}
```

You can include inventory for the default inventory location as well by setting the `includeDefaultLocationInventory` query parameter to `true`. For example, the following call returns inventory for the default location as well as the Warehouse13 location:

```
GET /ccadmin/v1/inventories/xsku5014?
  locationIds=Warehouse13&includeDefaultLocationInventory=true HTTP/1.1
Authorization: Bearer <access_token>
```

Note that if you do not use the `locationIds` parameter to specify a non-default location, inventory for the default location is returned. This is the case even if you do not set `includeDefaultLocationInventory` to `true`. If you do use `locationIds`, however, inventory for the default location is omitted unless you explicitly set `includeDefaultLocationInventory` to `true`.

Set an inventory record for a SKU using the location's `locationId`. For example:

```
POST /ccadmin/v1/inventories/ HTTP/1.1
Authorization: Bearer <access_token>

// IN STOCK

{
  "id": "Sku_13D",
  "locationId": "Warehouse13",
  "type": "variant",
  "stockThreshold": 5,
  "stockLevel": 3000,
  "availabilityDate": null,
  "preorderLevel": 0,
  "preorderThreshold": 0,
  "backorderLevel": 0,
```

```
  "backorderThreshold": 0
}

// OUT OF STOCK

{
  "id": "Sku_15DE",
  "locationId": "Warehouse13",
  "type": "variant",
  "stockThreshold": 5,
  "stockLevel": 1,
  "availabilityDate": null,
  "preorderLevel": 0,
  "preorderThreshold": 0,
  "backorderLevel": 0,
  "backorderThreshold": 0
}
```

9

Manage Inventory for Preorders and Backorders

To market and accept orders for items that are not yet available to ship, you must use the Admin API to enable backorders and preorders.

This section describes how to manage backorder and preorder inventory data.

Understand inventory

This section describes the data that Commerce uses to track inventory.

Commerce maintains the following inventory data that determines the availability of each SKU:

- `stockLevel` is the number that can be purchased.
- `preorderLevel` is the number that can be preordered.
- `backorderLevel` is the number that can be backordered.

If `stockLevel` is not 0, then the SKU is in stock.

If `stockLevel` is 0 but `backorderLevel` is not 0, then the SKU is backorderable.

If `stockLevel` and `backorderLevel` are both 0, but `preorderLevel` is not 0, then the SKU is preorderable.

If all three levels are 0, then the SKU is out of stock.

In addition to inventory levels, Commerce maintains the following inventory data for each SKU:

- `stockThreshold` is the threshold at which the status of the SKU changes to out of stock.
- `preorderThreshold` is the threshold at which the status of the SKU changes from preordered to out of stock.
- `backorderThreshold` is the threshold at which the status of the SKU changes from backordered to out of stock.
- `availabilityDate` is the date on which the SKU becomes available.
- `availabilityStatus` is the status of the SKU, for example, preordered, backordered, or out of stock.

Enable preorder and backorder functionality

You can use the `updateInventoryConfiguration` endpoint to enable preorder and backorder functionality.

By default, preorder and backorder functionality are disabled in Commerce. This section describes how to use the `updateInventoryConfiguration` endpoint to enable preorder and

backorder functionality. Preorders and backorders are enabled and disabled at the same time with the `preorderBackorderEnabled` property. You cannot enable or disable them separately.

The following sample request enables backorder and preorder functionality:

```
PUT /ccadmin/v1/merchant/inventoryConfiguration HTTP/1.1
Authorization: Bearer <access_token>

{
  "preorderBackorderEnabled": true
}
```

Access and update inventory data

This section describes how to set backorder and preorder levels and thresholds.

You can use the administration interface to set stock levels and stock thresholds, but not to set backorder and preorder levels and thresholds. Instead, you must use one of the following methods:

- Import inventory data for your catalog with the Commerce import feature. To learn how to import preorder and backorder inventory data, see Import and Export Catalog Items and Inventory.
- Set SKU inventory data with the Admin API `updateInventory` endpoint. This section describes how to use the `updateInventory` endpoint.
- In a system where Commerce interacts with an external system, import and export large amounts of inventory data with the Admin API. See Perform Bulk Export and Import for more information.

Before you set backorder and preorder inventory data, you must enable backorders and preorders with the `updateInventoryConfiguration` endpoint. See Enable preorder and backorder functionality for more information.

Use the `getInventory` endpoint to retrieve inventory information for a specific product or SKU. For details about using this endpoint, see Access inventory data.

Use the `updateInventory` endpoint to modify the inventory of a specific SKU. In the request body, specify new values for the properties you want to update.

The following sample request sets the backorder level for SKU to 200. The backorder threshold is 15, at which point the item will show as out of stock on the store.

```
PUT /ccadmin/v1/inventories/xsku5014 HTTP 1.1
Authorization: Bearer <access_token>

{
  "backorderThreshold": 15,
  "backorderLevel": 200
}
```

The following example shows a sample response:


```
{
  "preorderThreshold": 0,
  "backorderLevel": 200,
  "displayName": null,
  "availabilityDate": null,
  "availabilityStatusMsg": "backorderable",
  "stockThreshold": 5,
  "stockLevel": 0,
  "availableToPromise": null,
  "skuNumber": "xsku5014",
  "preorderLevel": 0,
  "locationId": null,
  "translations": null,
  "inventoryId": null,
  "backorderThreshold": 15,
  "links": [
    {
      "rel": "self",
      "href": "https://myserver.example.com:7002/ccadmin/v1/inventories/
xsku5014
  ],
  "availabilityStatus": 1003,
  "skuId": "xsku5014",
}
```


Update widgets for preorders and backorders

To implement support for preorders and backorders, make sure your layouts include the latest versions of the Shopping Cart and Product Details widgets.

If your store includes the scheduled orders feature, make sure the Scheduled Order layout includes the latest version of the Scheduled Order widget. To determine if you are using the latest version, and for information about replacing a widget with the latest version, see Customize your store layouts.

Customize email templates for preorders and backorders

If you configure your store to support preorders and backorders, you should also customize the templates for emails that contain order summaries so those emails can display the appropriate availability status for preordered and backordered items.

The following email templates include order summaries:

- Order Placed
- Items Shipped
- Order Pending For Approval
- Order Rejected
- Order Approved
- Order Quoted

- Quote Requested
- Quote Failed
- Payment Failure
- Abandoned Order
- Scheduled Order Placed Failed
- Agent Cancel Order
- Agent Edit Order
- Agent Return Order Refund
- Agent Return Order

The data available to the email templates to support preorders and backorders comes from the Orders resource in the Store REST API. Support has been added for preorders and backorders, in the form of the following properties:

- `backOrderedQuantity` is an integer that specifies the quantity is reserved from the `backorderLevel` for an item.
- `preOrderedQuantity` is an integer that specifies the quantity is reserved from the `preorderLevel` for an item.
- `availabilityDate` is a string that specifies the date that the preordered or backordered item will be available.

These properties are typically added to the template in the order items list.

Note: Before you customize the email templates, read the Configure Email Settings. For details about working with FreeMarker templates, see the Apache FreeMarker documentation at `freemarker.org`.

To display preorder and backorder information in an email template:

1. Download the email template as described in Customize email templates.
2. Update the `html_body.ftl` file.
3. Upload the updated template as described in Customize email templates.

10
Manage Orders

An order is created when a shopper successfully completes the checkout process on your store.

An Oracle Commerce order object stores a great deal of data about the transaction: the items purchased, the shopper's shipping address, the payment method, and so on. Once an order is created in Oracle Commerce, the data is sent off to an external order management system (OMS) for processing and fulfillment. This section describes how to manage orders in Oracle Commerce.

Integrate with an order management system

Once an order is created, your external order management system (OMS) is the system of record and is responsible for fulfilling the order.

Communication between Oracle Commerce and the OMS is handled as follows:

- Oracle Commerce uses webhooks to send order data to your OMS or to a gateway that transmits the data to the OMS.
- The order management system implements the Oracle Commerce Admin REST API to update the order's status information as the order is processed.

You need to configure your OMS or gateway to convert the data received from Oracle Commerce into the format used by the OMS, and to set up the OMS to make calls to the Oracle Commerce REST APIs to update the status of the order. Note that for certain order management systems, optional integration software is available to simplify this process.

Configure the integration points

Follow these steps to enable Oracle Commerce to communicate with your order management system:

1. Configure the webhooks you intend to use. At a minimum, you will need to configure the Order Submit webhook. If your store includes the Agent Console, you should also configure the Return Request Update webhook.
   To configure a webhook, you supply the URL to direct POST requests to, the headers to include, and authentication information. The values to supply are determined by your OMS. See Configure Webhooks for more information.
2. Register your OMS application with Oracle Commerce. Doing this generates an application ID and an application key that the OMS can use for authentication when it makes REST calls to Oracle Commerce. See Register applications for more information.

Order Submit webhook

When an order is submitted, the Order Submit webhook sends a `POST` request to the URL you have configured. (Typically this is the URL where your OMS or gateway listens for requests.) The body of the request contains the complete order data in JSON format. The order management system converts the JSON data into the system's native format, and returns an HTTP status code indicating whether the data was received successfully. A 200-level status

code indicates the `POST` was successful. Any other code indicates failure; if this occurs, Order Submit sends the `POST` request again. The webhook is executed up to five times until it succeeds or gives up.

Not all external systems you integrate with Oracle Commerce will comply with the Payment Card Industry Data Security Standard (PCI DSS). For example, while your order management system will likely comply with PCI DSS, systems that manage services like email marketing or customer loyalty programs might not be compliant. Oracle Commerce provides versions of the Order Submit webhook that exclude payment details from the order data you send to systems that do not comply with PCI DSS. See Understand webhooks and PCI DSS compliance for more information.

Order Submit request example

The following example shows the body of an Order Submit webhook POST request from Oracle Commerce. The request body is a JSON representation of the order.


```
{
    "site": {
        "siteURL": "http://www.example.com",
        "siteName": "Commerce Site"
    },
    "order": {
        "lastModifiedTime": 1403734373592,
        "shippingGroupCount": 1,
        "paymentGroupCount": 1,
        "shippingGroups": [
            {
                "specialInstructions": {},
                "id": "sg20005",
                "handlingInstructions": [],
                "trackingNumber": null,
                "priceInfo": {
                    "amount": 6.5,
                    "currencyCode": "USD",
                    "amountIsFinal": false,
                    "discounted": false,
                    "rawShipping": 6.5
                },
                "description": "sg20005",
                "state": 0,
                "locationId": null,
                "actualShipDate": null,
                "submittedDate": null,
                "shipOnDate": null,
                "shippingMethod": "ground",
                "shippingAddress": {
                    "middleName": "",
                    "lastName": "Smith",
                    "ownerId": null,
                    "state": "Alaska",
                    "address1": "101 TNT Dr",
                    "address2": "",
                    "address3": "",
                    "companyName": "",
                    "suffix": "",
```

```
                "country": "United States",
                "city": "Birmingham",
                "id": null,
                "postalCode": "99672",
                "faxNumber": "",
                "phoneNumber": "555-555-1212",
                "county": "",
                "email": "home@example.com",
                "prefix": "",
                "firstName": "Jean",
                "jobTitle": ""
            },
            "stateDetail": null
        }
    ],
    "commerceItems": [
        {
            "id": "ci2000007",
            "productDisplayName": "Military Jacket",
            "returnedQuantity": 0,
            "priceInfo": {
                "quantityDiscounted": 0,
                "amount": 291,
                "discountable": true,
                "onSale": false,
                "priceListId": "listPrices",
                "currencyCode": "USD",
                "rawTotalPrice": 291,
                "listPrice": 145.5,
                "amountIsFinal": false,
                "discounted": false,
                "currentPriceDetailsSorted": [
                    {
                        "amount": 291,
                        "itemPriceInfo": null,
                        "currencyCode": "USD",
                        "range": {
                            "lowBound": 0,
                            "class": "atg.core.util.Range",
                            "highBound": 1,
                            "size": 2
                        },
                        "tax": 0,
                        "amountIsFinal": false,
                        "discounted": false,
                        "quantity": 2,
                        "detailedUnitPrice": 145.5
                    }
                ],
                "salePrice": 0
            },
            "catalogId": null,
            "quantity": 2,
            "catalogKey": null,
            "catalogRefId": "sku40139",
```

```
                "productId": "prod20012"
            },
        ],
        "id": "o20005",
        "siteId": "siteUS",
        "priceInfo": {
            "total": 268.4,
            "amount": 261.9,
            "shipping": 6.5,
            "currencyCode": "USD",
            "tax": 0,
            "amountIsFinal": false,
            "discounted": true,
            "manualAdjustmentTotal": 0,
            "rawSubtotal": 291,
            "discountAmount": 29.1
        },
        "paymentGroups": [
            {
                "authorizationStatus": [
                    {
                        "errorMessage": "Request was processed
successfully.",
                        "amount": 261.9,
                        "authorizationDecision": "ACCEPT",
                        "transactionId": "403734370870017814762 6",
                        "reasonCode": "100",
                        "currency": "USD",
                        "transactionSuccess": true
                    }
                ],
                "currencyCode": "USD",
                "paymentId": "pg20005",
                "state": 1,
                "amountAuthorized": 261.9,
                "amount": 297.5,
                "id": "pg20005",
                "phoneNumber": "555-555-1212",
                "token": "9997000107329795",
                "expirationYear": "2021",
                "expirationMonth": "08",
                "submittedDate": {
                    "time": 1403734373000
                },
                "creditCardNumber": "1111",
                "paymentMethod": "tokenizedCreditCard"
            }
        ],
        "taxPriceInfo": {
            "amount": 0,
            "currencyCode": "USD",
            "countyTax": 0,
            "countryTax": 0,
            "amountIsFinal": false,
            "stateTax": 0,
```

```
            "discounted": false,
            "cityTax": 0,
            "districtTax": 0
        },
        "profileId": "120023",
        "creationTime": 1403734364000,
        "relationships": [
            {
                "amount": 0,
                "id": "r20003",
                "returnedQuantity": 0,
                "relationshipType": "SHIPPINGQUANTITY",
                "shippingGroupId": "sg20005",
                "quantity": 2,
                "commerceItemId": "ci2000007"
            },
            {
                "amount": 0,
                "id": "r20004",
                "returnedQuantity": 0,
                "relationshipType": "SHIPPINGQUANTITY",
                "shippingGroupId": "sg20005",
                "quantity": 1,
                "commerceItemId": "ci2000008"
            },
            {
                "id": "r20005",
                "amount": 261.9,
                "relationshipType": "ORDERAMOUNTREMAINING",
                "paymentGroupId": "pg20005",
                "orderId": "o20005"
            }
        ],
        "totalCommerceItemCount": 2
    }
}
```


Order Management REST APIs

Once the order data is successfully received by the order management system, any further processing of the order occurs in the OMS. For example, the status of the order changes in the OMS when payment is received and when the order is shipped.

To keep the order up to date in Oracle Commerce, the OMS can submit requests to the endpoints of the Orders resource when changes occur to the order. Typically, the update will involve changing the values of properties that store information about the state of either the order itself or components of the order, such as shipping groups. For example, when the OMS begins processing the order, it can use a `PUT` request to change the state property of the order object from `SUBMITTED` to `PROCESSING`:

```
PUT /ccadmin/v1/orders/o10406 HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en
Content-Type: application/json
```

```
{"state": "PROCESSING"}
```

The properties of an order are described in the documentation for the `/ccadmin/v1/orders/{id}` endpoint. (See Learn about the APIs for information about accessing endpoint documentation.) For information about the available states for orders and order components, see Understand order states.

Keep the following in mind when writing PUT requests with the Commerce REST APIs:

- The request typically does not need to include properties you are not updating. However, if the request includes a list or map property, that property must contain references to all the members of the list or map that should be retained. Even if you want to update only one item out of a list of 20, you must provide enough data to match the other 19 existing list members.
- References to an existing list member must contain enough data to match the item with an existing item. If no match can be found, a new item will be created.
- A request that includes a representation of an empty list or map removes all members of that list or map.

Return Request webhooks

An order or portion of an order is eligible for return only if it has been fulfilled and the items to be returned have not previously been returned. Even if it meets these conditions, the order may not be returnable (for example, if too much time has elapsed since the order was fulfilled), or individual items may not be returnable (for example, if a product or SKU's Not Returnable property is set to `true`, or if the OMS determines an item cannot be returned).

Oracle Commerce includes two webhooks that you can use to communicate with an order management system to process returns:

- The Return Request Validation webhook is a function webhook that queries the OMS to determine whether the order is returnable, and receives data back from the OMS indicating which items can be returned.
- The Return Request Update webhook is an event webhook that submits a return request to the OMS when it is initiated by a shopper or agent.

These webhooks can be used together to help manage returns. For example, the Return Request Validation webhook can be used to determine which items in the order are returnable, and therefore are eligible for inclusion in the return request. The Return Request Update webhook can then be used to notify the OMS once the return request is created.

To enable these webhooks, you configure them with the URL where your OMS or gateway listens for requests. The webhooks send data to the OMS as `POST` requests containing JSON data. The OMS may need to be configured to convert the JSON data into the system's native format.

Note: Oracle Commerce also provides versions of the Return Request Validation and Return Request Update webhooks that exclude payment details from the order data you send to systems that do not comply with PCI DSS. See Understand webhooks and PCI DSS compliance for more information.

Return Request Validation webhook

The Return Request Validation webhook is automatically invoked when Commerce needs to determine which items in an order are returnable. (For example, when displaying the order history, the webhook is used to determine whether to display a Return button next to an order). The webhook can also be invoked manually using the Return Requests endpoints in the Store REST API. These endpoints enable your store to accommodate a variety of different return workflows, such as shopper-initiated returns.

The webhook payload includes a context property that specifies the operation being performed. The following are the valid values for this property when validating a return request:

- `Initiate_Return` -- check if the order and its items are eligible for return
- `CalculateRefund_Return` -- determine refund amount
- `Submit_Return` -- validate the return request prior to creating it or saving the changes
- `Custom_Return` -- trigger the webhook from the Store API

The following are the valid values for this property when validating an exchange request:

- `Initiate_Exchange` -- check if the order and its items are eligible for exchange
- `Submit_Exchange` -- validate the exchange request prior to creating it or saving changes
- `Process_Exchange` -- validate that the exchange should be fulfilled once the returned goods are received

The following example shows a portion of the webhook payload that lists the items to determine the return eligibility of:

```
{
  "context": "Initiate_Return",
  "returnRequests": [
    {
      "returnItemList": [
        {
          "quantityToReturn": 0,
          "commerceItemId": "ci3000416",
          "quantityAvailable": 20,
          "quantityShipped": 20,
          "productId": "Product_27Fxyzii",
          "nonreturnable": true,
          "returnReason": null,
          "shippingGroupId": "sg40414",
          "catalogRefId": "Sku_27Gxyzii",
          "nonReturnableReason": "This is a non-returnable item."
        },
        {
          "quantityToReturn": 0,
          "commerceItemId": "ci3000417",
          "quantityAvailable": 15,
          "quantityShipped": 20,
          "productId": "Product_36Exy",
          "nonreturnable": false,
          "returnReason": null,
          "shippingGroupId": "sg40414",
          "catalogRefId": "Sku_36Fxy",
          "nonReturnableReason": null
```

```
        }
      ],
. . .
```

The response from the OMS indicates whether the individual items are eligible for return, and whether the order as a whole is returnable. (For example, the items might be eligible for return, but the order might not be returnable if too much time has elapsed since it was fulfilled.) The response can optionally include a return authorization number, a tracking number, a URL for accessing a return shipping label, and additional data that can be displayed to the shopper. It can also override the values of the nonreturnable flag from the request.

The following is a sample response returned by the OMS:

```
{
  "context": "Initiate_Return",
  "returnRequests": [
    {
      "returnItemList": [
        {
          "shippingGroupId": "sg40414",
          "productId": "Product_27Fxyzii",
          "nonreturnable": true,
          "nonReturnableReason": "This is a returnable item",
          "additionalProperties": {
            "name1": "value1",
            "name2": "value2"
          },
          "quantityAvailable": 1,
          "commerceItemId": "ci3000416"
        },
        {
          "shippingGroupId": "sg40414",
          "productId": "Product_36Exy",
          "nonreturnable": false,
          "nonReturnableReason": "This is a returnable item",
          "additionalProperties": {
            "name1": "value1",
            "name2": "value2"
          },
          "quantityAvailable": 1,
          "commerceItemId": "ci3000417"
        }
      ],
      "orderId": "o30411",
      "rma": "12345",
      "trackingNumber": "1234567890",
      "returnLabel": "a5445afg5",
      "nonReturnableReason": "Order exceeded no of days",
      "nonreturnable": false,
      "additionalProperties": {
        "name1": "value1",
        "name2": "value2"
      }
    }
```

```
  ]
}
```

Notice that this response indicates the individual items are both eligible for return (that is, they are not nonreturnable items), and that the order as a whole can be returned.

Return Request Update webhook

When a customer service agent or a shopper creates a return request or exchange request, the Return Request Update webhook sends a request to the OMS. The body of the request contains the following data:

- The complete order data from the original Order Submit `POST` request.
- The new or updated return or exchange request.
- For an exchange request, the new order data from the exchange.

The order management system should return an HTTP status code indicating whether the data was received successfully. A 200-level status code indicates the `POST` was successful. Any other code indicates failure; if this occurs, Return Request Update sends the request again. The webhook is executed up to five times until it succeeds or gives up.

Update the return request

Once the return request is received, the OMS is responsible for managing the return process. However, you may want Oracle Commerce to receive progress updates, so you can reflect the status of the return in the shopper's order history.

To update the return request in Commerce, the OMS can submit requests to the `updateReturnRequest` endpoint in the Admin API. For example, after the OMS authorizes the return, it can send Commerce the tracking number and related information:

```
PUT /ccadmin/v1/returnRequests/200001  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en
Content-Type: application/json

{
  "shippingTaxRefund": 0,
  "agentId": "service",
  "secondaryCurrencyShippingTaxRefund": 0,
  "actualShippingRefund": 12.5,
  "actualTaxRefund": 4,
  "otherRefund": 0,
  "secondaryCurrencyActualTaxRefund": 0,
  "refundMethodList": [
    {
      "refundType": "manualRefund",
      "amount": 66.49,
      "state": "INCOMPLETE"
    }
  ],
  "returnLabel": "https://www.example.com/returnLabel/234977gege4",
  "authorizationNumber": "200001",
  "returnFee": 0,
  "requestId": "200001",
  "secondaryCurrencyActualShippingRefund": 0,
```

```
  "links": [
    {
      "rel": "self",
      "href": "http://www.example.com/ccadmin/v1/returnRequests/200001"
    }
  ],
  "state": "PENDING_CUSTOMER_ACTION",
  "additionalProperties": {
    "key1": "value1"
  },
  "originOfReturn": "contactCenter",
  "trackingNumber": "178923",
  "returnItemList": [
    {
      "secondaryCurrencyActualTaxRefundShare": 0,
      "comments": null,
      "shippingGroupId": "sg70428",
      "secondaryCurrencyActualShippingSurchargeRefundShare": 0,
      "quantityWithFractionReceived": 0,
      "commerceItemId": "ci6000446",
      "secondaryCurrencyActualShippingRefundShare": 0,
      "actualShippingSurchargeRefundShare": 0,
      "returnReason": "defective",
      "actualShippingRefundShare": 12.5,
      "state": "AWAITING_RETURN",
      "additionalProperties": {},
      "actualTaxRefundShare": 4,
      "quantityReceived": 0,
      "refundAmount": 49.99
    }
  ],
  "actualShippingSurchargeRefund": 0,
  "secondaryCurrencyActualShippingSurchargeRefund": 0
}
```


At later stages of the return process, the OMS can use this endpoint to provide further updates, such as notification when the returned item has been received, and notification when a refund has been issued. Typically the OMS will update the return state (which describes the return as a whole) and the return item states (which describes each return item individually) at these times.

The following are the valid return states and return item states:

- Return states: `MANUAL_REFUND`, `PENDING_REFUND`, `INCOMPLETE`, `COMPLETE`, `FULL_RETURN`, `PARTIAL_RETURN`, `PENDING_CUSTOMER_ACTION`.
- Return item states: `RETURN_NOT_REQUIRED`, `AWAITING_RETURN`, `PARTIAL_RETURN`, `RETURNED`, `INITIAL`.

By default, the following are the valid values for the `returnReason` property, which indicates why the shopper returned the item: `defective`, `didNotLike`, `didNotMeetExpectations`, `incorrectColor`, `incorrectItem`, `incorrectSize`. You can

use the Reasons endpoints in the Admin REST API to add, delete, or inactivate individual return reasons. For example, to make the `defective` value inactive:

```
PUT /ccadmin/v1/reasons?type=returnReasons&id=defective
Authorization: Bearer <access_token>
x-ccasset-language: en
Content-Type: application/json

{
  "readableDescription": "Defective",
  "active": false,
  "description": "defective"
}
```

Note that the `getReturnRequest` and `updateReturnRequest` endpoints cannot be used in preview mode. For more information about these endpoints, including a list of the fields that can be updated, see the REST API documentation in the Oracle Help Center.

Update refund amounts

Refund amounts can be calculated and issued by either Commerce or by the OMS system. To calculate and issue refunds in Commerce, use the Agent Console or the `receiveReturnRequest` endpoint in the Agent API to indicate that the returned items have been received. Then start the refund process by calling the `updateReturnRequest` endpoint in the Agent API with the `op` value set to `initiateRefund`.

If refunds are calculated and issued by the OMS, use the Agent Console or the `receiveReturnRequest` endpoint in the Agent API to indicate that the returned items have been received. Then use the `updateReturnRequest` endpoint in the Admin API to update the refund-related fields in Commerce, and the `updateOrder` endpoint in the Admin API to update the `amountCredited` in the payment group of the original order.

Understand order states

This section lists the possible states of an order and of order subobjects.

The following table describes the possible states of an order:

State Name	Description
AGENT_REJECTED	When using order approvals, this indicates that the order has been rejected by the agent.
APPROVED	When using order approvals, this indicates that the order has been approved by the agent.
APPROVED_TEMPLATE	When using order approvals, this indicates that the scheduled order has been approved by the agent.
BEING_AMENDED	When using order approvals, this indicates that the order is being amended.
FAILED	The order failed.
FAILED_APPROVAL	When using order approvals, this indicates that the order has failed to get approval.
FAILED_APPROVAL_TEMPLATE	When using order approvals, this indicates that the scheduled order has failed to get approval.
INCOMPLETE	The order is still in the purchase process.

State Name	Description
NO_PENDING_ACTION	The order has been fulfilled, and processing of the order is complete. All shipping groups in the order are in a NO_PENDING_ACTION or REMOVED state, and order payment has been settled.
PENDING_AGENT_APPROVAL	When using order approvals, this indicates that the order is still awaiting review by the agent.
PENDING_APPROVAL	When using order approvals, this indicates that the order is still pending approval.
PENDING_APPROVAL_TEMPLATE	When using order approvals, this indicates that the scheduled order is still pending approval.
PENDING_CUSTOMER_ACTION	Processing of the order requires the customer's attention for some reason, such as an incorrect customer address.
PENDING_CUSTOMER_RETURN	Processing of the order requires the customer return submission.
PENDING_MERCHANT_ACTION	Processing of the order requires merchant attention for some reason, such as the failure of a payment group in the order.
PENDING_PAYMENT	When using account-based commerce, this indicates that the order is awaiting payment information.
PENDING_PAYMENT_TEMPLATE	When using order approvals, this indicates that the scheduled order is awaiting payment information.
PENDING_QUOTE	When using account-based commerce, this indicates that the quote for the order is still pending.
PENDING_REMOVE	A request was made to remove the order. The order is placed in this state until all shipping groups in the order are set to a PENDING_REMOVE state.
PROCESSING	The order is being processed by the order management system.
REMOVED	The order has been removed successfully.
SUBMITTED	The order has completed the purchase process and has been submitted to the order management system.
QUOTED	When using account-based commerce, this indicates that the order has been quoted.
QUOTE_REQUEST_FAILED	When using account-based commerce, this indicates that the quote for the order has failed.
REJECTED	When using account-based commerce, this indicates that the order has been rejected.
REJECTED_QUOTE	When using account-based commerce, this indicates that the quote for the order has been rejected.

In addition, the sections below describe the possible states of various order components:

`paymentGroup` states

An order's array of payment groups represent the payments that paid for the order. The following table describes the possible states of a payment group:

State Name	Description
AUTHORIZE_FAILED	Authorization of the payment group has failed.
AUTHORIZED	The payment group has been authorized and can be debited.
CREDIT_FAILED	Credit of the payment group has failed.
INITIAL	The payment group has not been acted on yet.
REMOVED	The payment group has been removed.
SETTLE_FAILED	Debit of the payment group has failed.
SETTLED	The payment group has been debited successfully.

`shippingGroup` states

A shipping group represents a shipment and includes `commerceItemRelationships` that represent which commerce items in what quantities are included in the shipment. The following table describes the possible states of shipping group:

State Name	Description
INITIAL	The shipping group is in a pre-fulfillment state.
PROCESSING	The shipping group has started the fulfillment process.
PENDING_REMOVE	A request for the removal of the entire order was made, and the removal of this shipping group is possible.
REMOVED	The shipping group has been removed.
FAILED	The shipping group has failed to process.
PENDING_SHIPMENT	The shipping group is ready to be shipped.
NO_PENDING_ACTION	The shipment of all the items in the shipping group is complete.
PENDING_MERCHANT_ACTION	An error occurred while trying to process the shipping group; the error requires the merchant's attention.

`commerceItem` states

Commerce items (sometimes referred to as line items) represent the SKUs included in an order. The following table describes the possible states of a `commerceItem`. These states are read-only and are calculated from the shipping group's `commerceItemRelationship`.

State Name	Description
BACK_ORDERED	The item is not available in the inventory; it has been backordered.
DISCONTINUED	The item is not available in the inventory; it cannot be backordered.
FAILED	The item has failed.
INITIAL	The item is in an initial state, that is, it is not yet associated with any shipping group.
ITEM_NOT_FOUND	The item could not be found in the inventory.
OUT_OF_STOCK	The item is not available in the inventory, and it has not been backordered.

State Name	Description
PENDING_REMOVE	The item will be removed pending verification that all item relationships referring to it can be removed.
PRE_ORDERED	The item is not available in the inventory; it has been preordered.
REMOVED	The item has been removed from the order.
SUBITEM_PENDING_DELIVERY	The item is available in the inventory, and it is being prepared for shipment to the customer.

Create custom properties for orders

This section describes how to add custom properties to orders.

This section describes how to use the Oracle Commerce REST web services APIs to add custom properties to orders. See Use the REST APIs for information you need to know before using the services.

Understand order types

Like shopper profiles, orders include a predefined set of properties. For example, orders have properties for storing data about when the order was submitted, the cost of the items, shipping information, and so on.

Just as the properties of a shopper profile are determined by its associated shopper type, the properties of an order are determined by its associated order type. The order type serves as a template for the order. Currently only one order type, whose ID is `order`, is available. This order type is associated with all Oracle Commerce orders.

You cannot create additional order types, but you can add custom properties to the `order` order type. For example, you could add a `gift_message` property that a customer can use to provide a note to include in the package when the order items ship.

You can use the Oracle Commerce Admin API to add custom properties to the `order` order type. The Order Types resource in the Admin API includes endpoints for creating and working with custom properties of the `order` order type, and the Orders resource in the Admin API includes endpoints that you can use to set the values of properties of individual orders, including custom properties that have been added to the `order` order type.

When you add a custom property to the `order` order type, the property is added to all orders, including any new orders customers create and any orders that already exist.

Note: Many of the properties of orders store either arrays or pointers to other resources. The Order Types endpoints do not expose these properties. You can add custom properties to the `order` order type and modify them using the Order Types endpoints, but you can only create and modify top-level scalar properties.

View an order

To view an existing order, first log into the Admin API on the administration server using an account that has the Administrator role. For example:

```
POST /ccadmin/v1/mfalogin HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=admin1@example.com&password=A3ddj3w2&totp_code=3
65214
```

Then issue a GET request to the `/ccadmin/v1/orders/{id}` endpoint, providing the ID of the order you want to view, and including the access token that was returned by `/ccadmin/v1/mfalogin`. For example:

```
GET /ccadmin/v1/orders/o10008 HTTP/1.1
Authorization: Bearer <access_token>
```

The response shows the predefined order properties that are exposed by Oracle Commerce, and the values of the properties in the order. You can modify the values of these properties for an order using the `PUT /ccadmin/v1/orders/{id}` endpoint on the administration server.

View an order type

To view an order type, issue a `GET` request to the `/ccadmin/v1/orderTypes/{id}` endpoint on the administration server. The following example illustrates calling this endpoint with `order` (the only order type currently available) specified as the value for `id`:

```
GET /ccadmin/v1/orderTypes/order HTTP/1.1
Authorization: Bearer <access_token>
```

Add custom properties to an order type

To add custom properties to an order type, issue a `PUT` request to the `/ccadmin/v1/orderTypes/{id}` endpoint on the administration server. Use the following format:

- The request header must specify the `x-ccasset-language` value.
- The request body is a map where each key is the name of a new property, and each value is an object that specifies the values of the attributes of the property.
- Each object is also a map, with each key being the name of an attribute and each value being the corresponding attribute value.

The attributes of order type properties that you can set through Order Types endpoints are the same as the attributes of shopper type properties you can set through Shopper Types endpoints. See Settable attributes of shopper type properties for descriptions of these properties.

The ID of a custom property must include the underscore character (_). This ensures that the ID will not conflict with any properties that Commerce adds to orders in the future. The endpoint produces an error if you attempt to create a custom property without an underscore in its ID.

The following example shows a sample request for adding a custom property to the `order` order type:

```
{
 "properties": {
     "gift_message": {
         "label": "Enter an optional gift message here:",
         "type": "richText",
         "uiEditorType": "richText",
         "internalOnly": false,
         "required": false
         }
} }
```

The following is a portion of the response that shows the new property:

```
{
  ...
    "properties": {
     ...
       "gift_message": {
            "writable": true,
            "localizable": false,
            "label": "Enter an optional gift message here:",
            "type": "richText",
            "uiEditorType": "richText",
            "textSearchable": false,
            "multiSelect": null,
            "dimension": false,
            "internalOnly": false,
            "default": null,
            "editableAttributes": [
                "textSearchable",
                "multiSelect",
                "dimension",
                "internalOnly",
                "default",
                "label",
                "required",
                "searchable"
            ],
            "length": 4000000,
            "required": false,
            "searchable": false
         },
         ...
     }
     ...
}
```

Implement robust order capture

Oracle Commerce includes robust order capture tools to ensure that orders are persisted and problems can be resolved if there are errors during the payment or order submission process.

The tools available depend on the options specified in the Setup tab in the Payment Processing settings in the administration interface. This tab has a Payment Options drop-down with two options:

- Allow Partial Payment/Early Persist -- When a shopper attempts to submit an order, Commerce saves it in the `PENDING_PAYMENT` state before processing payments.
- Full Payment Required -- When a shopper attempts to submit an order, the order state is `INCOMPLETE` until all payments are authorized.

The Setup tab also includes a Price Hold Period field. If Allow Partial Payment/Early Persist is selected, the Price Hold Period setting specifies the amount of time the shopper is given to provide missing payment information before an order is cancelled. The order's prices will not change during this interval.

Use early persist

If the Allow Partial Payment/Early Persist option is selected, when a shopper submits an order, the order is initially placed in the `PENDING_PAYMENT` state. If a payment fails but Commerce does not receive an error, the order remains in this state. During the price hold period, the shopper can add payments to the order, but cannot make other changes or delete the order. A record of the order is visible in the Agent Console, so an agent can also attempt to resolve the issue. At the end of the price hold period, if the order is still in the `PENDING_PAYMENT` state, it is marked for cancellation.

For example, the order processing might proceed as follows:

1. A shopper begins the checkout process using a web checkout system.
2. The shopper enters payment information on the payment provider’s screen.
3. An error occurs with the payment that prevents the order submission from completing successfully.
4. The order remains in the `PENDING_PAYMENT` state, and the shopper does not receive an order confirmation.
5. The shopper can see the order in the `PENDING_PAYMENT` state and resubmit the order with payment. The merchant can also see the order and take steps to resolve the issue.

To support this option, you must ensure that your order history and checkout widgets can handle the `PENDING_PAYMENT` state, and that your storefront allows the shopper to see order issues and take corrective action. For example, the storefront should enable the shopper to see order status, update payment information, and resubmit an order that has not been successfully submitted. On storefronts built using the Open Storefront Framework, the Allow Partial Payment/Early Persist option is recommended, and the included widgets are designed to handle these situations.

On storefronts built using Storefront Classic, you will need to use the Split Payment widget and customize it and other widgets on your storefront to support these behaviors. See Enable Split Payments for more information.

Note that if an anonymous shopper has a partially paid order, the order must be completed before the session times out, because the shopper has no way to access the order when

returning to the site later. However, if the Order Payment Initiated email is enabled, the shopper is sent an email with the order ID when the order is created, and can supply this ID to a customer service agent for completing the order.

Use full payment required

The Full Payment Required option should be used only on storefronts built using Storefront Classic. If this option is selected, and authorization for any payment method fails, an error is displayed, and any other authorizations that succeeded are voided. The shopper must re-enter all payments and resubmit the order.

However, if Commerce for some reason does not receive a response about an authorization, it cannot determine if there has been an error, and the order remains in the `INCOMPLETE` state. If the shopper then tries to remove the items from the shopping cart, the shopper is prevented from removing the last item, and the order is saved in the `FAILED` state if the system is aware of the payment attempt. The order can be viewed on the Order History page of the storefront and in the Agent Console.

Support zero-cost orders

You can enable your Commerce sites to handle orders whose price is zero.

On some Commerce sites, it is possible for the total cost of an order to be zero. For example, the customer may have a coupon for a free item, or the site may offer free samples, such as color swatches on a fabric site or a chapter of an electronic book on a bookstore site.

To support these situations, Commerce makes it possible for a shopper to bypass providing payment information when placing a zero-cost order. By default, if the total cost of an order is zero, the shopper does not need to provide payment information to place the order. This behavior is controlled by the `isPaymentsDisabled` function in the `OrderViewModel`:

```
OrderViewModel.prototype.isPaymentsDisabled = function(){
    var self = this;
    var disableRules = self.cart().total() == 0;
    if(disableRules){
       $.Topic(pubsub.topicNames.PAYMENTS_DISABLED).publish();
    }
    return disableRules;
}
```

As you can see in the code above, if the total cost of the shopping cart is zero, the function's `disableRules` variable is set to `true`, and a message is published to the `PAYMENTS_DISABLED` topic. Payment widgets can subscribe to this topic and disable payment inputs when a message is published to it.

If you want to require shoppers to provide payment information even when the order cost is zero, your payment widgets can ignore the `PAYMENTS_DISABLED` topic, so that payment inputs are not disabled. Requiring payment information is desirable for situations that involve recurring billing. If you capture an order that has no upfront charges, you will still need to collect payment information and include it with the order. This way a token will be returned, saved with the order, and passed to the fulfillment system using the Order Submit webhook.

Note that gift card and credit card payment widgets have a `triggerValidations` flag that determines whether to validate the required payment information, such as the card number and CVV. You can change the setting of this flag depending on the behavior your sites require.

See Understand widgets for information about extending widgets and view models.

Support shopper-initiated order management

Once an order is created, Oracle Commerce provides functionality that allows the shopper to manage the order.

This section describes how to configure Commerce so that shoppers can initiate cancellations, returns, or exchanges on orders they have placed through their account.

Support shopper-initiated cancellations

You can configure the Order Details widget on the Order Details layout to enable shopper-initiated cancellations. Once this is done a shopper may initiate a cancellation on an order they have placed provided the following criteria are met:

- The remorse period is enabled in the Admin settings for your site.
- The order is within the remorse period.
- The order status is QUEUED, or all of the following status criteria are met:
  - The order status is not CANCELLED, NO_PENDING_ACTION, PENDING_APPROVAL, PENDING_REMOVE, QUEUED, or REMOVED.
  - The `paymentGroup` status is not SETTLED.
  - The `shippingGroup` status is not NO_PENDING_ACTION.
  - No item within the order has a status of DELIVERED.

If each of these criteria are met, a **Cancel Order** button is displayed to the shopper when they review an order on the Order Details layout. When the shopper clicks on the button, they are asked to select a reason for cancelling the order and to confirm the cancellation. Once the cancellation is confirmed, Commerce uses the order cancellation webhook to instruct the fulfillment system to cancel the order and updates the order status accordingly. An email is then sent to the shopper telling them that their order has been cancelled.

To configure shopper-initiated cancellations on your site you must:

- Enable the remorse period for your site. For further information on enabling the remorse period, refer to the Set the customer remorse period.
- Ensure you have an Order Details layout that contains an Order Details widget instance. From the Design page, open the Grid View for the Order Details layout and click on the Settings icon on the Order Details widget instance. Check the Enable Shopper Initiated Cancel box and click Save.

If you need to add the Order Details widget to the Order Details layout on your site, refer to the Add widgets to a layout section of Customize your store layouts for more information.

Support shopper-initiated in-flight cancellations

You can configure Oracle Commerce so that user can cancel an order post the remorse period, using the extended remorse period option. The user can also cancel an order post the remorse period, that is, after the order is submitted for fulfillment by Order Submit Webhook.

To configure shopper-initiated in-flight cancellations on your site you must enable the Extended Remorse Period in the admin settings for your site. From the Oracle Commerce administration UI, **Agent Console Settings**, **Extended Remorse Period**, click the checkbox **Enable Extended Remorse Period**, and specify the number of days from the date of submission during which orders can be cancelled.

The `pointOfNoRevision` property, which is updated by the fulfillment system and when set to `true`, indicates that the item has passed the point at which the fulfillment system can prevent the item from being shipped or provisioned. When a shopper selects to cancel an inflight order, if any item in the order has passed the point of no revision (`pointOfNoRevision = true`) then the system returns an error indicating that this order cannot be cancelled.

Note: Remorse Period and Extended Remorse Period are independent settings that are not mutually exclusive, in that both periods start from the time that the shopper places the order. While the Remorse Period actually delays the submission of the order to the fulfillment system, the Extended Remorse period applies even if the order has already been submitted to fulfillment.

Setup for the `validateCancel` SSE

You need to setup the SSE `validateCancel` or shopper-initiated in-flight cancellations would not be able to validate if the order is within the `extendedRemorsePeriod`. The SSE `validateCancel` is only triggered with the sample widget and not the OOTB widget.

Below SSEs are available in the Admin Developer: `Validate-cancel-app-store.zip` (storefront), `Validate-cancel-app-agent.zip` (Agent), and `OrderQualification.zip` . The `Validate-cancel-lib.zip` SSE (required), is included in other two SSEs. You do not need not download it.

Download the SSE `validateCancel` by clicking **Admin UI**, **Developer** tab. Once you download the SSE, you will need to edit the configuration file in SSE. When you download each SSE, refer to the README file for instructions.

Note: You must upload these SSEs to the node server. You need not download the library file here, since the library file contains the classes and would be packaged as node-modules when either download of the store/agent SSEs.

How to initiate and submit flow for cancel order

To initiate a Cancel Order, you can use this endpoint:

```
/ccstoreui/v1/orders/initiateCancelOrder POST
/ccagentui/v1/orders/initiateCancelOrder POST
```

Or, if you are using custom widget, after making a call to the SSE, which validates if the order is still within the `ExtendedRemorsePeriod`, you can click the **Cancel order** button in the custom widget Order details.

When you click the **Cancel order** button in the order details page (after remorse period has expired and within extended remorse period), following occurs:

1. A Cancel order is created (clone of original order with prices made zero)
2. An in-memory return request is created. This means a return request object that is not persisted and just stays in memory. This return request facilitates the refunds for items with one time price.

3. The Cancel summary page displays the details of the cancel order and return request. **Note:** This cancel order summary is an OOTB widget. The order details widget has code to handle the invocation of the `initiateCancelOrder` endpoint and redirection to this page and display this widget.

To submit a Cancel Order, you can use this endpoint:

```
/ccstoreui/v1/orders/submitCancelOrder PUT
/ccagentui/v1/orders/submitCancelOrder PUT
```

Or you can use the Cancel order summary widget in cancel checkout page

Once the user reviews the Cancel order summary and clicks on **Confirm** button, the submit cancel order flow is triggered and the following occurs:

1. The Order qualification webhook is triggered which will trigger the Order qualification pipeline and the validate cancel module will check if the cancellation is still allowed. You need to use this webhook to check whether the order is still in extended remorse period. If you do not use this webhook, it will not check for `extendedRemorsePeriod`. You will need to configure this webhook before using this feature, using the latest version of `OrderQualification` SSE.
2. Other validations are performed to check if there is any change in the original order.
3. When validation passes, the in-memory return request (created in the initiate cancel order) is submitted and confirmed. **Note:** The first time the return request is created, it is just to return the refund values and is not persisted on the server. On submit cancel, the return request is created again and that one is submitted and confirmed. All items in this return request will be marked as `returnRequired = false`. This is because in 19A, cancel in-flight can be triggered only when all items have `PONR=false`. Therefore, none of these items need to be returned (since they never got shipped). The return request is created for the sole purpose of refund.
4. The cancel order is then submitted (as a zero value order) using the existing submit order flow and sent to the fulfillment system for further processing. The state of the original order is changed to REMOVED, while the order still remains in the database.

Note: If you use the Cancel order summary widget in cancel checkout page, and if before submitting, you try to navigate to some other page, you cannot resume your cancellation. You will have to initiate the cancel again and submit the order. For orders for which cancel was initiated but not completed, a scheduler runs to clean them up.

Initiate the appropriate Terminate asset operation in the asset management system (CPQ)

When an in-flight order is cancelled, some items in that order may be assets in Oracle CPQ and need to be terminated. Manage this by using the `Order Asset Operation` SSE and related webhooks:

Server Side Extension Endpoint `Order Asset Operation`

Item	Description
SSE	Order Asset Operation
Endpoint Name	Order Asset Operation
Trigger	The endpoint can be triggered by the Order Asset Operation webhook.

Item	Description
Inputs:	Order - the original order that is being cancelled. Operation - Terminate. This webhook may be used for `Terminate` and `Get Assets`
Returns	Collection of items containing: BOM - A configured item or Bill of Materials that represents the terminate asset instructions. Asset ID - The asset ID to which the BOM relates. Commerce Item ID - the ID value for the order item that corresponds with the asset record, that is, the order item that resulted in the creation of the asset.

Note the following:

- If any item in the original order has become an asset in CPQ, the cancel order flow will take care of the termination of those assets.
- Sub items of an configurable item (with `soldAsPackage=false`) can be tracked individually for return.

  If an configurable item in the original order has `soldAsPackage=false`, each sub item will have its own `shippingGroupCommerceItemRelationships`. When a return request is created there will be corresponding return items for those sub items, which will help in tracking the return status of the sub items when return is initiated on the root item.

  Note: Return of a sub item alone is not supported.

Non-Shippable items

Non-shippable items are items that cannot be physically shipped, such as mobile phone tariffs, IPTV packages, downloadable items, etc. For these items, the administrator can set the shippable flag at the item level. From Oracle Commerce administration, select **Catalog**, **item**, **General**, and then click on the **Shippable** checkbox. The Shippable property is selected by default and must be unchecked to indicate that a product is non-physical.

Support shopper-initiated returns

In addition to shopper-initiated cancellations, you can also configure your storefront to allow shoppers to initiate returns of fulfilled orders. See Order Details for information about how to do this.

Note that some items may not be returnable. If a product or SKU's Not Returnable property is set to `true`, neither a shopper nor an agent can initiate a return for that item. If order information is maintained in an external system, you can use the Return Request Validation webhook to communicate with the system to determine whether an order can be returned. For more information about this webhook, see Return Request webhooks.

The Agent Console also includes tools that enable a customer service agent to initiate a return or exchange on behalf of a shopper. See Understand the Agent Console for information about these tools.

Enable returns on partially fulfilled orders

Commerce can optionally support returns on partially fulfilled orders.

By default, an item received by a shopper can be returned only if the order that it is part of has been fulfilled completely. For example, suppose an order contains two items, and only one item is shipped to the shopper because the other item is out of stock. If the shopper decides to return the first item, he or she must wait until the second item arrives.

This section describes how to enable returns on partially fulfilled orders, so shoppers can return received items even if not all of the items in the order have been received. Note that this feature is currently supported only through the REST APIs.

Enable returns

If you want to allow shoppers to return items in orders that have been only partially fulfilled, use the `updateCloudConfiguration` endpoint in the Admin API to set the `allowReturnOnPartiallyFulfilledOrder` property to `true`. For example:

```
PUT /ccadmin/v1/merchant/cloudConfiguration  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
  "allowReturnOnPartiallyFulfilledOrder": true
}
```

Modify the order state

This example illustrates using the Admin API to mark one of the items in an order as delivered. The order contains two items, but only one item has been delivered because the other is out of stock. Both items have the same shipping address and use the same shipping method.

1. Call the `getOrder` endpoint in the Admin API to view the order. For example:

   ```
   GET /ccadmin/v1/orders/o20005
   ```

   The response shows the data from the order. The order should have one shipping group with two `commerceItemRelationships` objects (a separate object for each item).

2. Call the `updateOrder` endpoint in the Admin API and update the state of the `commerceItemRelationships` object associated with the delivered item to `DELIVERED`.

Initiate the return

To initiate the return of the delivered item, you use either the Agent API or the Store API. This section illustrates using the Agent API.

1. Call the `handleOrderActions` endpoint in the Agent API to check if the order is returnable. For example:

   ```
   POST /ccagent/v1/orders/o20005
   {"op": "validActions"}
   ```

The response should show the `isReturnOrder` flag as `true`. If `isReturnOrder` is `false`, items in the order cannot be returned.

2. Initiate a return request for the above order using the Agent API `initiateReturn` endpoint. For example:

```
POST /ccagent/v1/returnRequests
{
    "op": "initiateReturn",
        "orderId": "o20005"
}
```

The response includes a list of the items in the order. For an item that has not been delivered yet, the `nonreturnable` property is `true` and the `nonReturnableReason` property indicates that the item has not been delivered.

3. Now create the return request using the `initiateReturn` endpoint with `"op": "createReturnRequest"`, and specify the item to be returned.
Once the return request is created, the remainder of the return flow (such as receiving the returned item and processing the refund) can be performed using the UI.

Support add-on products

Add-on products are optional extras, like monogramming, gift wrap, or warranties, which shoppers can purchase to customize or enhance purchases.

An add-on is a product that you can link to a main product so shoppers see it on the main product's details page and can optionally purchase it along with the main product. See Create add-on products to learn how to create an add-on product and link it to a main product.

You must make changes to several storefront layouts to allow your store to support add-on products. The modifications described in this section involve adding new widgets to page layouts and also making sure the latest versions are used for some widgets that are included in the page layouts out of the box. To determine if you are using the latest version, or to replace a widget with the latest version, see Customize your store layouts.

The following widgets incorporate add-on products functionality into your storefront:

- The Order Confirmation and Order Details widgets have been updated to support add-on products. Make sure you are using the latest version of these widgets, which allow a shopper to see any add-on products that are part of the order.
- The Product Details widget must be updated to display add-on products that shoppers can select, customize (if appropriate), and add to the cart along with the main product. See Product Details widget for add-ons for more information, including a sample version of the Product Details widget that lets shoppers select different types of gift wrapping and add a custom gift message.
- The Shopping Cart and Cart Summary widgets have been updated to support add-on products. Make sure you are using the latest versions of these widgets. The new version of the Shopping Cart Summary widget allows a shopper to see, but not remove or edit, any add-on products that are part of the order. See Cart

Summary widget for add-ons for a sample version of the Shopping Cart Summary widget that lets shoppers remove or edit add-on products.

Product Details widget for add-ons

There is no one-size-fits-all solution for displaying add-on products in the Product Details layout, so by default, the Product Details layout does not include components for add-on products. To allow shoppers to see and purchase add-on products, you must customize the layout's Product Details widget. This section describes an example based on the Product Details widget that is included in Commerce. The sample updates the widget so it displays the details about all add-on products linked to the main product when the shopper views the main product's details page. If an add-on product offers multiple SKUs, the shopper can select a SKU. If an add-on product allows shopper input, such as a gift message, the shopper can specify that value.

This sample assumes that you have already created and linked add-on products as described in Create add-on products. The add-on products in this sample include two product types, whose IDs are Warranty and GiftWrap. The GiftWrap product type includes a short text Shopper Input property that allows the shopper to add a gift message. Note that the code in this section is for illustrative purposes only; it is not intended to be production-ready, and may not adequately handle all possible use cases or implement the exact behavior you want. In addition, you may need to customize other widgets that handle add-on items.

Access add-on properties via the `productTypesViewModel`

The `productTypesViewModel` is populated with the `ProductTypes` data available from the data initializer. This view model is cacheable, and maintains a cache of `ProductTypes` data. The `productTypesViewModel` supports the following methods:

- `getInstance (data)` gets the instance of the `productTypesViewModel` object. data is an optional object that contains the array of `productTypes` information.
- `setContextData (data)` populates the `productTypes` list from the data fetched from `Repositorydata`. data is an object that contains the array of `productTypes` information.
- `retrieveShopperInputsData (productTypes, success, error)` gets the ShopperInput for the requested productTypes.

Note: The `dynamicProperty` view model is used to store the `shopperInput` data.

Create an element to display an add-on product

The out-of-the-box version of the Product Details widget is separated into elements. (See Fragment a Widget into Elements for more information.) To create an element to display the add-on products, this sample's `template.txt` file provides the HTML rendering code for the element.

```
<!-- ko if: initialized() -->
<div class="col-md-12">
  <!--  ko if: $data.addOnPopulated -->
  <div data-bind="foreach: addOnProducts">
    <br>
    <div style="border: .5px solid #a1a1a1;padding: 10px 40px;border-radius:
 7px;display: inline-block;background-color: #EEEEEE;margin-bottom: 10px;"
 class="col-md-12">
      <div class="col-md-12" style="left: -10px;">
        <input type="checkbox" data-bind="checked: isSelected, disable:
 (stockStatus != 'IN_STOCK' )" />
```

```
           <span data-bind="text: $data.displayName"></span>
      </div>
      <div class="col-md-12" data-bind="if: isSelected">
        <div class="col-md-12">
        <!-- ko with: $data.shopperInput -->
          <!-- ko foreach: $data -->
            <label data-bind="text: $data.label"></label>
            <!-- ko if: ($data.uiEditorType() == "shortText") -->
              <input  class="form-control"
type="text" data-bind="validatableValue: $data.value"><br>
            <!-- /ko -->
            <!-- ko if: ($data.uiEditorType() == "longText") -->
              <textarea class="form-control" data-
bind="validatableValue:
 $data.value"></textarea><br>
            <!-- /ko -->
            <!-- ko if: ($data.uiEditorType() == "number") -->
              <input  class="form-control"
type="number" data-bind="validatableValue: $data.value"><br>
            <!-- /ko -->
            <!-- ko if: ($data.uiEditorType() == "date") -->
              <input  class="form-control" type="date"
data-bind="validatableValue: $data.value"><br>
            <!-- /ko -->
            <!-- ko if: ($data.uiEditorType() == "checkbox") -->
              <input  class="form-control" type="checkbox" data-
bind="checked:
 $data.value, validatableValue: $data.value"><br>
            <!-- /ko -->
            <!-- ko if: ($data.type() == "enumerated") -->
                 <select class="form-control" type="text"  data-
bind="options:
 $data.values,
optionsCaption: $parents[2].listShopperInputPlaceHolderText,
 validatableValue: $data.value" ></select><br>
            <!-- /ko -->
            <!-- Validation message place holder -->
            <div>
              <p class="text-danger" id="CC-shopperInput-error"
              data-bind="validationMessage: $data.value"
role="alert"></p>

            </div>
          <!-- /ko -->
        <!-- /ko -->
        </div>
        <!-- ko if: ($data.addOnOptions && $data.addOnOptions.length >
0) -->

          <!-- ko if: ($data.addOnOptions[0].product.type ==
'GiftWrap' ||
 $data.addOnOptions[0].product.type == 'Normal') -->
            <span data-bind="widgetLocaleText: 'optionsText' "></
span><br>
            <div class="col-md-12" style="display: inline-flex;">
```

```
              <!-- ko foreach: $data.addOnOptions -->
                <div class="col-md-4">
                  <img class="imageSize" data-
bind="productVariantImageSource:
 {src: $data.product, imageType: 'thumb', alt:$data.product.displayName,
 errorSrc:'/img/no-image.jpg', errorAlt:'No Image Found'},
click: $parents[2].addOnIconChanged.bind($parents[2], $parent)" /><br>
              <div style="display:block;word-break:break-all;width:100%;">
                <span data-bind="text: $data.sku.repositoryId"></span>
                <span> - </span>
                <span data-bind="currency: {price: $data.product.listPrice,
currencyObj: $parents[2].site().selectedPriceListGroup().currency,
nullReplace: $parents[2].priceUnavailableText(), prependNull: false}"></span>
                </div>
              </div>
              <!-- /ko -->
            </div>
          <!-- /ko -->
          <!-- ko if: ($data.addOnOptions[0].product.type == 'Warranty') -->
            <div class="col-md-12">
              <!-- ko foreach: $data.addOnOptions -->
                <input type="radio" data-bind="id:
{name: $data.repositoryId}, checked: $parent.selectedAddonSku,
value: $data.repositoryId,
click: $parents[2].addOnRadioChanged.bind($parents[2], $parent) ">
                <span id="cc-add-on-product-name" data-
bind="text : $data.sku.repositoryId "></span>
                <span id="cc-add-on-product-price" data-bind="currency:
{price: $data.product.listPrice,
currencyObj: $parents[2].site().selectedPriceListGroup().currency,
nullReplace: $parents[2].priceUnavailableText(), prependNull: false}"></span>
                </input><br>
              <!-- /ko -->
            </div>
          <!-- /ko -->
        <!-- /ko -->
        <div class="col-md-12" class="text-danger" >
          <br>
          <span data-bind="text: $data.stockValidationMessage "></span>
        </div>
      </div>
    </div>
  </div>
  <!-- /ko -->
</div>
<!-- /ko -->
```

In this sample `element.json` meta-data file, the element is made available for use by the Sample Product Details widget:

```
{
 "inline" : false,
 "supportedWidgetType" : ["sampleProductDetails"],
 "translations" : [
```

```
 {
  "language" : "en_EN",
  "title" : "addons",
  "description" : "Displaying add-on products in the product details
widget"
 }
 ]
}
```

In order to use the new element in a widget, you need to add some additional tags to the widget's `display.template` and `widget.template` files that enable the element to be rendered as part of the output page and to be managed on the administration interface Design page. If the widget has already been broken into elements, you will, at a minimum, need to add an `oc` section tag for the new element:

```
<!-- oc section: product-addOn -->
    <div data-bind="element: 'sample-product-addOn'"></div>
<!-- /oc -->
```

Add an add-on product to the cart

The cart-item view model has been updated to include the following new fields:

- `isAddOnItem` is a Boolean that is set to `true` for add-on products. This distinguishes between add-on items and Oracle CPQ child items.
- `shopperInput` is a place holder field to capture the shopper input value, such as a gift message.
- `configurablePropertyId` is the repository ID of the `ConfigurableProperty` that corresponds to the selected add-on product.
- `configurationOptionId` is the repository ID of the `ConfigurationOption` that corresponds to the selected add-on product.

When a shopper selects an add-on product displayed on Product Details page and clicks the Add To Cart button, the `addItem` method of `CartViewModel` is triggered for the main product data, which is also the case for products without add-ons. The new field `selectedAddOnProductsObj` contains information that describes the selected add-on products, and is passed to `addItem` with the product.

`addItem` iterates over the `selectedAddOnProductsObj` array and creates a new `CartItem` object corresponding to each `selectedAddon` object. `isAddOnItem` is set as `true` and `shopperInput` is populated if the add-on product contains shopper input data. If no add-on products were selected by the shopper, then the `childItems` property of main product is undefined.

The following sample method iterates over the add-on products structure and trims any options that the shopper did not select before adding the main product and add-on products to the cart.

```
processAddonBeforeAddtoCart: function(addOnProducts) {
  var selectedAddonProducts = [];
  for (var i=0; i<addOnProducts.length; i++) {
    selectedAddonProducts.push(ko.toJS(addOnProducts[i]));
  }
```

```
  // Set the add-on products ShopperInputs
  var iAddonProdsSize = selectedAddonProducts.length - 1;
  var iSelectedSKUsSize = 0;
  for (var i=iAddonProdsSize; i>=0; i--) {
    if(!selectedAddonProducts[i].isSelected) {
      selectedAddonProducts.splice(i, 1);
      continue;
    }

    var shopperInput = {};
    if (selectedAddonProducts[i].shopperInput &&
selectedAddonProducts[i].shopperInput.length > 0) {
      for(j=0; j<selectedAddonProducts[i].shopperInput.length; j++) {
        // If a shopperInput is not entered then no need to send this further
        if(selectedAddonProducts[i].shopperInput[j].value ||
(selectedAddonProducts[i].shopperInput[j].required &&
 selectedAddonProducts[i].shopperInput[j].value === false)) {
          shopperInput[selectedAddonProducts[i].shopperInput[j].id] =
 selectedAddonProducts[i].shopperInput[j].value;
        }
      }
    }

    iSelectedSKUsSize = selectedAddonProducts[i].addOnOptions.length - 1;
    for (var j=iSelectedSKUsSize; j>=0; j--) {
      if(!selectedAddonProducts[i].addOnOptions[j].isSelected) {
        selectedAddonProducts[i].addOnOptions.splice(j, 1);
        continue;
      }
      selectedAddonProducts[i].addOnOptions[j].shopperInput = shopperInput;
      selectedAddonProducts[i].addOnOptions[j].quantity = 1;
    }

    // If none of the config options are selected, there is no need
   //  to pass the ConfigProperty
    if(selectedAddonProducts[i].addOnOptions.length == 0) {
      selectedAddonProducts.splice(i, 1);
    }
  }
  return selectedAddonProducts;
},
```

Create the sample Product Details `widget.json` file

The sample's `widget.json` file defines meta-data for the widget and should look something like this:

```
{
  "name": "Sample Product Details",
  "javascript": "product-details",
  "availableToAllPages": true,
  "i18nresources": "sampleProductDetails",
  "imports": [
    "product",
```

```
    "imageRootUrl",
    "loaded",
    "productVariantOptions",
    "productTypes"
  ],
  "config" : {
  }
}
```

Cart Summary widget for add-ons

By default, add-on products that appear in the Cart Summary cannot be edited. This section describes an example based on the Cart Summary widget that is included in Commerce. The sample updates the widget so shoppers can edit or remove add-on products that are already in the cart. Note that the code in this section is for illustrative purposes only; it is not intended to be production-ready, and may not adequately handle all possible use cases or implement the exact behavior you want. In addition, you may need to customize other widgets that handle add-on items.

This sample assumes that you have already created and linked add-on products as described in Create add-on products.

When a shopper clicks the Edit button for an add-on product (`childItem`) associated with a main product (`cartItem`), the click handler opens a modal dialog and passes the selected add-on product ID, and the main product `cartItem productData`.

The JavaScript file for the widget defines a `displayEditAddonModal()` function that implements the logic for the dialog:

```
displayEditAddonModal : function(mainItemProduct,
selectedAddOn, element) {
     var widget = this;
     //Modal related functionality
     $('#CC-addonSelectionpane').on('show.bs.modal', function() {
       widget.selectedAddOnChildItem = selectedAddOn;
       if(widget.addonProductsMap[mainItemProduct.id]) {
         // Add-on data is already present.
         // No need to construct the data
         var tempAddonData =
widget.addonProductsMap[mainItemProduct.id];
         for(var i=0; i<tempAddonData.length; i++) {
           if(tempAddonData[i].repositoryId ==
selectedAddOn.configurablePropertyId) {
             widget.editedAddonData(tempAddonData[i]);
             for(var j=0;
j<widget.editedAddonData().addOnOptions.length; j++) {

if(widget.editedAddonData().addOnOptions[j].repositoryId ==
selectedAddOn.configurationOptionId) {

widget.editedAddonData().addOnOptions[j].isSelected(true);
                 break;
               }
             }
             if(widget.editedAddonData().shopperInput) {
```

```
                for(var j=0; j<widget.editedAddonData().shopperInput.length;
j++) {
                  var shopperInputId =
widget.editedAddonData().shopperInput[j].id();
                  if(selectedAddOn.shopperInput[shopperInputId])
{widget.editedAddonData().shopperInput[j].value
(selectedAddOn.shopperInput[shopperInputId]);
                  }
                }
              }
              widget.addOnPopulated(true);
              break;
            }
          }
        } else {
       widget.getAddOnProductData(mainItemProduct.id, selectedAddOn,
mainItemProduct.addOnProducts);
        }
      });
      $('#CC-addonSelectionpane').modal('show');
      $('#CC-addonSelectionpane').on('hidden.bs.modal', function() {
        widget.addOnPopulated(false);
        widget.editedAddonData(null);
        widget.selectedAddOnChildItem = null;
      });
    },

  <script type='text/html' id='expand-item'>
  <li style="display : inline;">
    <!-- Expanding the childItems -->
    <!-- ko if: !$data.childItems -->
      <!-- ko if: !$data.addOnItem -->
        <div><a data-bind="ccLink: productData, attr:
{ id: 'CC-shoppingCart-configDetails-' + $data.repositoryId}">
<span data-bind="text: displayName"></span></a>
        <!-- ko foreach: $data.selectedOptions -->
          <!-- ko if: $data.optionValue -->
            (<span data-bind="widgetLocaleText :
{value:'option', attr:'innerText', params:
{optionName: $data.optionName,
            optionValue: $data.optionValue}},
            attr: { id: 'CC-shoppingCart-childProductOptions-'+
$parents[0].productId + $parents[0].catRefId  +
($parents[0].commerceItemId ? $parents[0].commerceItemId: '') +
$parents[0].removeSpaces($data.optionValue)}">
            </span>)
          <!-- /ko -->
        <!-- /ko -->
        <span data-bind="currency: { price: $data.externalPrice(),
currencyObj: $widgetViewModel.site().selectedPriceListGroup().currency}">
</span> -x<span data-bind="text: quantity"></span>
        <!-- ko foreach: externalData -->
          <div>
            <small>
              <!-- ko with: values -->
```

```
                <span data-bind="text: $data.label"></span>:
                <span data-bind="text: $data.displayValue"></span>
              <!-- /ko -->
              <!-- ko if: actionCode -->
                (<span data-bind="text: actionCode"></span>)
              <!-- /ko -->
            </small>
          </div>
        <!-- /ko -->
        </div>
      <!-- /ko -->
      <!-- ko if: $data.addOnItem -->
       <!-- ko if: $data.productData -->
        <br>
        <div data-bind="attr: {id: 'CC-shoppingCart-
productAddonItems-' +
$parent.productId + $parent.catRefId + $parent.commerceItemId
+ $index()}">
          <strong>
            <span data-bind="text: $data.productData().displayName"></
span>
            <span>  -  </span>

ko if: ($data.detailedItemPriceInfo) -->
              <span data-
bind="currency:{price:$data.detailedItemPriceInfo()[0]
.detailedUnitPrice,
currencyObj:$parents[3].site().selectedPriceListGroup().currency}">
</span>
               <!-- /ko -->
            <a href="#" data-bind=" click:
$parents[3].handleRemoveAddonFromCart.bind($parents[3], $data) ">
              <img data-bind="widgetLocaleText :
{value:'handleRemoveAddonFromCart', attr:'alt'},
              attr:{id:'CC-shoppingCart-removeAddonItem-' + productId
+ catRefId + (commerceItemId ? commerceItemId: '') }"
src="/img/remove.png" alt="Remove">
            </a>
          </strong>
          <br>
          <!-- ko if: $data.shopperInput -->
            <!-- ko foreach: Object.keys($data.shopperInput) -->
              <span data-bind="text: $data"></span>
              <span>:  </span>
              <span data-bind="text:
$parent.shopperInput[$data]"></span><br>
            <!-- /ko -->
          <!-- /ko -->
          <span data-bind="text: $data.productData().displayName">
</span>
          <span>:  </span>
          <span data-bind="text: $data.catRefId"></span>
          <a href="#" data-bind="
click:$parents[3].displayEditAddonModal.bind($parents[3], $parent,
$data)" tabindex="0" data-toggle="modal">
```

```
            <u><span data-bind="widgetLocaleText:
'editAddonsText'">Edit</span></u>
          </a>
          <br>
        </div>
      <!-- /ko -->
       <!-- /ko -->
    <!-- /ko -->
    <!-- ko if: $data.childItems -->

      <div class = "alignChild"><a data-bind="click:
$widgetViewModel.setExpandedFlag.bind($data, $element),
 attr: { href: '#CC-shoppingCart-configDetails-' +
$data.repositoryId}" data-toggle="collapse"
class="configDetailsLink collapsed"
role="configuration"></a> <a data-bind="ccLink: productData">
<span data-bind="text: displayName"></span></a>
        <!-- ko foreach: $data.selectedOptions -->
                <!-- ko if: $data.optionValue -->
                  (<span data-bind="widgetLocaleText :
{value:'option', attr:'innerText', params: {optionName:
$data.optionName,
                  optionValue: $data.optionValue}},
                  attr: { id: 'CC-shoppingCart-productOptions-'+
$parents[0].repositoryId +
$parents[0].removeSpaces($data.optionValue)}">
                  </span>)
                <!-- /ko -->
        <!-- /ko -->
        <!-- ko ifnot: ($data.expanded) -->
         <span data-bind="if: $data.expanded,currency:
{ price: $data.itemTotal(), currencyObj:
$widgetViewModel.site().selectedPriceListGroup().currency}">
</span> -x<span data-bind="text: quantity"></span>
        <!-- /ko -->
        <!-- ko if: ($data.expanded) -->
         <span data-bind="currency:
{ price: $data.externalPrice(), currencyObj:
$widgetViewModel.site().selectedPriceListGroup().currency}">
</span> -x<span data-bind="text: quantity"></span>
         <!-- /ko -->
        <!-- ko foreach: externalData -->
          <div>
            <small>
              <!-- ko with: values -->
                <span data-bind="text: $data.label"></span>:
                <span data-bind="text: $data.displayValue"></span>
              <!-- /ko -->
              <!-- ko if: actionCode -->
                (<span data-bind="text: actionCode"></span>)
              <!-- /ko -->
            </small>
          </div>
        <!-- /ko -->
        <ul data-bind="template: {name: 'expand-item',
```

```
foreach: $data.childItems}, attr:
{ id: 'CC-shoppingCart-configDetails-' + $data.repositoryId}"
class="collapse">
        </ul>
      </div>
    <!-- /ko -->
  </li>
  </script>
  <!-- /ko -->
<!-- /ko -->
```

The JavaScript file defines a `cancelEditAddon()` function that implements logic for closing the dialog without making changes to the selected add-on product:

```
cancelEditAddon : function() {
                // Modal related functionality
                $('#CC-addonSelectionpane').modal('hide');
```

The JavaScript file defines a `continueEditAddon()` function that implements logic for closing the dialog when the shopper clicks the Save button to save changes to the selected add-on product:

```
continueEditAddon : function() {
        var widget = this;
        // Modal related functionality
        $('#CC-addonSelectionpane').modal('hide');

        var configOptions = widget.editedAddonData().addOnOptions;
        for(var i=0; i<configOptions.length; i++) {
          if(configOptions[i].isSelected()) {
            widget.selectedAddOnChildItem.catRefId =
configOptions[i].sku.repositoryId;
            widget.selectedAddOnChildItem.configurationOptionId =
configOptions[i].repositoryId;
            if(widget.editedAddonData().shopperInput &&
widget.editedAddonData().shopperInput.length > 0) {
              var shopperInput = {};
              for(var j=0;
j<widget.editedAddonData().shopperInput.length; j++) {
         // If a shopperInput is not entered then no need to send this
further
                if(widget.editedAddonData().shopperInput[j].value() ||
(widget.editedAddonData().shopperInput[j].required() &&
widget.editedAddonData().shopperInput[j].value() === false)) {

shopperInput[widget.editedAddonData().shopperInput[j].id()] =
widget.editedAddonData().shopperInput[j].value();
                }
              }
              widget.selectedAddOnChildItem.shopperInput =
shopperInput;
            }
          }
        }
```

```
        console.log(widget.selectedAddOnChildItem);
        // Use cart VM method to update the cart Item data
        widget.cart().editChildItemFromCart(widget.selectedAddOnChildItem);
      },
```

The JavaScript file defines a `validateEditAddon()` function that implements logic for validating the shopper's changes to the add-on product:

```
validateEditAddon : function() {
        var widget = this;
        if(!widget.editedAddonData()) {
          // If the editedAddonData is not yet created,
          // then there is nothing to validate.
          return;
        }

        var addonProduct = widget.editedAddonData();
        // 1. Check if at least one Config Option is selected
        var isConfigOptionSelected = false;
        for(var i=0; i<addonProduct.addOnOptions.length; i++) {
          if(addonProduct.addOnOptions[i].isSelected()) {
            isConfigOptionSelected = true;
            break;
          }
        }
        if(!isConfigOptionSelected) {
          return false;
        }
        // 2. Validate Shopper Input
        if(addonProduct.shopperInput) {
          for(var i=0; i<addonProduct.shopperInput.length; i++) {
            if(!addonProduct.shopperInput[i].validateNow()) {
              return false;
            }
          }
        }
        return true;
```

The JavaScript file defines a `handleRemoveAddonFromCart()` function that implements logic for removing the selected add-on product from the cart:

```
handleRemoveAddonFromCart: function(childCartItem) {
        var widget = this;
        console.log("remove ..");
        widget.cart().removeChildItemFromCart(childCartItem, true);
      },
```

The widget's `display.template` file contains the following code for rendering the dialog:

```
<!-- MODAL dialog for editing or removing an add-on product -->
  <div class="modal fade col-md-12" id="CC-addonSelectionpane"
tabindex="-1" role="dialog">
    <div class="modal-dialog cc-config-modal-dialog">
```

```
      <div class="modal-content">
        <div class="modal-header CC-header-modal-heading">
          <!--  ko if: $parent.addOnPopulated -->
            <h3 data-bind="text:$parent.editedAddonData()
.displayName "></h3>
          <!-- /ko -->
        </div>
        <div class="modal-body cc-modal-body">
          <!--  ko if: $parent.addOnPopulated -->
            <div class="col-md-12">
              <!-- ko with: $parent.editedAddonData().shopperInput -->
                <!-- ko foreach: $data -->
                  <label data-bind="text: $data.label"></label>
                  <!-- ko if: ($data.uiEditorType() == "shortText") -->
                    <input  class="form-control"
type="text" data-bind="validatableValue: $data.value"><br>
                  <!-- /ko -->
                  <!-- ko if: ($data.uiEditorType() == "longText") -->
                    <textarea class="form-control"
data-bind="validatableValue: $data.value"></textarea><br>
                  <!-- /ko -->
                  <!-- ko if: ($data.uiEditorType() == "number") -->
                    <input  class="form-control" type="number"
data-bind="validatableValue: $data.value"><br>
                  <!-- /ko -->
                  <!-- ko if: ($data.uiEditorType() == "date") -->
                    <input  class="form-control" type="date"
data-bind="validatableValue: $data.value"><br>
                  <!-- /ko -->
                  <!-- ko if: ($data.uiEditorType() == "checkbox") -->
                    <input  class="form-control" type="checkbox"
data-bind="checked: $data.value, validatableValue: $data.value"><br>
                  <!-- /ko -->
                  <!-- ko if: ($data.type() == "enumerated") -->
                    <select class="form-control" type="text"
data-bind="options: $data.values,
optionsCaption: $parents[2].listShopperInputPlaceHolderText,
validatableValue: $data.value" ></select><br>
                  <!-- /ko -->
                  <!-- Validation message place holder -->
                  <div>
                    <p class="text-danger" id="CC-shopperInput-error"
                    data-bind="validationMessage:
$data.value" role="alert"></p>
                  </div>
                <!-- /ko -->
              <!-- /ko -->
            </div>

            <br>
            <!-- ko if: ($parent.editedAddonData().addOnOptions.
length > 0) -->
              <!-- ko if:
($parent.editedAddonData().addOnOptions[0].product.type ==
'GiftWrap' || $parent.editedAddonData().addOnOptions[0].product.type ==
```

```
 'Normal') -->
             <div class="col-md-12" style="display: inline-flex;">
               <!-- ko foreach: $parent.editedAddonData().addOnOptions -->
                 <div class="col-md-3">
                        <img style="max-height: 75px;
max-width: 75px;min-height: 75px;min-width: 75px;" data-
bind="productVariantImageSource: {src: $data.product,
imageType: 'thumb', alt:$data.product.displayName,
errorSrc:'/img/no-image.jpg', errorAlt:'No Image Found'},
 click: $parents[1].addOnIconChanged.bind($parents[1],
$parents[1].editedAddonData()) "><br>
           <div style="display:block;word-break:break-all;width:100%;">
                 <span data-bind="text: $data.sku.repositoryId"></span>
                       <span> - </span>
                       <span data-bind="currency: {price:
$data.product.listPrice, currencyObj:
$parents[1].site().selectedPriceListGroup().currency, nullReplace:
$parents[1].priceUnavailableText(), prependNull: false}"></span> 
                            </div>
                   </div>
                 <!-- /ko -->
               </div>
             <!-- /ko -->
             <!-- ko if:
($parent.editedAddonData().addOnOptions[0].product.type == 'Warranty') -->
               <div class="col-md-12">
                 <!-- ko foreach:
$parent.editedAddonData().addOnOptions -->
                   <input type="radio" data-bind="attr:{id:
 $data.repositoryId, name:$parents[1].editedAddonData().repositoryId},
 checked: $parent.selectedAddonSku, value: $data.repositoryId, click:
 $parents[1].addOnRadioChanged.bind($parents[1],
 $parents[1].editedAddonData()) ">
                     <span id="cc-add-on-product-name"
data-bind="text: $data.sku.repositoryId "></span>
                     <span id="cc-add-on-product-price"
data-bind="currency: {price: $data.product.listPrice,
currencyObj: $parents[1].site().selectedPriceListGroup().currency,
 nullReplace: $parents[1].priceUnavailableText(), prependNull:
 false}"></span>
                   </input><br>
                 <!-- /ko -->
               </div>
             <!-- /ko -->
                <div class="col-md-12" class="text-danger" >
               <br>
               <span data-bind="text:
$parent.editedAddonData().stockValidationMessage "></span>
             </div>
           <!-- /ko -->
         <!-- /ko -->
       </div>
       <div class="modal-footer CC-header-modal-footer">
         <button data-bind="click: $parent.cancelEditAddon"
type="button" class="cc-button-secondary">Cancel</button>
```

```
          <button data-bind="enable:
$parent.validateEditAddon.bind($parent)(), click:
$parent.continueEditAddon.bind($parent, $parent.editedAddonData())"
type="button" class="cc-button-primary">Save</button>
        </div>
      </div>
      <!-- /.modal-content -->
    </div>
    <!-- /.modal-dialog -->
  </div>
  <!-- /.modal -->
```

11

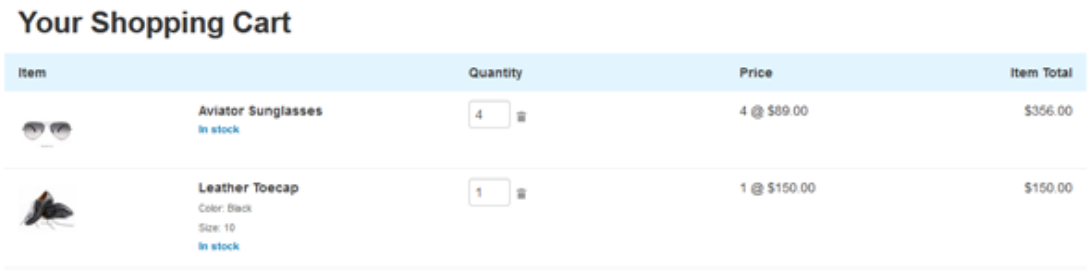
Customize Order Line Items

This section describes how you can enable shoppers to customize items in orders by splitting line items and setting custom properties on the resulting items.

Understand customization of order line items

Orders are broken down into line items that each contain data about an individual SKU being purchased.

For example, if a shopper adds a specific SKU to the shopping cart and specifies a quantity of 4, a single line item is created that stores information about the SKU, the parent product, the SKU price, the quantity (4), and the total price of the 4 SKUs. When you view the shopping cart page, it shows a separate entry for each line item. For example, the following illustration shows two line items, one with a quantity of 4 and the other with a quantity of 1:



If the shopper modifies a quantity value on this page, the corresponding line item is updated to reflect the new quantity and total price.

Line items have a predefined set of properties. In some cases, you may want to store additional data that does not correspond to one of these properties. This is especially useful if you have SKUs that can be customized in some way. For example, if you sell items that can be monogrammed, your site needs a way to store the initials for the monogram.

To enable storing such data, Commerce provides support for adding custom properties to line items. You can use these properties for customization data, such as the initials for a monogram.

If you add custom properties to line items, your storefront needs to provide a way for shoppers to specify the values of these properties. It should also enable shoppers to split a line item whose quantity is greater than 1 into multiple line items, so that each item can be customized individually.

For example, suppose a shopper adds a coffee mug to the cart, and specifies a quantity of 3. Doing this creates a single line item. If the shopper sets a custom monogram property on the line item, the value applies to all three mugs. To specify different values for each mug, he or she will first need to break the single line item whose quantity value is 3 into three separate line items, each with a quantity of 1.

Commerce provides support for splitting line items in this way. The Implement a custom cart summary widget section shows an example of how you can modify your storefront to provide the controls for splitting line items and specifying values of custom properties.

Note that this feature does not support splitting or personalizing line items representing products configured through the integration of Oracle Commerce and Oracle CPQ.

Create custom properties for line items

This section describes how to add custom properties to line items.

This section describes how to use the Oracle Commerce REST web services APIs to add custom properties to line items. See Use the REST APIs for information you need to know before using the services.

View the `commerceItem` item type

Order line items are stored internally as instances of the `commerceItem` item type. You can view this item type with the following call:

```
GET /ccadmin/v1/itemTypes/commerceItem HTTP/1.1
Authorization: Bearer <access_token>
```

The following example shows a portion of the response representing one of the `commerceItem` properties. Each property has a group of attributes whose values control the behavior associated with the property:

```
...
"productId": {
  "length": 254,
  "label": "Product id",
  "type": "shortText",
  "required": false,
  "searchable": false,
  "writable": true,
  "internalOnly": false,
  "uiEditorType": "shortText",
  "default": null,
  "audienceVisibility": null,
  "localizable": false,
  "textSearchable": false,
  "dimension": false,
  "multiSelect": null,
  "editableAttributes": [
    "internalOnly",
    "default",
    "audienceVisibility",
    "textSearchable",
    "label",
    "dimension",
    "required",
    "searchable",
    "multiSelect"
  ]
}
...
```

You can use the `updateItemType` endpoint to modify the `commerceItem` item type:

- Modify existing properties by changing the values of their attributes.
- Create custom properties by specifying their attributes.

See Settable attributes of shopper type properties for descriptions of these attributes.

The next section provides an example of creating a custom property.

Add custom properties to the `commerceItem` item type

You can use the `updateItemType` endpoint in the Commerce Admin API to add custom properties to the `commerceItem` item type. When you add a custom property to the `commerceItem` item type, the property is added to all line items in all orders.

Note that sites that sell configurable products such as telecommunications plans may use certain item types that extend the `commerceItem` item type. When you add custom properties to the `commerceItem` item type, they are automatically added to these extensions as well. For example, you could add a fulfillment status property for separately tracking each line item in a service plan.

The ID of a custom property must include the underscore character (_). This ensures that the ID will not conflict with any properties that Commerce adds to the `commerceItem` item type in the future. The endpoint produces an error if you attempt to create a custom property without an underscore in its ID.

The following example illustrates using the `updateItemType` endpoint to add a custom property. Note that the request header must specify the `x-ccasset-language` value:

```
PUT /ccadmin/v1/itemTypes/commerceItem  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "id": "commerceItem",
  "specifications": [
    {
       "id": "monogram_initials",
       "label": "Initials for monogramming",
       "type": "shortText",
       "uiEditorType": "shortText",
       "internalOnly": false,
       "required": false,
       "default": null
    }
  ]
}
```

The response includes the custom property you added:

```
...
{
      "length": 254,
      "label": "Initials for monogramming",
      "type": "shortText",
      "required": false,
```

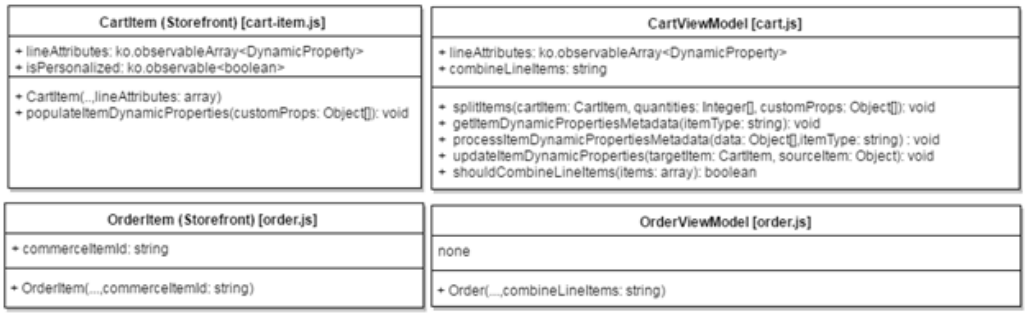
```
        "searchable": false,
        "writable": true,
        "internalOnly": false,
        "uiEditorType": "shortText",
        "default": null,
        "audienceVisibility": null,
        "localizable": false,
        "textSearchable": false,
        "id": "monogram_initials",
        "dimension": false,
        "multiSelect": null,
        "editableAttributes": [
          "internalOnly",
          "default",
          "audienceVisibility",
          "textSearchable",
          "label",
          "dimension",
          "required",
          "searchable",
          "multiSelect"
        ]
      }
...
```

Note that for a `commerceItem` custom property, you will typically want to set `default` to `null` and `required` to `false`, so that the property is not set unless the shopper explicitly chooses to set it.

Understand view model support for line items

This section describes view model support for splitting line items.

The class diagram below shows the specific properties and methods of the view models that support setting custom properties and splitting line items.



The `CartViewModel` implements the `splitItems()` function for splitting existing line items. It also includes functions for handling dynamic properties (custom properties).

The `combineLineItems` string property on the `CartViewModel` determines the behavior when a shopper adds instances of a SKU to a shopping cart that already contains that SKU. If `combineLineItems` is set to `yes` (the default), the SKUs are combined into a single line item. For example, if there is a line item with a quantity of 3 for a certain SKU, and the shopper adds that SKU to the cart again with a quantity of 2, the default

behavior is to modify the existing line item, resulting in a single line item with a quantity of 5.

The `splitItems()` function sets `combineLineItems` to `no`, to prevent merging of line items that have different customizations. In the example above, if `combineLineItems` is set to `no`, the result is two separate line items for the SKU, one with quantity 3 and one with quantity 2.

Implement a custom cart summary widget

After you add custom properties to the `commerceItem` item type, you need to provide a way for a shopper to specify the values of these properties and to split individual line items into multiple line items for customization.

To enable a shopper to specify the values of these properties and to split individual line items into multiple line items for customization, you replace the Cart Summary widget on your shopping cart page with a custom widget that implements these options.

This section describes a custom widget that you could create to add these capabilities to your storefront. It assumes that you have previously created the `monogram_initials` custom property shown in the Add custom properties to the commerceItem item type section. Note that the code in this example is for illustrative purposes only; it is not intended to be production-ready, and may not adequately handle all possible use cases or implement the exact behavior you want. In addition, you may need to customize other widgets that display order data to handle split line items and custom properties.

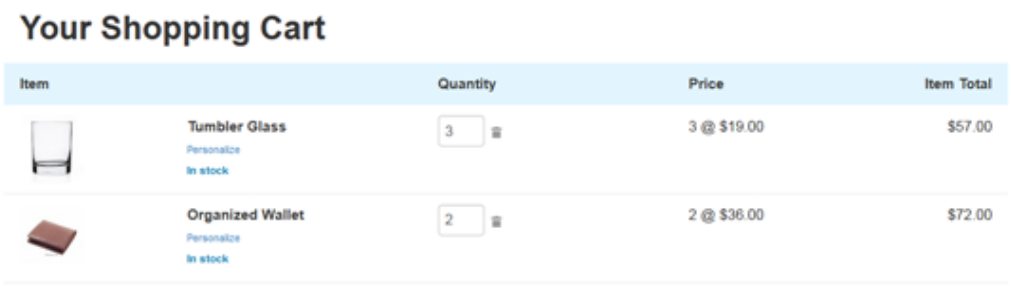
Also, keep in mind that when you create a dynamic property for order line items, the property is added to all line items. You may not want to expose the property in all cases. For example, your store may offer monogramming only for certain items; for other items, you do not want a personalization option to appear. You may need some additional logic in your custom widget to conditionally expose or hide personalization options, depending on the item. For example, you could expose personalization options only for certain custom product types.

Display links for personalization

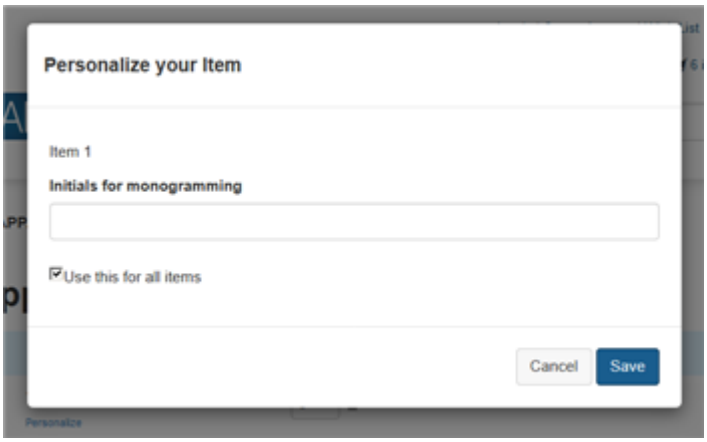
The custom widget's `display.template` file conditionally displays one of two links for each line item:

```
<!-- ko ifnot: $parent.isPersonalized -->
<a data-bind="click: $parents[2].personalizeItem.bind($data, $parent,
  $parents[2])" data-toggle="modal">Personalize</a>
<!-- /ko -->
<!-- ko if: $parent.isPersonalized -->
<a data-bind="click: $parents[2].editItem.bind($data, $parent, $parents[2])"
  data-toggle="modal">Edit</a>
<!-- /ko -->
```

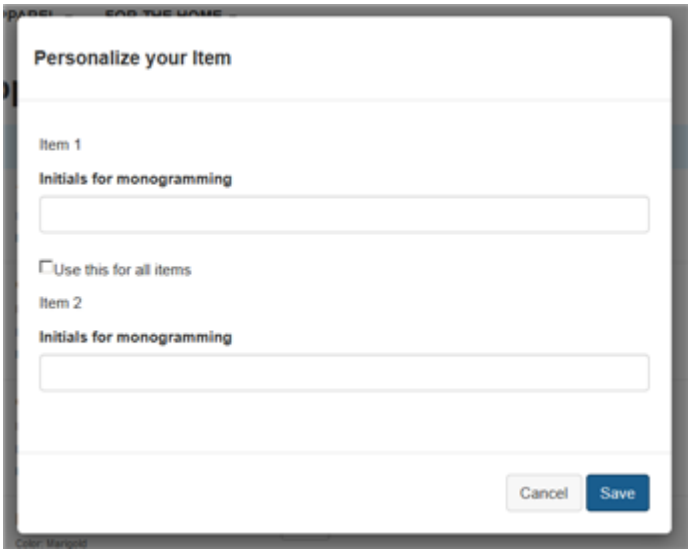
When the cart is initially displayed, none of the line items have been personalized, so the shopping cart page shows a Personalize link for each line item. For example:



Clicking a line item's Personalize link opens a modal dialog for splitting the line item and personalizing the resulting items. For example, if the shopper clicks the Personalize link for the Organized Wallet line item, the following dialog is displayed:



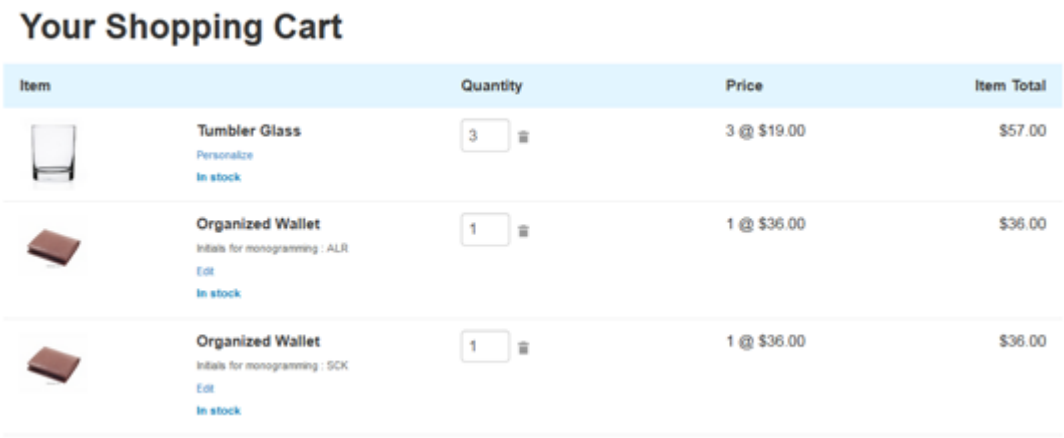
If the checkbox is checked, the line item will not be split when the shopper clicks Save, and the value the shopper supplies for the `monogram_initials` property will be applied to both wallets. If the checkbox is unchecked, the line item will be split, and the dialog expands to display fields for specifying the custom property values for each item individually:



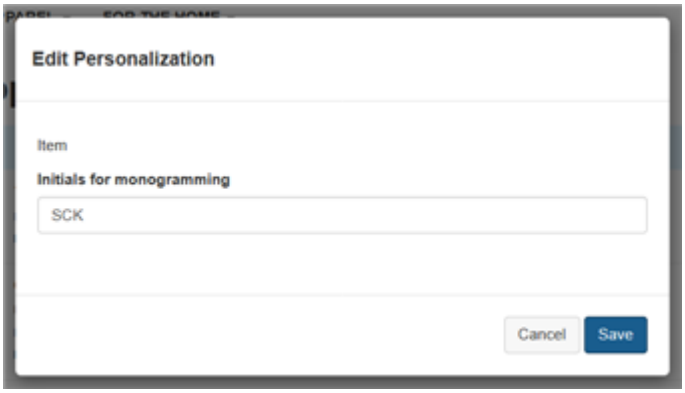
After the shopper fills in the monogram values and clicks Save, the Organized Wallet line item is split into two line items, and the `monogram_initials` property is set

separately on each one. The widget's `display.template` file displays the value of the property for each item it is set on:

```
<!-- ko if:($parents[1][$data.id()]) -->
  <span data-bind = "text: $data.label"></span> : <span data-bind = "text:
  $parents[1][$data.id()]"></span><br>
<!-- /ko -->
```



Notice that there are now two line items for the Organized Wallet, each with a quantity of 1, and each with a different value for the custom property. The Tumbler Glass line item still has a Personalize link, but the Organized Wallet line items now have Edit links instead. Clicking one of the Edit links opens a dialog for changing the monogram for the wallet associated with that link. For example:



Create the dialog for splitting and personalizing line items

The JavaScript file for the widget defines a `personalizeItem()` function that implements the logic for the dialog:

```
personalizeItem: function(item, widget) {
     //Personalizing the item
     var totalQuantity = item.quantity();
     if(widget.cart().lineAttributes().length > 0) {
       for(var i=0; i< totalQuantity; i++) {
         var propObj = {};
         for(var j=0; j< widget.cart().lineAttributes().length;j++) {
           //Injecting default values of properties from the metadata
```

```
            propObj[widget.cart().lineAttributes()[j].id()] =
            ko.observable(widget.cart().lineAttributes()[j].value());
          }
          //Pushing each key-value pair to the result object to show
onto the modal
          widget.itemProps.push(propObj);
        }
      }
      //Modal related functionality
      $('#cc-personalizationPane').on('show.bs.modal', function() {
          widget.item(item);
      });
      $('#cc-personalizationPane').modal('show');
      $('#cc-personalizationPane').on('hidden.bs.modal', function() {
          widget.itemProps([]);
      });
 },
```

If the custom properties have default values, these values are used to populate the dialog fields. However, providing defaults for these values is not recommended, because they will be applied to all line items, including ones that cannot actually be personalized.

The widget's `display.template` file contains the following for rendering the dialog:

```
<!-- Personalization Modal -->
<div class="modal fade" id="cc-personalizationPane" tabindex="-1"
role="dialog">
  <div class="modal-dialog cc-modal-dialog">
    <div class="modal-content">
    <!-- ko if: $parent && $parent.item()!=null -->
      <div class="modal-header CC-header-modal-heading">
        <h4>Personalize your Item</h4>
      </div>
      <div class="modal-body cc-modal-body">
      <h5>Item 1</h5>
      <!-- ko with: lineAttributes -->
      <!-- ko foreach: $data -->
        <label  class="control-label" data-bind="text: label"></label>
        <!-- ko if: $parents[2].itemProps()[0] -->
        <!-- ko if: uiEditorType() == "shortText" || uiEditorType() ==
"richText"
          || uiEditorType() == "number" || uiEditorType() == "date" -->
        <input  class="form-control" type="text" data-bind="attr:
{name : id},
          value: $parents[2].itemProps()[0][id()]"><br>
        <!-- /ko -->
        <!-- ko if: uiEditorType() == "checkbox" -->
        <input  class="form-control" type="checkbox" data-bind="attr:
{name : id},
          checked: $parents[2].itemProps()[0][id()]"><br>
        <!-- /ko -->
        <!-- /ko -->
      <!-- /ko -->
      <!-- /ko -->
```

```
      <input type="checkbox" data-bind="checked: $parent.noRepeat">Use this
        for all items</input>
      <div data-bind="visible: !$parent.noRepeat()">
      <!-- ko foreach: new Array($parent.item().quantity()-1) -->
      <h5><p>Item <span data-bind="text: $index()+2" /></p></h5>
      <!-- ko with: $parent.lineAttributes -->
      <!-- ko foreach: $data -->
        <label class="control-label" data-bind="text: label"></label>
        <!-- ko if: $parents[3].itemProps()[$parentContext.$index()+1] -->
        <!-- ko if: uiEditorType() == "shortText" || uiEditorType() ==
"richText"
          || uiEditorType() == "number" || uiEditorType() == "date" -->
        <input class="form-control" type="text" data-bind="attr: {name : id},
          value: $parents[3].itemProps()[$parentContext.$index()+1][id()]"/
><br>
        <!-- /ko -->
        <!-- ko if: uiEditorType() == "checkbox" -->
        <input class="form-control" type="checkbox" data-bind="attr: {name :
id},
          checked: $parents[3].itemProps()[$parentContext.$index()+1][id()]"/
><br>
        <!-- /ko -->
       <!-- /ko -->
       <!-- /ko -->
      <!-- /ko -->
      <!-- /ko -->
      </div>
      </div>
      <div class="modal-footer CC-header-modal-footer">
        <button data-
bind="click: $parent.cancelPersonalization.bind($parent)"
           type="button" class="cc-button-secondary">Cancel</button>
         <button data-bind="click: $parent.savePersonalization.bind($parent)"
          type="button" class="cc-button-primary">Save</button>
    </div>
    <!-- /ko -->
    </div>
  </div>
</div>
```

The JavaScript file for the widget also includes a `savePersonalization()` function, which is executed when the shopper clicks Save:

```
savePersonalization: function() {
    var widget= this;
    //Saving personalized values
    if(widget.noRepeat()) {
      //If the flag is checked, populate the entire quantity with the same
set
      //of values.
      widget.item().populateItemDynamicProperties(widget.itemProps()[0]);
      widget.item().isPersonalized(true);
      widget.cart().markDirty();
    } else {
```

```
      //Splitting all quantities to 1 each if the flag is unchecked.
      //This can be customized further to split total quantity in any
manner.
        var quantityList = new Array(widget.item().quantity()
+1).join(1).
          split('').map(function(){return 1;})
      //Calling split items function to create multiple lines with
      //different custom properties provided.
        widget.cart().splitItems(widget.item(), quantityList,
        widget.itemProps());
      }
      //Modal related functionality
       $('#cc-personalizationPane').modal('hide');
},
```

If the shopper chooses to split a line item, the widget splits it into line items whose quantity is 1. For example, if the line item has a quantity of 3, it is split into three line items with a quantity of 1. After an item is split, the shopper can increase the quantity of one of the resulting items and then split that item. If the shopper splits an item and then adds more of the same SKU to the shopping cart, the addition is treated as a separate line item and not combined with the split items.

Note that the `splitItems()` function of the `CartViewModel` supports splitting in other ways than the above code implements. For example, `splitItems()` can split a line item with quantity 3 into two line items, one with a quantity of 1 and one with a quantity of 2. You can support this option in your own custom widget by creating controls that enable shoppers to specify different splitting options.

When the customer edits property values, the sample widget triggers one pricing call per edit. You can reduce the number of pricing calls by implementing a way for your custom widget to trigger pricing only after all personalization is complete.

Create the dialog for modifying personalized line items

The `isPersonalized` boolean on the `CartItem` is used to indicate whether a line item has been personalized. By default it is set to `false`; when a shopper clicks a Personalize link on a line item to invoke the widget's `personalizeItem()` function, the widget sets the `isPersonalized` property to `true`. This causes the Edit link to be displayed for the resulting line items. Clicking the Edit link invokes the widget's `updatePersonalization()` function, which enables further changes to the custom property values, but not further splitting of the line items:

```
updatePersonalization: function(){
    var widget = this;
    //Calling the method to update properties of the item specified
    //by the user in the modal
    widget.item().populateItemDynamicProperties(widget.itemProps()[0]);
    $('#cc-editPane').modal('hide');
    widget.cart().markDirty();
},
```

You could extend this function to support further splitting of line items as well.

The widget's `display.template` file contains the following for rendering the dialog:

```
<!-- Edit Personalization Modal -->
 <div class="modal fade" id="cc-editPane" tabindex="-1" role="dialog">
   <div class="modal-dialog cc-modal-dialog">
     <div class="modal-content">
     <!-- ko if: $parent && $parent.item()!=null -->
       <div class="modal-header CC-header-modal-heading">
         <h4>Edit Personalization</h4>
       </div>
       <div class="modal-body cc-modal-body">
       <h5>Item</h5>
       <!-- ko with: lineAttributes -->
       <!-- ko foreach: $data -->
         <label  class="control-label" data-bind="text: label"></label>
         <!-- ko if:  $parents[2].itemProps()[0] -->
         <!-- ko if: uiEditorType() == "shortText" || uiEditorType() ==
          "richText" || uiEditorType() == "number" || uiEditorType() ==
"date" -->
         <input  class="form-control" type="text" data-bind="attr: {name :
id},
           value: $parents[2].itemProps()[0][id()]"><br>
          <!-- /ko -->
          <!-- ko if: uiEditorType() == "checkbox" -->
          <input  class="form-control" type="checkbox" data-bind="attr:
{name :
            id}, checked: $parents[2].itemProps()[0][id()]"><br>
          <!-- /ko -->
         <!-- /ko -->
       <!-- /ko -->
       <!-- /ko -->
       </div>
       <div class="modal-footer CC-header-modal-footer">
         <button data-bind="click: $parent.cancelEdit.bind($parent)"
type="button"
           class="cc-button-secondary">Cancel</button>
         <button data-
bind="click: $parent.updatePersonalization.bind($parent)"
           type="button" class="cc-button-primary">Save</button>
     </div>
     <!-- /ko -->
     </div>
   </div>
 </div>
```

12

Ship an Order to Multiple Addresses

The split shipping feature makes it possible for a shopper to split a single order so that portions of it are shipped to different addresses.

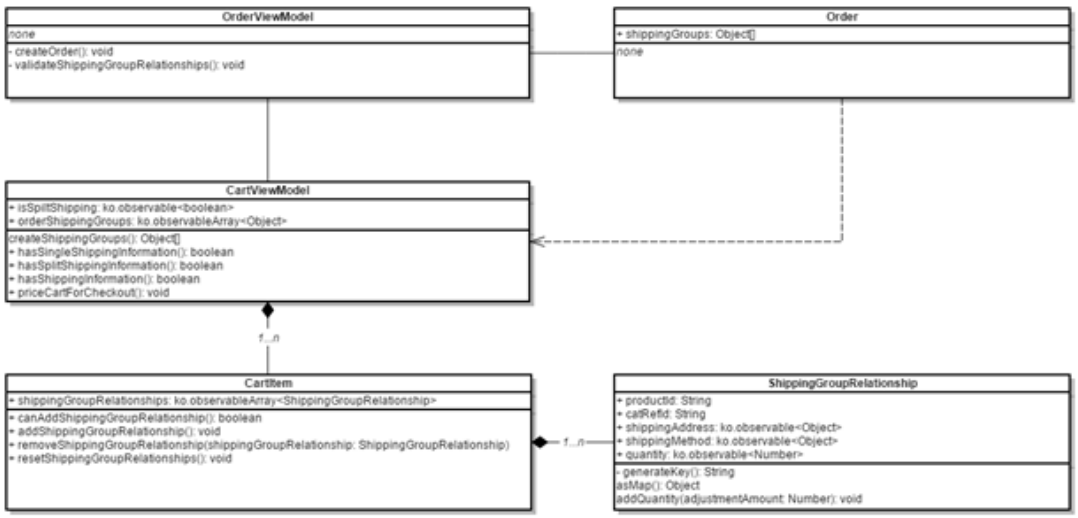
This section provides details on how to implement this feature.

Understand view model support for split shipping

This section provides information on the view models that contain data related to split shipping and the APIs you use to interact with them.

Affected view models

The class diagram below shows the properties and methods added to the view models to support multiple shipping groups. Detailed descriptions of these APIs follow the diagram.



`CartViewModel.isSplitShipping`

The property that indicates if split shipping is activated. When the shopper chooses the split shipping option, this property must be set to true. You should also use this property to control the visibility of split shipping/single shipping UI elements.

`ShippingGroupRelationship`

The view model class that represents an association between a cart item and a shipping group. Strictly speaking, shipping group relationships associate a specified quantity of a cart item with a shipping address and shipping method (not a shipping group). However, the shipping groups array that supports split shipping is directly generated from the `ShippingGroupRelationship` instances. When the shopper selects a shipping address and shipping method for a quantity of a given cart item, it is this class that captures those selections. See Understand REST support for split shipping for more details on the shipping groups array.

CartItem.shippingGroupRelationships

The collection of `ShippingGroupRelationship` instances for a cart item. By default, there is one `ShippingGroupRelationship` instance per cart item, meaning that each cart item will be associated with at least one shipping group.

CartItem.addShippingGroupRelationship

In order to ship the same cart item (SKU) to several different addresses (shipping groups), it is necessary to create several associations (shipping group relationships) between a cart item and the different shipping groups. This method creates additional shipping group relationship instances, allowing multiple associations per single cart item. The maximum number of shipping group relationship instances is equal to the cart item quantity, beyond which it is not possible to split the cart item any further (as there would be more associations than cart items available).

CartItem.canAddShippingGroupRelationship

Determines if it is possible to add another shipping group relationship instance (that is, associate the cart item with another shipping group). The maximum number of shipping group relationship instances is equal to the cart item quantity, beyond which it is not possible to split the cart item any further (as there would be more associations than cart items available).

CartItem.removeShippingGroupRelationship

Removes a `ShippingGroupRelationship` instance from the cart item's `shippingGroupRelationships` array.

CartViewModel.hasSingleShippingInformation

When in single shipping mode (that is, `isSplitShipping` is false), determines if the single shipping address and shipping method are populated.

CartViewModel.hasSplitShippingInformation

When in split shipping mode (that is, `isSplitShipping` is true), determines if all shipping group relationships are populated with shipping addresses and shipping methods, and the `shippingGroupRelationships` array is valid. See ShippingGroupsRelationships array validation for details.

CartViewModel.hasShippingInformation

This property is true if either `hasSplitShippingInformation` or `hasSingleShippingInformation` is true, otherwise it is false.

CartViewModel.priceCartForCheckout

The method that calls the Store `priceOrder` endpoint to price the shopping cart only if the cart has shipping information; that is, `hasShippingInformation` is true. This method's internal logic accounts for both single and split shipping scenarios.

CartViewModel.orderShippingGroups

The latest shipping groups array (if any) returned from a web service call. See Understand REST support for split shipping for more details on this array

Implement split shipping UI controls

It is standard e-commerce practice that shipping selections are implemented as part of the checkout flow.

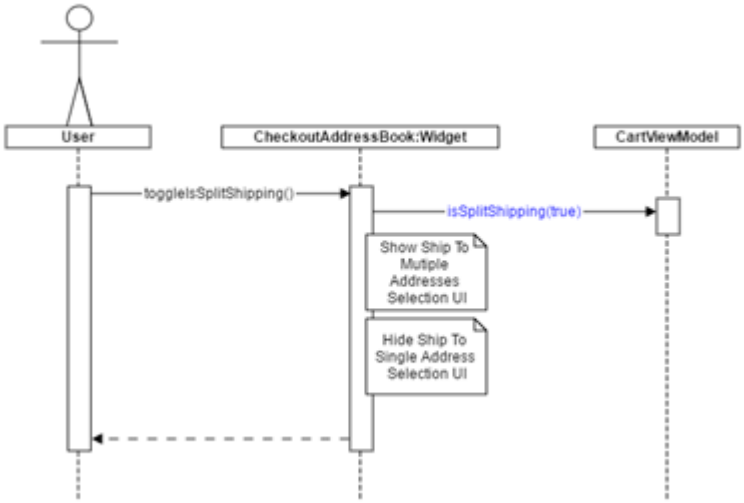
Commerce already implements single shipping this way, so for the purposes of continuity, it is recommended that split shipping is also implemented in checkout. A non-standard implementation (such as on the cart page), although possible, would require more custom coding and may have additional side effects that require mitigation.

Split shipping toggle

A split shipping toggle button allows users to activate or deactivate split shipping for the current order. The button toggles the state of the `isSplitShipping` property. Also, it may be necessary to toggle the visibility of split shipping/single shipping UI elements.

The following sequence diagrams shows how you might choose to activate and deactivate the split shipping toggle on your storefront.

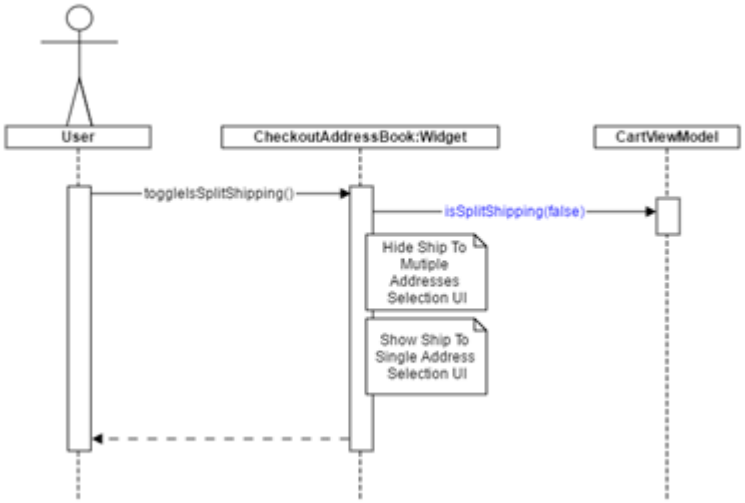
Activate



In this diagram, the following happens:

- The shopper clicks a Use Split Shipping button on the UI, which calls the Checkout Address Book widget's `toggleSplitShipping()` method.
- The `toggleSplitShipping()` method sets the `CartViewModel.isSplitShipping` property to true.
- The Checkout Address Book widget shows the Ship To Multiple Addresses UI and hides the Ship To Single Address UI.

Deactivate

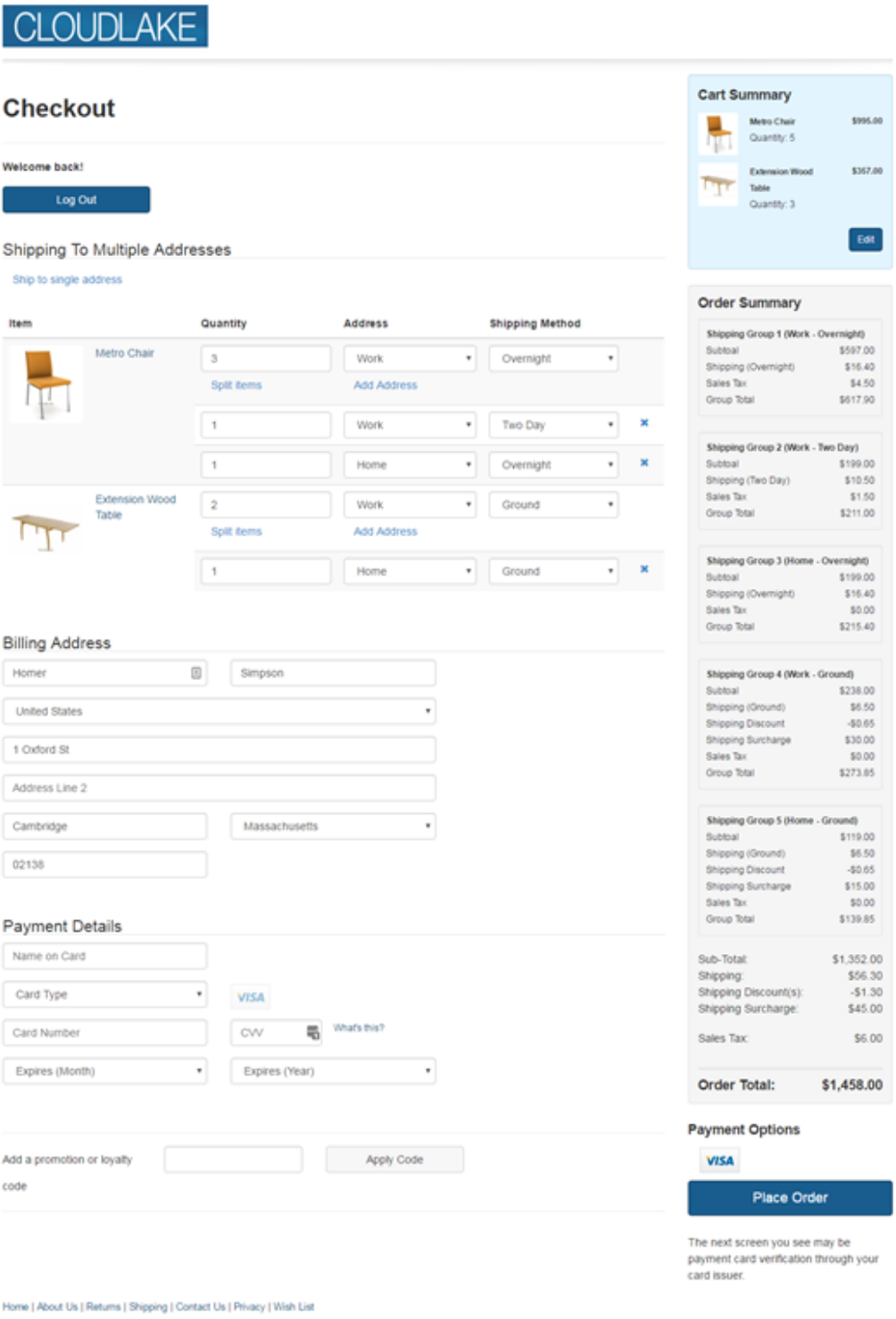


In this diagram, the following happens:

- The shopper clicks the Use Single Shipping button on the UI, which calls the Checkout Address Book widget's `toggleSplitShipping()` method.
- The `toggleSplitShipping()` method sets the `CartViewModel.isSplitShipping` property to false.
- The Checkout Address Book widget shows the Ship To Single Address UI and hides the Ship To Multiple Addresses UI.

Split shipping web form

A split shipping web form allows users to populate shipping options for each cart item. The `shippingGroupRelationships` property (which is an observable array) captures the split shipping options for each cart item. Each instance of a `ShippingGroupRelationships` object associates a quantity of cart item with a given shipping address and shipping method. A single cart item can have several `ShippingGroupRelationships` instances, allowing the cart item to be split across several shipping groups.



Quantity field

The Quantity field allows the shopper to specify the portion of cart item to be associated with a given shipping group. The following code shows a sample UI binding pattern for this feature:

```
<!-- ko foreach: cart().items -->
  <!-- ko foreach: shippingGroupRelationships -->
```

```
    <input type="number" name="quantity" class="form-control" data-
bind="
      value: quantity,
      event: {change: $parents[1].priceSplitShippingCartForCheckout}">
  <!-- /ko -->
<!-- /ko -->
```

Note: A change of Quantity will cause the widget to make a pricing call, provided the split shipping form is complete and valid.

Shipping Address field

The Shipping Address field allows the shopper to select a shipping address for the specified quantity of the cart item. The following code shows a sample UI binding pattern for this feature:

```
<!-- ko foreach: cart().items -->
  <!-- ko foreach: shippingGroupRelationships -->
    <select
      class="form-control"
      name="shippingAddress"
      data-bind="options: $parents[1].user().shippingAddressBook(),
                 optionsText: $parents[1].getOptionTextForAddress,
                 value: shippingAddress,
                 optionsCaption: 'Select shipping address',
                 event: {change: $parents[1].lookupShippingOptions}">
    </select>
  <!-- /ko -->
<!-- /ko -->
```

Shipping Address options should be retrieved from the profile's shipping address book so that updates to the shipping address book will automatically be reflected in the options list. A change of Shipping Address must trigger a method in your widget that makes an AJAX service call to retrieve the valid shipping options for the selected address and product. The product ID must be passed in this call because some products may have a shipping surcharge and not all shipping methods can be used for products with surcharges.

Shipping Method field

The Shipping Method field allows the shopper to select a shipping method for the specified quantity of the cart item. The following code shows a sample UI binding pattern for this feature:

```
<!-- ko foreach: cart().items -->
  <!-- ko foreach: shippingGroupRelationships -->
    <select
      class="form-control"
       name="shippingMethod"
       data-bind="options: shippingOptions,
                  optionsText: 'displayName',
                  value: shippingMethod,
                  optionsCaption: 'Select shipping method',
                  enable: shippingAddress,
                  event:
```

```
{change: $parents[1].priceSplitShippingCartForCheckout}">
    </select>
  <!-- /ko -->
<!-- /ko -->
```

The Shipping Method options displayed to the shopper must be populated by a method in your widget that makes an AJAX service call to retrieve the valid shipping options for the selected address. A change of Shipping Address should trigger this method.

A change of Shipping Method should cause the widget to make a pricing call, provided the split shipping form is complete and valid.

Split Items button

The Split Items button creates another shipping group relationship instance for this cart item, allowing the same cart item to be associated with more than one shipping group. The following code shows a sample UI binding pattern for this feature:

```
<!-- ko foreach: cart().items -->
  <!-- ko if: $parent.canAddShippingGroupRelationship($parent) -->
    <button class="btn btn-link" data-bind="click:
addShippingGroupRelationship">
      Split items
    </button>
  <!-- /ko -->
<!-- /ko -->
```

It is only possible to split a cart item if the cart item quantity is greater than `shippingGroupRelartionships.length`.

Remove Item (X) button

The Remove Item button, shown as an X in the sample UI displayed in this section, removes a shipping group relationship instance. The following code shows a sample UI binding pattern for this feature:

```
<!-- ko foreach: cart().items -->
  <!-- ko foreach: shippingGroupRelationships -->
    <button class="btn btn-sm btn-link" data-bind="click:
       parents[1].removeShippingGroupRelationship.bind($parent)">
      <span class="glyphicon glyphicon-remove"></span>
    </button>
  <!-- /ko -->
<!-- /ko -->
```

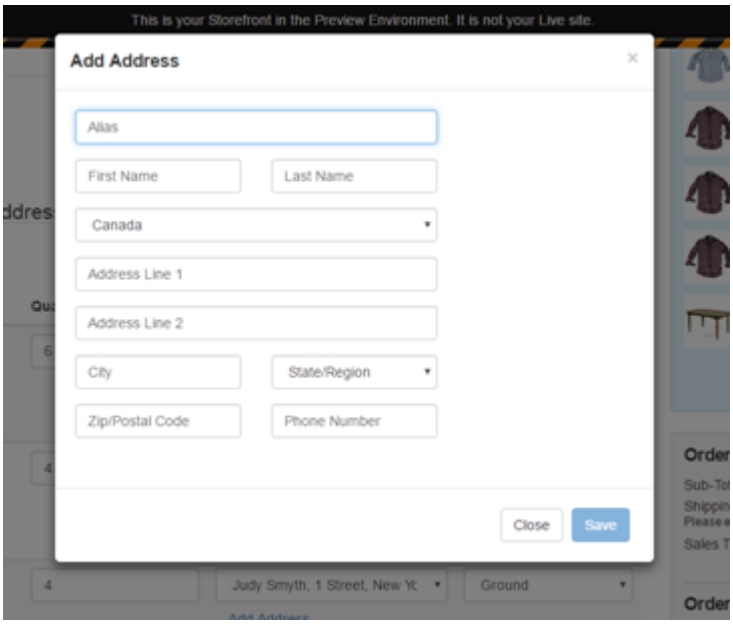
The `removeShippingGroupRelationship` method, used in the click binding above, is a widget method and not the `CartItem.removeShippingGroupRelationship` method. It does, however, delegate to `CartItem.removeShippingGroupRelationship`, and also makes a pricing call, provided the split shipping form is complete and valid.

Add Address button

The Add Address button opens an address form where a shopper can save a new address to his profile address book. Once created, the new address will automatically appear in the Shipping Address options in the split shipping form. Apart from the inclusion of an alias field,

no new address management APIs are required for split shipping. Re-using existing address management functionality is wholly sufficient to for this purpose.

The following illustration shows what an Add Address form might look like with fields for name, address, and phone number information. Note the addition of the Alias field.

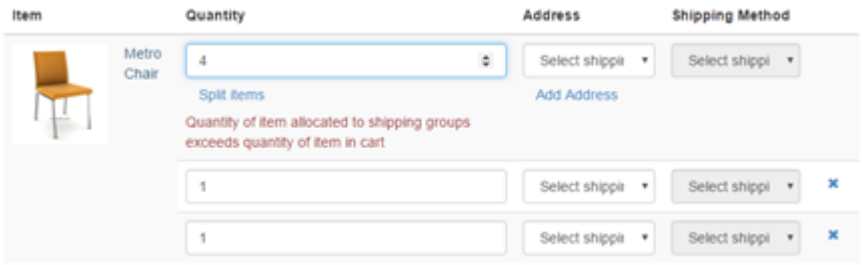


`ShippingGroupsRelationships` array validation

The `shippingGroupRelationships` property has two predefined custom Knockout validators:

- Quantity of item allocated to shipping groups exceeds quantity of item in cart: Checks that the sum of the shipping group quantities is not greater than the cart item quantity.
- Cart item quantity not fully allocated to shipping groups: Checks that the sum of the shipping group quantities is not less than the cart item quantity.

The above validators are computed automatically. The illustration below shows an error message that indicates to the shopper when a validation has failed.



To output the error message on screen, use the `validationMessage` binding shown below:

```
<!-- ko foreach: cart().items -->
          <div class="text-danger" data-bind="validationMessage:
           shippingGroupRelationships" role="alert"></div> <!-- /ko -->
```

Price order

The pricing method `CartItem.priceCartForCheckout` handles both single and split shipping pricing. There is no change to the API for the split shipping.

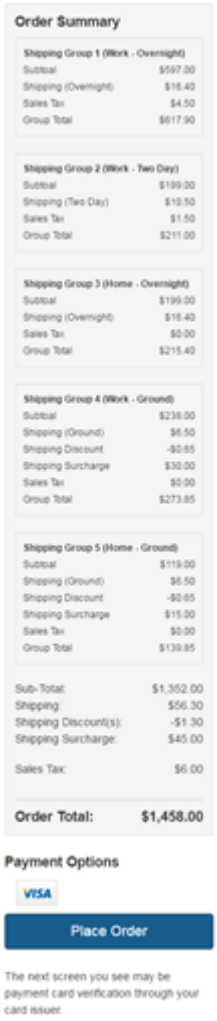
As you create your widgets, you should consider when pricing is called. For example, you should call pricing when:

- The Quantity field changes.
- The Shipping Method field changes.
- A shipping group relationship is removed (clicking the X button in the sample UI shown in this section).

The `priceCartForCheckout` method uses the `isSplitShipping` property to determine which pricing request to make. The `priceCartForCheckout` method will only make a pricing request if the split shipping form is complete and valid.

Order summary

Your storefront may need to show a pricing breakdown by shipping group in an Order Summary section, as shown in the following example which displays the shipping group name, subtotal before tax and shipping, shipping costs, sales tax, and total cost for each group.



The following binding pattern outputs the price info per shipping group in the widget shown above.

```
<!-- ko if: cart().isSplitShipping() -->
          <!-- ko foreach: cart().orderShippingGroups -->
            <!-- ko if: $data.hasOwnProperty("priceInfo") -->
              <div class="well well-sm small">         <strong>
                  Shipping Group
                  <span data-bind="text: ($index() + 1)"></span>
                  (<span data-bind="text: shippingAddress.alias"></
span> -
                   <span data-bind="text:
                       shippingMethod.shippingMethodDescription"></
span>)
                </strong>         <div class="row">
                <div class="col-xs-7">Subtoal</div>
                <div class="col-xs-5 text-right">
                  <span data-bind="currency: {
                     price: priceInfo.subTotal,
                     currencyObj:
                          $parent.site().selectedPriceListGroup().cur
rency}">
```

```
                    </span>          </div>
                </div>         <div class="row">
                  <div class="col-xs-7">
                    Shipping (<span data-bind="text:
                        shippingMethod.shippingMethodDescription"></span>)
                  </div>          <div class="col-xs-5 text-right">
                    <span data-bind="currency: {
                       price: priceInfo.shipping,
                       currencyObj:
                            $parent.site().selectedPriceListGroup().currency}
">
                    </span>          </div>
                </div>
                <!-- ko if: $data.hasOwnProperty("discountInfo") -->
                  <!-- ko if: discountInfo.shippingDiscount !== 0 -->
                    <div class="row">
                      <div class="col-xs-7">Shipping Discount </div>
                      <div class="col-xs-5 text-right">
                        <span data-bind="currency: {
                            price: -discountInfo.shippingDiscount,
                            currencyObj:
                                $parent.site().selectedPriceListGroup().curre
ncy}">
                        </span>              </div>
                    </div>          <!-- /ko -->
                <!-- /ko -->
                <!-- ko if: priceInfo.shippingSurchargeValue &&
                     priceInfo.shippingSurchargeValue !== 0 -->
                  <div class="row">
                    <div class="col-xs-7">Shipping Surcharge</div>
                    <div class="col-xs-5 text-right">
                      <span data-bind="currency: {
                         price: priceInfo.shippingSurchargeValue,
                         currencyObj:
                            $parent.site().selectedPriceListGroup().currency}
">
                      </span>            </div>
                  </div>        <!-- /ko -->
                <!-- ko if: $parent.cart().showTaxSummary -->
                  <div class="row">
                    <div class="col-xs-7">Sales Tax</div>
                    <div class="col-xs-5 text-right">
                      <span data-bind="currency: {
                         price: priceInfo.tax,
                         currencyObj:
                              $parent.site().selectedPriceListGroup().currenc
y}">
                      </span>            </div>
                  </div>        <!-- /ko -->
                <!-- ko if: (taxPriceInfo.isTaxIncluded &&
                             $parent.cart().showTaxSummary) -->
                  <div class="row">
                    <div class="col-xs-7">Group Total (excluding tax)</div>
                    <div class="col-xs-5 text-right">
                      <span data-bind="currency: {
```

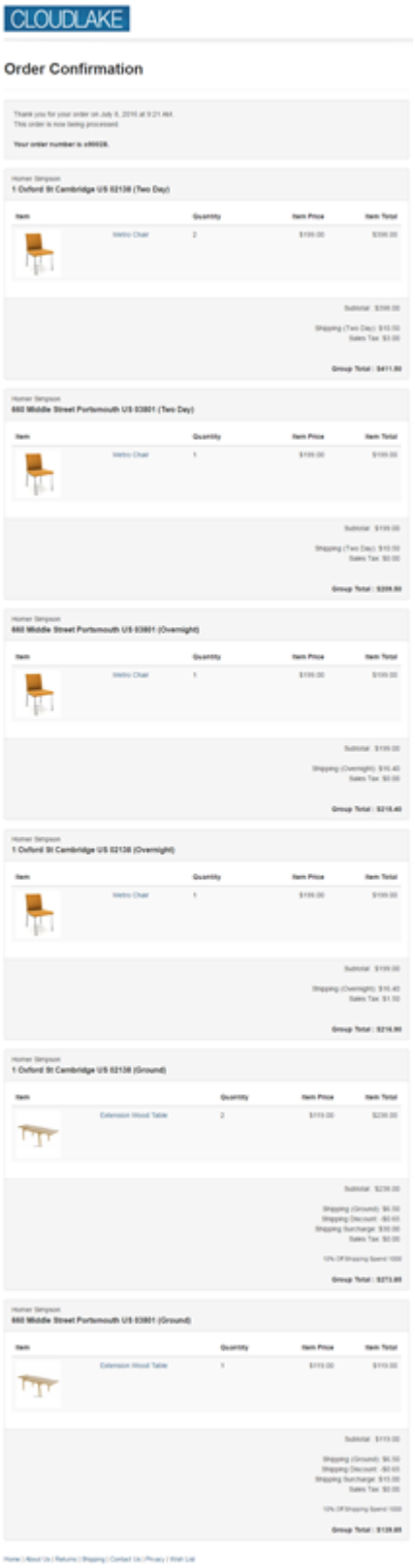
```
                        price: priceInfo.totalWithoutTax,

currencyObj: $parent.site().priceListGroup.currency}">
                      </span>             </div>
                  </div>         <!-- /ko -->
                <div class="row">           <div class="col-xs-7">
                    Group Total
                    <!-- ko if: (taxPriceInfo.isTaxIncluded &&
                                 $parent.cart().showTaxSummary) -->
                      <span data-bind="widgetLocaleText:
'includingTaxText'"></span>
                    <!-- /ko -->           </div>
                  <div class="col-xs-5 text-right">
                    <span data-bind="currency: {
                       price: priceInfo.total,
                       currencyObj:
                            $parent.site().selectedPriceListGroup().cur
rency}">
                    </span>           </div>         </div>       </
div>     <!-- /ko -->   <!-- /ko -->
        <!-- /ko -->
```

Place order

The `OrderViewModel.handlePlaceOrder` method handles placing both single and split shipping orders. There is no change to the API for split shipping. The `handlePlaceOrder` method should be called when the Place Order button is clicked.

Order confirmation

Order confirmation should display each shipping group and its relevant information such as the addressee, the shipping method, the items in the shipping group, the subtotal before tax and shipping, shipping costs, sales tax, and a total cost for each group.

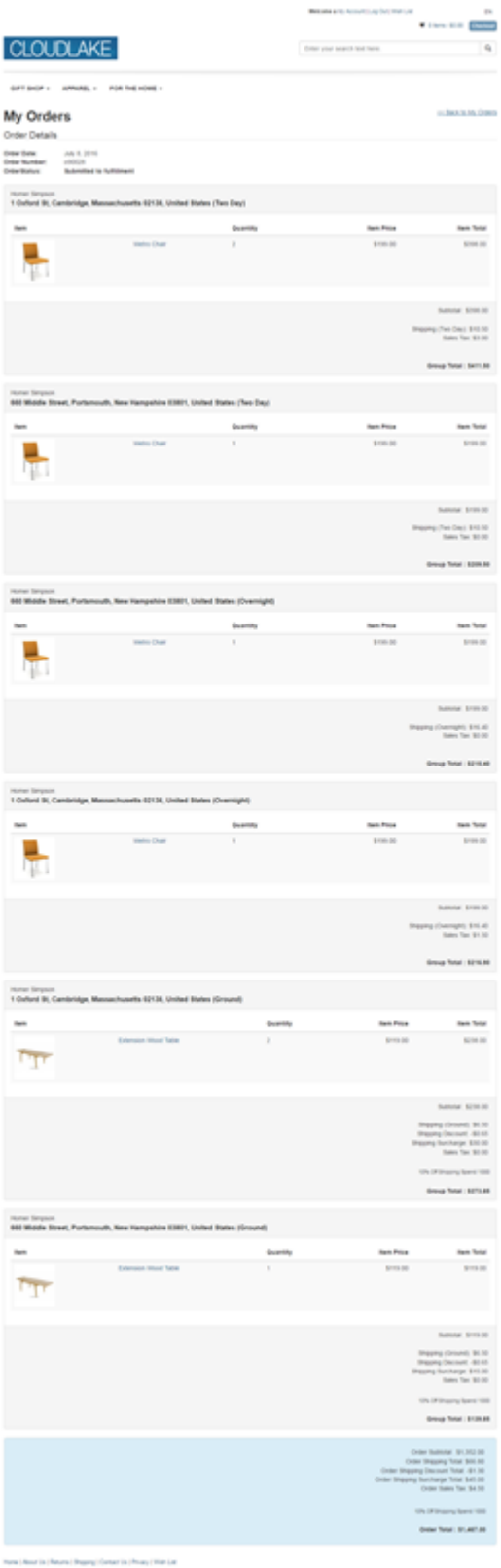

CLOUDLAKE
Order Confirmation

The following binding pattern iterates over the `shippingGroups` array in the widget shown above.

```
<!-- ko with: confirmation -->   <!-- ko foreach: shippingGroups -->
            Mark-up for shipping group here...   <!-- /ko -->
        <!-- /ko -->
```

Order details

Order details, like order confirmation, should display each shipping group and its relevant information such as the addressee, the shipping method, the items in the shipping group, the subtotal before tax and shipping, shipping costs, sales tax, and a total cost for each group.


CLOUDLAKE
My Orders
Order Details

The following binding pattern iterates over the `shippingGroups` array in the widget shown above.

```
<!-- ko with: orderDetails-->
   <!-- ko foreach: shippingGroups -->
   Mark-up for shipping group here...
   <!-- /ko -->
<!-- /ko -->
```

Understand REST support for split shipping

To support the split shipping feature, updates have been made to the Profile and Orders resources in the Store REST API.

These resources contain endpoints for managing shopper profiles and orders, respectively.

- For the Profile resources, support has been added for an alias property to identify addresses in the address book.
- For the Orders endpoints, support has been added for multiple shipping groups, in the form of a `shippingGroups` array, and for an alias property to identify the addresses contained in those shipping groups. Each of the Orders endpoints that take detailed order or cart information in their request payload support the `shippingGroups` array. All of the Orders endpoints that return detailed order information will return a `shippingGroups` array in the response as long as there are cart items associated with the shipping groups. If none of the shipping groups for a cart or order have cart items associated with them then the `shippingGroups` array is suppressed in the endpoint response. This will typically be the case for persisted carts (incomplete orders) that existed in the order repository before the split shipping feature was deployed.

Understand the `shippingGroups` array

The `shippingGroups` array is supported in request and response payloads when appropriate. In request payloads, it is optional. In response payloads, it is returned as long as item relationships exist for one or more shipping groups in the cart/order. Each object in the `shippingGroups` array holds information about:

- The shipping address (`shippingAddress` object) for the shipping group. This `shippingAddress` object has an alias property that identifies the address.
- The shipping method (`shippingMethod` object) for the shipping group.
- The item relationships (items array) for the shipping group.
- The unique ID of the shipping group (`shippingGroupId`), if known.

Existing `shippingAddress` and `shippingMethod` properties

Prior to the introduction of the split shipping feature, the Store API endpoints only supported a single shipping group. All items in the cart inherently belonged to that shipping group though the relationship between shipping group and cart items was not established in the order repository until the order was submitted. To that end, the pricing and order-related endpoints supported specification of the shipping address and shipping method for the default shipping group, via the `shippingAddress` and

`shippingMethod` properties, in the request and response payloads. For backwards compatibility these properties continue to be supported in the request and response payloads with the caveats described below.

In request payloads, the `shippingAddress` and `shippingMethod` properties are:

- Ignored if the `shippingGroups` array is specified.
- Used to set the shipping address and method of the first shipping group in the `shippingGroups` array when the `shippingGroups` array is not explicitly specified.

In response payloads, the `shippingAddress` and `shippingMethod` properties refer to the shipping address and shipping method of the first shipping group in the `shippingGroups` array.

Customize email templates for split shipping

If you configure your store to support split shipping, you should also customize the templates for emails that contain order summaries so those emails can display the appropriate shipping addresses.

The data available to the email templates to support the split shipping feature comes from the Orders resource in the Store REST API. Support has been added for multiple shipping groups, in the form of a `shippingGroups` array, and for an alias property to identify the addresses contained in those shipping groups. For more information, see Understand REST support for split shipping.

Note: Before you customize the email templates, read Configure Email Settings. For details about working with FreeMarker templates, see the Apache FreeMarker documentation at `freemarker.org`.

To display multiple shipping addresses in an email template:

1. Download the email template as described in Customize email templates.
2. Update the `html_body.ftl` file. See Sample email template for sections of a sample `html_body.ftl` file that supports split shipping.
3. Upload the updated template as described in Customize email templates.

Sample email template

The following sample shows split shipping customizations you might make to the `html_body.ftl` file for the Order Placed email template. Order Placed emails let customers know that an order has been submitted for fulfillment. The `html_body.ftl` file is the `FreeMarker` template file that configures the HTML body of the email.

IMPORTANT: This sample code is not production ready and should be used for informational purposes only. It has not been tested for accessibility, internationalization, or unexpected path flows.

The default `html_body.ftl` file for the Order Placed email displays the shipping address, shipping method, payment methods, and a cart summary for an order. This sample uses the macro directive to allow the email body to display multiple shipping addresses, along with their associated shipping methods and cart summaries. (A `FreeMarker` macro is a template fragment associated with a variable.)

The following macro is used to display the appropriate shipping method.

```
<!-- ko if: cart().isSplitShipping() -->
          <!-- ko foreach: cart().orderShippingGroups -->
            <!-- ko if: $data.hasOwnProperty("priceInfo") -->
              <div class="well well-sm small">         <strong>
                  Shipping Group
                  <span data-bind="text: ($index() + 1)"></span>
                  (<span data-bind="text: shippingAddress.alias"></
span> -
                   <span data-bind="text:
                       shippingMethod.shippingMethodDescription"></
span>)
                </strong>         <div class="row">
                  <div class="col-xs-7">Subtoal</div>
                  <div class="col-xs-5 text-right">
                    <span data-bind="currency: {
                       price: priceInfo.subTotal,
                       currencyObj:
                            $parent.site().selectedPriceListGroup().cur
rency}">
                    </span>           </div>
                </div>         <div class="row">
                  <div class="col-xs-7">
                    Shipping (<span data-bind="text:
                        shippingMethod.shippingMethodDescription">></
span>)
                  </div>           <div class="col-xs-5 text-right">
                    <span data-bind="currency: {
                       price: priceInfo.shipping,
                       currencyObj:
                            $parent.site().selectedPriceListGroup().cur
rency}">
                    </span>           </div>
                </div>
                <!-- ko if: $data.hasOwnProperty("discountInfo") -->
                  <!-- ko if: discountInfo.shippingDiscount !== 0 -->
                    <div class="row">
                      <div class="col-xs-7">Shipping Discount </div>
                      <div class="col-xs-5 text-right">
                        <span data-bind="currency: {
                            price: -discountInfo.shippingDiscount,
                            currencyObj:
                                $parent.site().selectedPriceListGroup()
.currency}">
                          </span>               </div>
                    </div>           <!-- /ko -->
                <!-- /ko -->
                <!-- ko if: priceInfo.shippingSurchargeValue &&
                     priceInfo.shippingSurchargeValue !== 0 -->
                  <div class="row">
                    <div class="col-xs-7">Shipping Surcharge</div>
                    <div class="col-xs-5 text-right">
                      <span data-bind="currency: {
                         price: priceInfo.shippingSurchargeValue,
```

```
                        currencyObj:
                           $parent.site().selectedPriceListGroup().currency}
">
                      </span>             </div>
                  </div>         <!-- /ko -->
                <!-- ko if: $parent.cart().showTaxSummary -->
                  <div class="row">
                    <div class="col-xs-7">Sales Tax</div>
                    <div class="col-xs-5 text-right">
                      <span data-bind="currency: {
                         price: priceInfo.tax,
                         currencyObj:
                              $parent.site().selectedPriceListGroup().currenc
y}">
                      </span>             </div>
                  </div>         <!-- /ko -->
                <!-- ko if: (taxPriceInfo.isTaxIncluded &&
                             $parent.cart().showTaxSummary) -->
                  <div class="row">
                    <div class="col-xs-7">Group Total (excluding tax)</div>
                    <div class="col-xs-5 text-right">
                      <span data-bind="currency: {
                         price: priceInfo.totalWithoutTax,

currencyObj: $parent.site().priceListGroup.currency}">
                      </span>             </div>
                  </div>         <!-- /ko -->
                <div class="row">           <div class="col-xs-7">
                    Group Total
                    <!-- ko if: (taxPriceInfo.isTaxIncluded &&
                                 $parent.cart().showTaxSummary) -->
                      <span data-bind="widgetLocaleText:
'includingTaxText'"></span>
                    <!-- /ko -->           </div>
                  <div class="col-xs-5 text-right">
                    <span data-bind="currency: {
                       price: priceInfo.total,
                       currencyObj:
                            $parent.site().selectedPriceListGroup().currency}
">
                    </span>           </div>         </div>       </div>
<!-- /ko -->   <!-- /ko -->
        <!-- /ko -->
```

The following macro is used to display the cart summary for each shipping group.

```
<#macro displayShippingItems shippingItems>

 <table width="100%" align="center" border="0" cellpadding="0"
cellspacing="0"
 class="devicewidthinner">
    <tbody>
      <tr>
        <td width="30%"
```

```
          style="font-family: Helvetica, arial, sans-serif;
          font-size: 18px; color: #FFFFFF; text-align: left;
          line-height: 24px; background: #1c73a3;
          padding: 5px 10px 5px 10px;"
          st-title="3col-title1">
          ${getString("ORDER_PLACED_ITEM_TITLE")}
        </td>
        <td width="40%"
          style="font-family: Helvetica, arial, sans-serif; font-size:
18px;
          color: #ffffff; text-align: center; line-height: 24px;
          background: #1c73a3; padding: 5px 10px 5px 10px;"
          st-title="3col-title1"> </td>
        <td width="10%"
          style="font-family: Helvetica, arial, sans-serif; font-size:
18px;
          color: #ffffff; text-align: center; line-height: 24px;
          background: #1c73a3; padding: 5px 10px 5px 10px;"
          st-title="3col-title1">
          ${getString("ORDER_PLACED_QUANTITY_TITLE")}
        </td>
        <td width="20%"
          style="font-family: Helvetica, arial, sans-serif; font-size:
18px;
          color: #ffffff; text-align: right; line-height: 24px;
          background: #1c73a3; padding: 5px 10px 5px 10px;"
          st-title="3col-title1">
          ${getString("ORDER_PLACED_PRICE_TITLE")}
        </td>
      </tr>
      <#list shippingItems as product>

      <tr>
        <td
          style="font-family: Helvetica, arial, sans-serif; font-size:
14px;
          color: #687078; text-align: left; line-height: 24px;
          padding: 5px 10px 5px 10px;"
          st-content="3col-content1" width="30%">
          <img src="${product.imageLocation}"
          alt="${product.title!}" width="50%">
        </td>
        <td
          style="font-family: Helvetica, arial, sans-serif; font-size:
14px;
          color: #687078; text-align: left; line-height: 24px;
          padding: 5px 10px 5px 10px;"
          st-content="3col-content1" width="40%">
          <a href="${product.location}">${product.title!}</a>
          <!-- Variants -->
          <#if product.variants??>
            <br />
            <#list product.variants as variant>
              ${variant.optionName}: <#if variant.optionValue??>
                                     ${variant.optionValue}</#if>
```

```
              <br />
            </#list>
          </#if>
        </td>
        <td
          style="font-family: Helvetica, arial, sans-serif; font-size: 14px;
          color: #687078; text-align: center; line-height: 24px;
          padding: 5px 10px 5px 10px;"
          st-content="3col-content1" width="10%">
          ${product.quantity}</td>
        <td
          style="font-family: Helvetica, arial, sans-serif; font-size: 14px;
          color: #687078; text-align: right; line-height: 24px;
          padding: 5px 10px 5px 10px;"
          st-content="3col-content1" width="20%">
          ${product.price}</td>
      </tr>

      </#list>
    </tbody>
  </table>

</#macro>
```

The following macro is used to display a shipping group's shipping method.

```
<#macro displayShippingMethodMacro shippingMethod >
          <table width="186" align="right" border="0"
            cellpadding="0" cellspacing="0" class="devicewidth">
            <tbody>       <tr>         <td>
                  <!-- start of text content table -->
                  <table width="186" align="center" border="0"
                    cellpadding="0" cellspacing="0"
                    class="devicewidthinner">             <tbody>
                      <!-- title -->               <tr>
                        <td
                          style="font-family: Helvetica, arial, sans-serif;
                          font-size: 18px; color: #666666; text-align:
center;
                          line-height: 24px;"
                          st-title="3col-title3">
                          ${getString("ORDER_PLACED_SHIPPING_METHODS_TITLE")}
                        </td>               </tr>
                      <!-- end of title -->
                      <!-- Spacing -->               <tr>
                        <td width="100%" height="15"
                          style="font-size: 1px; line-height: 1px;
                          mso-line-height-rule: exactly;"> </td>
                      </tr>              <!-- Spacing -->
                      <!-- content -->               <tr>
                        <td
                          style="font-family: Helvetica, arial, sans-serif;
                          font-size: 14px; color: #687078; text-align:
center;
```

```
                        line-height: 24px;"                    st-
content="3col-content3">                    $
{shippingMethod!}<br />                 </td>
</tr>              <!-- end of content -->            </
tbody>           </table>
                </td>       </tr>
              <!-- end of text content table -->     </tbody>
          </table> </#macro>
```

The following macros are used to display a single shipping group or multiple shipping groups.

```
<#macro displayShippingGroupMacro shippingGroup>     <!-- 3 Start of
Columns -->
          <table width="100%" bgcolor="#ffffff" cellpadding="0"
cellspacing="0"
            border="0" id="backgroundTable">     <tbody>       <tr>
                <td>           <@displayShippingAddressMacro
                       shippingAddress=shippingGroup.shippingAddress />
                </td>         <td>
                  <@displayShippingMethodMacro
                       shippingMethod=shippingGroup.shippingMethod />
                </td>       </tr>       <tr>
                <td colspan="2">
                  <@displayShippingItems
shippingItems=shippingGroup.shippingItems />
                </td>       </tr>     </tbody>   </table> </#macro>
<#macro displayMultipleShippingGroupsMacro shippingAddresses >
          <div>Calling details </div>
          <#list shippingAddresses as currShippingGroup>
            <@displayShippingGroupMacro
shippingGroup=currShippingGroup />   </#list> </#macro>
```

The following code calls `displayMultipleShippingGroupsMacro` to display the order details for multiple shipping groups if the order includes them.

```
<div> Multi shipping methods</div>
<#if data.shippingGroups??>
  <@displayMultipleShippingGroupsMacro
      shippingAddresses=data.shippingGroups />
</#if>
```

Retaining shipping group information

Commerce's default behavior is to reset shipping group relationships whenever the shopper makes changes to the cart.

You may want to override this behavior to retain shipping group information for the shopper so he does not have to re-enter it when he resumes the checkout process.

To avoid resetting shipping group relationships, you can use an application-level JavaScript module that configures the `resetShippingGroupRelationships` flag. Specifically, you must create an extension that uploads an application-level JavaScript

module that depends on the `cc-store-configuration-1.0.js` library and sets the `resetShippingGroupRelationships` flag to false. The following code sample shows what the contents of this JavaScript module might look like (for general information on creating an application-level JavaScript extension, see Understand widgets):

```
define(
  //-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
  ['ccStoreConfiguration'],
  //-------------------------------------------------------------------
  // Module definition
  //-------------------------------------------------------------------
  function(CCStoreConfiguration) {

    "use strict";

    return {
      // Override the default value (true) of
resetShippingGroupRelationships to
      // false to retain shipping group relationship on cart updates
      CCStoreConfiguration.getInstance().resetShippingGroupRelationships =
false;
    };
});
```

Extending the `CartItem` and `ShippingGroupRelationship` view models

The `CartItem` and `ShippingGroupRelationship` view models are publicly available and you can extend them to behave in whatever way your storefront requirements demand.

To override the methods belonging to these view models, you must create an extension that uploads an application-level JavaScript module that depends on the view models. The following code samples show what the contents of these JavaScript modules might look like. The first sample shows how to extend the `canAddShippingGroupRelationship` method in the `CartItem` view model.

```
define(
  //-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
  ['viewModels/cart-item'],
  //-------------------------------------------------------------------
  // Module definition
  //-------------------------------------------------------------------
  function(CartItem) {

    "use strict";

    return {
      onLoad: function() {
        CartItem.prototype.canAddShippingGroupRelationship = function () {
          // Override code goes here
```

```
        };
      }
    };
});
```

This sample shows how to extend the `addQuantity` method in the `ShippingGroupRelationship` view model.

```
define(
  //-------------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------------
  ['viewModels/shipping-group-relationship'],
  //-------------------------------------------------------------------------
  // Module definition
  //-------------------------------------------------------------------------
  function(ShippingGroupRelationship) {

    "use strict";

    return {
      onLoad: function() {
        ShippingGroupRelationship.prototype.addQuantity = function (x)
{
          // Override code goes here
        };
      }
    };
});
```

Note: For general information on creating an application-level JavaScript extension, see Understand widgets.

13

Exclude Items from Shipping Methods and Costs

This section describes how to use the Commerce Admin APIs to exclude certain items from the calculation of shipping costs or from being shipped by certain methods.

- You can exclude item from a specific shipping method. For example, you might want to restrict certain products, such as oversized items or furniture, to ground shipping only. When a shopper adds a product from an excluded category to his or her cart, restricted shipping options are unavailable.
- You can exclude items from the order total for the purpose of calculating shipping costs. For example gift wrapping, from the order total for the purposes of calculating shipping costs.

Exclude items from shipping methods

Commerce allows you to exclude collections of items from a specific shipping method.

For example, you might want to restrict certain products, such as oversized items, hazardous items, or furniture, to ground shipping only. When a shopper adds one of these items to his or her cart, the restricted shipping options are not displayed and so cannot be selected by the shopper.

If you have implemented the split shipping feature, only portions of the order that contain products that are excluded from certain shipping methods will not show those shipping methods. If other portions of the order do not include products that are excluded from shipping methods, all appropriate shipping methods are displayed. See Ship an Order to Multiple Addresses for more information about split shipping.

To exclude items from a shipping method, perform the following tasks:

1. Create one or more non-navigable collections and add the products you want to exclude from shipping methods to them. See Create collections for the excluded items for more information.
2. Update the shipping methods from which you want to exclude items with one or more of the collections you created in the previous step. See Update shipping methods for more information.

Exclude items from shipping cost calculations

Commerce allows you to exclude certain items, for example gift wrapping or electronic downloads, from the order total for the purposes of calculating shipping costs.

For example, if an order contains a DVD that costs $25 and three electronic gift cards that cost $50 each, the order total used for shipping cost calculation is $25, not $175.
To exclude items from shipping cost calculations, perform the following tasks:

1. Create one or more non-navigable collections and add the items you want to exclude from shipping methods to them. See Create collections for the excluded items for more information.
2. Update the shipping methods from which you want to exclude items with one or more of the collections you created in the previous step. See Update shipping methods for more information.

Create collections for the excluded items

Commerce excludes collections of products, not individual products, from shipping methods and shipping calculations.

Create one or more non-navigable collections for the products you want to exclude from shipping methods or shipping calculations. See Organize products in collections to learn about working with collections in the Commerce administration interface.

Keep the following points in mind when you create collections for excluded items:

- Collections you use for shipping method or shipping cost exclusion should be non-navigable, meaning shoppers cannot browse to the collections as part of the hierarchical catalog on your store.
- It is a good idea to create a separate collection for each type of product. For example, you might create one collection for oversized items and another for hazardous items. Similarly, you could create one collection for all products you want to exclude from shipping charges calculation, but organize different types of products, like gift wrap or electronic gift cards, in child collections.
- When you use a collection for shipping method or shipping cost exclusion, all the products in all its child collections are also excluded.

Update shipping methods

Once you create and populate non-navigable collections for products to exclude, configure the exclusion features through the `createShippingMethod` and `updateShippingMethod` endpoints, which contain the following array properties:

- `shippingMethodExclusions` identifies the collections you want to exclude from the shipping method. All products in the specified collections and any child collections are excluded.
- `shippingChargeExclusions` identifies the collections you want to exclude from shipping cost calculations. All products in the specified collections and any child collections are excluded from shipping cost calculations.

The following sample request updates the Ground shipping method to exclude products in two collections (gift wrap and electronic gift cards) from shipping cost calculations.

```
PUT /ccadmin/v1/shippingMethods/standardShippingMethod HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
  "shippingChargeExclusions": {
    "excludedCategoriesShippingCharge": [
```

```
      "catGW",
      "catDownloads"
    ]
  }
}
```

The following sample request updates the Next Day shipping method to exclude products in two collections (furniture and oversized).

```
PUT /ccadmin/v1/shippingMethods/overnightShippingMethod HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
  "shippingMethodExclusions": {
    "excludedCategories": [
      "catFurniture",
      "catOversized"    ]
 }
}
```

Update the Order Summary – Checkout widget

To implement shipping method exclusions, make sure your checkout layouts include the latest version of the Order Summary – Checkout widget.

To replace a widget with the latest version, see Upgrade deployed widgets in Customize your store layouts.

14

Manage Countries and Regions for Shipping and Billing Addresses

By default, Oracle configures your Commerce instance to include the countries and regions that you require for addresses such as shipping and billing addresses.

You can, however, use the Admin API to add, delete, and update countries and regions available to your store.

Understand countries and regions

Commerce uses countries and their associated regions in a number of places on your storefront, in the administration interface, and in the REST APIs.

For example, a shopper's shipping-address country specifies the country where their purchases are shipped. Countries and regions you add with the Admin API `/ccadmin/v1/countries/:` endpoints are available in all areas of your Commerce instance.

You can use Commerce settings and widgets to narrow the list of countries used by . For example:

- You can create shipping methods that allow shoppers to have their purchases shipped only to the contiguous 48 states, even when all 50 states and Armed Forces PO Boxes are available in your Commerce instance
- You can narrow the available list of billing countries for a country store so that only shoppers whose billing addresses are in specific countries can purchase items from the store.

If you Commerce instance supports multiple sites, the list of countries is available to all sites. See Configure Sites for more information about multiple sites.

Each country and region you add to Commerce must match a valid ISO 3166 code. For details about ISO 3166, codes, visit the International Standards for Organization website at https://www.iso.org.

The following table describes the properties that are part of requests and responses to the `/ccadmin/v1/countries/:` endpoints.

Property	Description
countryCode	A valid ISO 3166 country code. For example, AR is the country code for Argentina.
regionCode	A valid ISO 3166 subdivision code. For example, AR-B is the code for Buenos Aires.
abbreviation	The second part of a regionCode. For example, abbreviation for Buenos Aires is B
displayName	A string that identifies a country or region on the storefront and in the administration interface.

Retrieve a list of countries and regions

To see the countries that are currently part of your Commerce instance, issue a GET request to the `/ccadmin/v1/countries` endpoint.

The following example shows a sample response body for this request:

```
[
  {
    "repositoryId": "AR",
    "countryCode": "AR",
    "displayName": "Argentina"
  },
  {
    "repositoryId": "AU",
    "countryCode": "AU",
    "displayName": "Australia"
  },
  {
    "repositoryId": "BD",
    "countryCode": "BD",
    "displayName": "Bangladesh"
  },
  {
    "repositoryId": "BR",
    "countryCode": "BR",
    "displayName": "Brazil"
  },
  {
    "repositoryId": "CA",
    "countryCode": "CA",
    "displayName": "Canada"
  },
]
```

To see a country's regions that are currently part of your Commerce instance, issue a `GET` request to the `/ccadmin/v1/countries/{id}` endpoint. For example, to see all the Canadian provinces in your Commerce instance, issue a `GET` request to the `/ccadmin/v1/countries/{CA}` endpoint. The following example shows a sample response body for this request. Notice that the response does not return all the Canadian provinces,, only the ones added to the Commerce instance being queried.

```
{
  "regions": [
    {
      "regionCode": "CA-AB",
      "displayName": "Alberta",
      "repositoryId": "CA-AB",
      "abbreviation": "AB"
    },
    {
      "regionCode": "CA-BC",
      "displayName": "British Columbia",
```

```
      "repositoryId": "CA-BC",
      "abbreviation": "BC"
    },
    {
      "regionCode": "CA-MB",
      "displayName": "Manitoba",
      "repositoryId": "CA-MB",
      "abbreviation": "MB"
    },
    {
      "regionCode": "CA-PE",
      "displayName": "Prince Edward Island",
      "repositoryId": "CA-PE",
      "abbreviation": "PE"
    },
    {
      "regionCode": "CA-QC",
      "displayName": "Quebec",
      "repositoryId": "CA-QC",
      "abbreviation": "QC"
    },
    {
      "regionCode": "CA-SK",
      "displayName": "Saskatchewan",
      "repositoryId": "CA-SK",
      "abbreviation": "SK"
    },
  ],
  "countryCode": "CA",
  "displayName": "Canada",
  "repositoryId": "CA",
  "links": [
    {
      "rel": "self",
      "href": "http://servername:9080/ccadmin/v1/countries/CA"
    }
  ]
}
```


Create and update countries and regions

To add a country to Commerce, issue a `POST` request to the `/ccadmin/v1/countries/addCountries` endpoint.

You can also create the country's regions in the same request.

The following example shows a sample request body for creating two countries (India and Sri Lanka), each with two regions.


```
{
    "countries": [{
            "countryCode": "IN",
            "displayName": "India",
```

```
            "regions": [{
                    "regionCode": "IN-KA",
                    "displayName": "Karnataka",
                    "abbreviation": "KA"
                },
                {
                    "regionCode": "IN-SK",
                    "displayName": "Sikkim",
                    "abbreviation": "SK"
                }
            ]
        },
        {
            "countryCode": "LK",
            "displayName": "SriLanka",
            "regions": [{
                    "regionCode": "LK-GA",
                    "displayName": "Galle",
                    "abbreviation": "GA"
                },
                {
                    "regionCode": "LK-CO",
                    "displayName": "Colombo",
                    "abbreviation": "CO"
                }
            ]
        }
    ]
}
```

The following example shows the response body returned:

```
[
  {
    "repositoryId": "IN",
    "regions": [
      {
        "regionCode": "IN-KA",
        "displayName": "Karnataka",
        "repositoryId": "IN-KA",
        "abbreviation": "KA"
      },
      {
        "regionCode": "IN-SK",
        "displayName": "Sikkim",
        "repositoryId": "IN-SK",
        "abbreviation": "SK"
      }
    ],
    "countryCode": "IN",
    "displayName": "India"
  },
  {
    "repositoryId": "LK",
```

```
    "regions": [
      {
        "regionCode": "LK-CO",
        "displayName": "Colombo",
        "repositoryId": "LK-CO",
        "abbreviation": "CO"
      },
      {
        "regionCode": "LK-GA",
        "displayName": "Galle",
        "repositoryId": "LK-GA",
        "abbreviation": "GA"
      }
    ],
    "countryCode": "LK",
    "displayName": "SriLanka"
  }
]
```

To add regions to an existing country, issue a `PUT` request to the `/ccadmin/v1/countries/{id}/addRegions` endpoint.

The following example shows a sample request body for creating two new regions in an existing country.

```
{
    "countryCode": "IN",
    "displayName": "India",
    "regions": [
       {
            "regionCode": "IN-TG",
            "displayName": "Telangana",
            "abbreviation": "TG"
        }
    ]
}
```

The following example shows the response body returned:

```
{
    "countryCode": "IN",
    "displayName": "India",
    "regions": [
       {
            "regionCode": "IN-TG",
            "displayName": "Telangana",
            "abbreviation": "TG"
        }
    ]
}
```

Delete countries and regions

When you delete a country or region, it is no longer available to merchants or shoppers.

Deleting a country automatically deletes all its associated regions.

Before you can delete a country or region, make sure you have removed it from all lists of billing or shipping countries and regions. When you issue the DELETE request, Commerce checks that the countries or regions in the body of the request are not currently referenced in billing or shipping lists. If they are, the request returns an error and the items are not deleted.

To remove a country from your Commerce instance, issue a DELETE request to the `ccadmin/v1/countries/deleteCountries` endpoint.

The following sample request body removes three countries and all their associated regions from a Commerce instance:

```
{
    "ids": ["CN", "PK", "LK"]
        }
```

To remove a region from a country, issue a DELETE request to the `/ccadmin/v1/countries/{id}/deleteRegions` endpoint, where id is the repositoryId of the country whose regions you want to delete.

For example, to delete three Canadian provinces, issue a DELETE request to the `/ccadmin/v1/countries/{CA}/deleteRegions` endpoint. The following example shows a sample body for this request:

```
{
    "ids": ["CA-AB", "CA-SK", "CA-PE"]
 }
```

Customize address formats using the API

You can make modifications to address formats for countries that require additional address customizations.

In addition to using Oracle Commerce Cloud's default address formats, you can use the REST API to create multi-country custom address formats. This allows you to create country-specific address formats or ensure that your address formats align with the requirements of any external service that you might use. Addresses that appear in profiles, accounts, registration requests, payments and order addresses can be customized.

You can also use the REST API to create address types, which you use to identify an address' purpose.

Understand customized address formats

Some countries require a county or a district in addition to the default address, city, state, postal code and phone number fields. Creating custom address formats allows

you to create address formats that change based upon the shopper's country locale. Using the API, you can map custom fields to the administration interface. This also allows you to work with address verification services and ensure that your address customizations are in line with these services.

Address customizations can contain fields with variable label text. These fields can have different requirements and default values. Once you have created the custom address, you associate it with a specific country or countries so that it is used only on storefronts (and administration and agent interfaces) with that locale. When a shopper from that locale sees the site, the address format is changed to match the format identified for that locale.

You can customize addresses that are found on profiles, accounts, account registration requests, as well as payment and shipping group pages. Addresses found on credit card, inventory location or the Ship from Warehouse location address in the tax processing setting pages cannot be customized.

Custom address formats are defined by creating a server-side extension (SSE). The server-side extension allows you to specify the values for the address properties. Each locale can have an individual address customization, or can share customizations. Note that multiple countries can share the same address format, but must have different values for the properties. Server-side extensions also allow you to upload a file containing multiple address formats in bulk using a JSON format, or export, modify or re-import address formats. For information about working with server-side extensions, see Develop Server-Side Extensions.

Before you can create custom addresses, you must implement a server-side extension from the Commerce Cloud administration server. See Use server-side extensions in Using Oracle Commerce for details on how to download server-side extensions. If you are using one of Commerce's default tax processors, it is also suggested that you verify that any address modifications you want to make will continue to work with the tax processor validation.

Note: Because Avalara and Vertex Tax Processors do not support adding custom properties, you cannot create customized address formats for either of these integrations.

Server-side extensions define REST endpoints that allow you to customize countries and region/states addresses. Each country and region/state also has a localizable display name.

Work with a server-side extension for customized addresses

Oracle Commerce Cloud provides an example server-side extension that creates customized address formats. The example contains country-specific address formats, which you can use as a template for creating your own address formats. Address properties that are defined using this SSE are stored as custom address properties, which behave just like any other custom address properties. These address properties are included in all APIs and web hooks that use the `address` property, with the exception of those addresses on credit cards, inventory pages or tax processing ship-from-warehouse location addresses.

The server-side extension references a widget that specifies the address formats maintained by JSON files. These JSON files include metadata files, resource bundles and files that map values to each of the properties.

When you download the server-side extension, it is copied locally to your system in a ZIP file. Each ZIP file contains the following files:

- `Package.json` – This file contains the metadata information for the server-side extension. It contains the main entry point, name and public URLS, as well as the description and `devDependencies` and files that are contained in the extension.
- `Index.js` – This file calls the HTTP requests and responses.

- `Readme.md` – A file that describes the server-side extension's classes and endpoints and includes information on installing and extending the extension.

The example SSE contains the following directories:

- `/lib` – This directory contains the class that processes address metadata, which can be extended to call endpoints or make modifications to value mappings. It also contains module constants and address format, metadata and value mappings for three different example address locations for the United Arab Emirates (UAE), the Unites States (US) and Singapore (SG).
- `/node_modules` – This contains a library that wraps the logging utility and should be copied into the `/node_modules` directory of the server-side extension.
- `/out` – This directory contains the output, including styles and the HTML that displays the JavaScript documentation for the SSE.

Define country-specific properties using metadata

The server-side extension metadata allows you define the properties that are included in the address customization. Properties are specific to the country they display. The metadata also allows you to list the order in which the properties are displayed within the administration interface.

Properties can be identified as required or as an internal-only property, and can include default values.

For a property to be displayed with a localized string, the label field for the property must be configured as `resources.`*propertyName*. If the `resources` prefix is not used in the `label` field, the property name will not be localized. The following is an example of a United States-specific address metadata JSON file. For example, note that the properties label field value is prefixed with resources. This ensures that the field is localized.

Property types allow you to indicate if the property is `shortText`, `richText`, `date`, `checkbox` or `enumerated`. Enumerated address properties can be related to one another, which lets you provide a list of values that are displayed across related properties. Although multiple countries can share the same address format, they must have different values for any enumerated properties.

The following is from the `Metadata.json` file of the SSE example:

```
{
   "properties": {
     "address1": {
       "id": "address1",
       "type": "shortText",
       "uiEditorType": "shortText",
       "label": "resources.addressLine1",
       "validations": [
         {
           "type": "required",
           "value": true
         },
         {
           "type": "maxLength",
           "value": 60
         }
```

```
    ]
  },
"address2": {
  "id": "address2",
  "type": "shortText",
  "uiEditorType": "shortText",
  "label": "resources.addressLine2",
  "validations": [
   {
      "type": "required",
      "value": false
    },
    {
      "type": "maxLength",
      "value": 60
    }
  ]
},
"postalCode": {
  "id": "postalCode",
  "type": "shortText",
  "uiEditorType": "shortText",
  "label": "resources.postalCode",
  "validations": [
    {
      "type": "required",
      "value": true
    },
    {
      "type": "maxLength",
      "value": 10
    },
    {
      "type": "regex",
      "pattern": "^[0-9]{5}([ -][0-9]{4})?$"
     }
   ]
},
"state": {
   "id": "state",
   "type": "singleSelect",
   "uiEditorType": "singleSelect",
   "label": "resources.state",
   "parent": "country",
   "validations": [
     {
       "type": "required",
       "value": true
     }
   ]
},
"city": {
   "id": "city",
   "type": "shortText",
   "uiEditorType": "shortText",
```

```
            "label": "resources.city",
            "parent": "state",
            "validations": [
              {
                "type": "required",
                "value": true
              }
             ]
        },
        "phoneNumber": {
            "id": "phoneNumber",
            "type": "shortText",
            "uiEditorType": "shortText",
            "label": "resources.phoneNumber",
            "validations": [
              {
                "type": "required",
                "value": false
              },
              {
                "type": "maxLength",
                "value": 15
              },
              {
                "type": "regex",
                "pattern": "^[0-9()+ -]+$"
              }
            ]
        }
    },
    "propertiesOrder": [
        [
          "address1"
        ],
        [
          "address2"
        ],
        [
          "city",
          "state"
        ],
        [
          "postalCode",
          "phoneNumber"
        ]
     ]
}
```

Notice that the JSON file also identifies the order in which properties will be displayed to the shopper. In the SSE example, the properties included in the US-specific metadata are mapped to the following default properties and in the order in which they will be displayed:

Order No.	Property	Type	Required	Mapped to Default Property	Parent Property
1	address1	ShortText	Yes	address1	N/A
2	address2	ShortText	No	address2	N/A
3	city	ShortText	Yes	city	N/A
4	state	SingleSelect	Yes	state	country
5	postalCode	ShortText	Yes	postalCode	N/A
6	phoneNumber	ShortText	No	phoneNumber	N/A

For contrast, the following is the SSE example of a metadata mapping for the United Arab Emirates (UAE):

Order No.	Property	Type	Required	Mapped to Default Property	Parent Property
1	addressLine1	ShortText	Yes	address1	N/A
3	area	SingleSelect	Yes	city	N/A
4	emirate	SingleSelect	Yes	state	N/A
5	zipPostalCode	ShortText	Yes	postalCode	country
6	phoneNumber	ShortText	N/A	phoneNumber	N/A

Understand resource bundles

The SSE contains sample address formats. You can also upload a file that contains multiple address formats with the associated country mapping. Additionally, you can export address formats, modify or populate them and then import them.

These JSON files provide the locale resources for country-specific properties. The resource bundle JSON file for the US-specific metadata might be:

```
{
   "addressLine1": "Address Line 1",
   "addressLine2": "Address Line 2",
   "state": "State/Region",
   "city": "City",
   "postalCode": "Zip/Postal Code",
   "phoneNumber": "Phone Number"
}
```

While the locale resources for Spain-specific metadata might be:

```
{
   "addressLine1": "Dirección Line 1",
   "addressLine2": "Dirección Line 2",
   "state": "Estado / Región",
   "city": "Ciudad",
   "postalCode": "Código postal",
```

```
   "phoneNumber": "Número de teléfono"
}
```

Understand value mappings

Value mappings are JSON files that provide a value array for the country-specific properties. These mappings allow you to identify specific address data, such as city or state values.

The following is an example of `state.json` in the UAE-specific value mapping file. Note that the `displayName` field value is prefixed with `resources`, ensuring that the display name is localized.

```
{"AE" : [
    {
      "regionCode": "AE-AZ",
      "displayName": "resources.Abu Dhabi",
      "repositoryId": "AE-AZ",
      "abbreviation": "AZ"
    },
    {
      "regionCode": "AE-AJ",
      "displayName": "resources.Ajman",
      "repositoryId": "AE-AJ",
      "abbreviation": "AJ"
    },
    {
      "regionCode": "AE-DU",
      "displayName": "resources.Dubai",
      "repositoryId": "AE-DU",
      "abbreviation": "DU"
...
```

Value mappings are also supported by resource files. The value mapping resource file provides locale information when the `resources` prefix is added to the field. The following is the example of the UAE-specific `ar.json` file, which allows the `displayName` field to show the localized text:

```
{
  "Abu Dhabi": "أبـو ظبـي",
  "Ajman": "عجمـان",
  "Dubai": "دبـي",
  "Fujairah": "الـفـجيـرة",
  "Ras al-Khaimah": "رأس الـخيـمة",
  "Sharjah": "الـشارقـة",
  "Umm al-Quwain": "ام الـقـيـويـن",
  "United Arab Emirates": "الإمـارات الـعربـيـة الـمتحدة"
}
```

Associate endpoints

The address customization server-side extension uses endpoints to exchange address information. Use the `/addresscustomformat/getAddressMetadata` and `/getAddressPropertyValue` endpoints to get the address metadata and properties

respectively. The address formatting endpoints return the address information with the specified parameters. Each country, region and state has a localizable display name. These properties become available for profiles, accounts, self-registration and order addresses, and can be displayed in the administration interface. Note that custom address properties do not appear in the address list of the Account interface and the Registration Request interface.

The example server-side extension for address formats uses custom REST endpoints, which use the prefix `/ccstorex/custom`. For example:

```
/ccstorex/custom/v1/addresscustomformat/getAddressMetadata
```

The `/addresscustomformat/getAddressMetadata` endpoint gets the metadata for the country or context that you select. Issue a `GET` command to obtain the address metadata. For example:

```
GET /ccstorex/custom/v1/addresscustomformat/getAddressMetadata?contextId=US
```

You can use the `/addresscustomformat/getAddressPropertyValue` endpoint to obtain any address property's value. To do this, issue a `GET` command. Note that the `contextId`, `propertyName` and `parentContextID` query parameters are mandatory.

```
/ccstorex/custom/v1/addresscustomformat/getAddressPropertyValue?
contextId=US&parentContextId=US&locale=en&propertyName=state
```

The `locale` parameter can be passed in using a query parameter or the X-CCAsset-Language request header. If you do not pass in the locale, the system will use the default locale passed in by the server-side extension. The response may be something similar to the following:

```
[
   {
     "regionCode": "US-AL",
     "displayName": "Alabama",
     "repositoryId": "US-AL",
     "abbreviation": "AL"
   },
   {
     "regionCode": "US-AK",
     "displayName": "Alaska",
     "repositoryId": "US-AK",
     "abbreviation": "AK"
   },
...
```

Understand the address customization widget

You can display the address customizations to your shoppers by employing it in a widget. The checkout Address Book customization widget example allows you to present your shoppers with a list of countries. The widget displays the address properties included in the associated address format for each locale. The widget allows you to create custom fields, as well as different label text for each field. You can also determine if your customized fields are required or not, as well as provide a default value for each field. For each enumerated property, the widget displays a drop-down of the configured options. Any properties that are

hierarchical display the child property options that are configured for the selected parent value.

The following are portions of the changes made in the example customized `checkout-address-book.js` widget file.

Review the customized widget example

The first thing the example shows is how to add the `ccRestClient`, `viewModels/dynamicProperty` and `ccstoreConfiguration` parameters to the dependencies and model definitions.

Note: Custom address properties are available to Storefront using the view models for contact addresses, account addresses, order details and submission of registration request.

For example:

```
//-------------------------------------------------------------------
// DEPENDENCIES
//-------------------------------------------------------------------
   ['knockout', 'viewModels/address', 'ccConstants', 'pubsub',
    'koValidate', 'notifier', 'ccKoValidateRules', 'storeKoExtensions',
    'spinner', 'navigation', 'storageApi', 'CCi18n','ccRestClient',
    'viewModels/dynamicProperty', 'ccStoreConfiguration'   ],
//-------------------------------------------------------------------
// MODULE DEFINITION
//-------------------------------------------------------------------
function(ko, Address, CCConstants, pubsub, koValidate, notifier,
   rules, storeKoExtensions, spinner, navigation, storageApi, CCi18n,
ccRestClient,
       DynamicProperty, CCStoreConfiguration) {
```

The example then adds the following observables:

```
shippingAddressproperties: ko.observableArray([]),
 shippingAddressPropertiesOrder: ko.observableArray([]),
 addressConfiguredInSSE: ko.observable(false),
```

For example:

```
// Switch between 'view' and 'edit' views
isUsingSavedAddress: ko.observable(false),
isSelectingAddress: ko.observable(false),
billingAddressEnabled: ko.observable(),
addressSetAfterWebCheckout: ko.observable(false),
addressSetAfterOrderLoad: ko.observable(false),
showPreviousAddressInvalidError: ko.observable(false),
previousSelectedCountryValid: ko.observable(false),
shippingAddressBook: ko.observableArray().extend({ deferred: true }),
shippingAddressProperties: ko.observableArray([]),
shippingAddressPropertiesOrder: ko.observableArray([]),
```

```
addressConfiguredInSSE : ko.observable(false),
loadPersistedShipping : ko.observable(false),
```

The example then adds the definition to the default `checkoutAddressBook` widget. The first function is the `populateMetadataForContextConfiguredInSSE`, which initiates an SSE endpoint to retrieve the metadata for a specific locale. Once the widget has performed various validation functions, it uses the `populateSelectedCountryMetadata`.

Note that if you want to use address custom formats with an external address verification service, you should ensure that you develop widgets that contain the format and properties that are required by that service.

Work with address types

An address type is a string, such as Billing or Shipping, that can be associated with a Commerce address.

Address types help shoppers, business users, and administrators keep track of profile and account addresses. Address types can also be used in integrations, as they are included with address details in the bodies of webhooks and API responses. By default, two address types are available in Commerce: Shipping and Billing.

Note: When multiple sites are run from a single Commerce instance, a shopper's profile is shared by all sites. This means that any billing or shipping addresses added to a shopper's profile are available to all sites. By default, any profile address (with any country) can be marked as a shipping or billing address, even if the address might not be a valid shipping or billing address on some sites, or in some account contexts.

Address types can be assigned to profile and account addresses, as well as addresses used in account registration requests. Business users who work with account addresses can automatically select existing address types when creating or editing addresses in the administration interface. However, since default widgets do not include support for address types, shoppers, delegated administrators, and Agent Console users will not be able to work with address types unless you customize widgets that let them see and work with addresses. You can also assign address types with REST API endpoints.

The `Address Types` resource in the Admin API includes endpoints for creating and working with address types. The `Profiles` and `Organizations` resources include endpoints that you can use to set the values of properties of address types.

View address types

You can view address types using the REST API. To view existing address types, first log into the Admin API on the administration server using an account that has the Administrator role. For example:

```
POST /ccadmin/v1/mfalogin HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=admin1@example.com&password=A3ddj3w2&totp_code=3
65214
```

Then issue a `GET` request to the `ccadmin/v1/addressTypes` endpoint.

The following is an example of the response returned. Note that `BILLING` and `SHIPPING` are default address types that are included with Commerce.

```
{
  "total": 3,
  "totalResults": 3,
  "offset": 0,
  "limit": 250,
  "links": [
    {
      "rel": "self",
      "href": "http://myserver.example.com:7002/ccadminui/v1/
addressTypes"
    }
  ],
  "sort": [
    {
      "property": "displayName",
      "order": "asc"
    }
  ],
  "items": [
    {
      "displayName": "Billing",
      "repositoryId": "BILLING",
      "id": "BILLING"
    },
    {
      "displayName": "Office",
      "repositoryId": "at100001",
      "id": "at100001"
    },
    {
      "displayName": "Shipping",
      "repositoryId": "SHIPPING",
      "id": "SHIPPING"
    }
  ]
}
```

Create an address type

To create a new address type, issue a `POST` request to the `/ccadmin/v1/addressTypes` endpoint on the administration server. Specify the value of the `displayName` property in the body of the request. For example:

If the address type is created successfully, the response body returned includes the ID for the new address type and a link to the URL used in the request:

```
{
  "displayName": "Main Campus",
  "repositoryId": "at100002",
  "links": [
    {
      "rel": "self",
```

```
      "href": "http://myserver.example.com:7002/ccadminui/v1/addressTypes"
    }
  ],
  "id": "at100002"
}
```

Assign a type to an address

Business users who work with account addresses can automatically select existing address types when creating or editing addresses in the administration interface. For example, a Commerce administrator can select address types when creating or editing an account's addresses. For more information, see Work with account addresses.

You can also assign an address type with REST API endpoints that create or update profile and account addresses. The following sample request assigns the `Main Campus` address type to a specified account address.

```
PUT /ccadmin/v1/organizations/or-100001/secondaryAddresses/at100001  HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
    "addressType":"Main Campus"
}
```

Customize address validation

You can use the Admin API to specify whether a default address field is required and to create patterns that validate user input for default address fields.

You can customize validations for addresses that are found on profiles, accounts, account registration requests, as well as payment and shipping group pages. Addresses found on credit card, inventory location or the Ship from Warehouse location address in the tax processing setting pages cannot include custom validations.

All the validations described in this section are customized with the Admin API `itemtypes` endpoint and the customizations made to the address field apply on all sites and for all countries, through both the Commerce UIs and the REST APIs. If you need to configure country-specific validations, you must work with a server-side extension (SSE) and custom widget for customized addresses. See Customize address formats using the API for more information.

Validate address input

You can configure Commerce to perform server-side and client-side validation on any default or custom address field to ensure that the value entered matches a regular expression validation pattern you specify. For example, you could require that first and last names contain only letters and be no longer than 20 characters with the validation pattern ^[A-Za-z]{20}.

Note: For number type custom properties, values sent in the payload will be converted to decimals (double) in the API and then validated. To apply validations for this, the validation pattern should be in the form of ^[0-9]+([.][0-9]+)?$, where the first 0-9 represents the integer part and the second 0-9 represents the decimal part. This pattern should be customized according to the requirements of the particular address field.

When you specify a validation pattern for an address field, Commerce applies the validation to values entered through the storefront or administration interface, REST API calls, CSV imports, and bulk imports. It is important to keep in mind that Commerce does not verify that your validation-pattern string is a valid regular expression. Make sure that your validation patterns are appropriate and not contradictory for the fields to which they apply. Otherwise, shoppers and administrators and administrators might get unexpected results when trying to enter addresses.

Use the Admin API `updateItemType/contactInfo` endpoint to add a validation pattern to an address field.

For example, the following sample issues a `PUT` request to `/ccadmin/v1/itemTypes/contactInfo` to apply a validation pattern to US mobile phone numbers so that they are formatted as 111-222-3333:

```
PUT /ccadmin/v1/itemTypes/contactInfo  HTTP 1.1
Authorization: Bearer <access_token>

{"specifications":[
  {
    "id" : "phoneNumber",
    "required" : false,
    "validationPattern": "^[0-9]{3}([ -][0-9]{3}([ -][0-9]{4})?$",
    "default": null
  }
]}
```

Note: In this example, no default value is provided. However, if you do provide a default value, make sure it adheres to the validation pattern set for the property. Additionally, if you are working with custom addresses that you created with server side extensions, make sure that validation patterns for fields such as `postalCode` are broad enough to allow values for all countries your custom addresses support. See Customize address formats using the API for more information.

Make an address field required or optional

Commerce requires that a default address (that is, an address that ships with the product) contain values for a number of address fields, such as City, State, or Country. You can use the Admin API to mark a number of address fields as required or optional, depending on their usage. When you change an address field's `required` property value, that value applies to the field everywhere it is used.

Note: This section describes only default address properties and not to custom properties you have created. For more information about custom address properties, see Customize address formats using the API.

The following table shows which address fields are required and optional by default for different types of Commerce addresses and tasks.

Property	Account: New Address	Individual Shopper: New Address	Account: Place Order	Individual Shopper: Place Order
firstName	Optional	Optional	Required	Required
lastName	Optional	Optional	Required	Required
companyName	Required	Optional	Optional	Optional

Property	Account: New Address	Individual Shopper: New Address	Account: Place Order	Individual Shopper: Place Order
phoneNumber	Optional	Optional	Optional	Optional
address1	Required	Required	Required	Required
city	Required	Required	Required	Required
state	Required	Required	Required	Required
postalcode	Required	Required	Required	Required
country	Required	Required	Required	Required

The `firstName`, `lastName`, `country`, and `companyName` fields, which are required by default, cannot be made optional. Additionally, you cannot make required fields optional for addresses found on credit card, inventory location or the Ship from Warehouse location address in the tax processing setting pages.

Use the Admin API `updateItemType/contactInfo` endpoint to make an address field required or optional.

For example, the following sample issues a `PUT` request to `/ccadmin/v1/itemTypes/contactInfo` to make the first line of an address field optional:

```
PUT /ccadmin/v1/itemTypes/contactInfo  HTTP 1.1
Authorization: Bearer <access_token>

{"specifications":[
{
  "id" : "address1",
  "required" : false
}
]}
```

Keep in mind that the Commerce REST API documentation describes default property values and does not update to match any customizations you make to default address fields. For example, if you make a required address field optional, the REST API documentation will still describe the field as being required.

15

Configure Buy Online Pick Up In Store

By default, Commerce is configured to ship orders to a shopper-provided shipping address, but you can also let shoppers pick up their orders from specified locations, such as brick-and-mortar stores. This section describes how to configure in-store pick up.

Understand buy online pick up in store

You can configure in-store pick up for the following scenarios.

In either scenario, you must create pick-up locations and inventory. See Manage inventory for in-store pick up for more information.

- A shopper picks a store location and the items are in stock at the store. This is the default scenario supported by Commerce and the one described in this documentation.
- A shopper picks a store location and the goods are not in stock, but are shipped from an online inventory location, such as a warehouse, to the shopper's selected store location.

For information about more complex integrations, for example, externally-configured SKUs and asset-based orders, see *Using Oracle CPQ Cloud Features with Oracle Commerce*.

Configure in-store pick up to work with other Commerce Cloud features

This section describes points to keep in mind when you configure in-store pick up to work with your Commerce instance and integrations.

- Commerce orders include a new properties that support in-store pick-up. These properties appear wherever order information is incorporated, for example, REST API endpoints and webhook bodies. See Understand in-store pick up shipping groups to learn about these properties.
- If a shopper's order qualifies for a free gift with purchase and the shopper selects in-store pick up, Commerce adds the gift to the cart but moves it to the in-store shipping group only if it is in stock at the pick-up location. See Create a gift with purchase promotion for more information about how Commerce a
- If you want shoppers to be able to pick up items that include add-on products, the add-ons must also be in stock at the pick-up location. Commerce checks add-on products for inventory to make sure they are available for pick up. If a shopper selects an add-on product that is not available for pick up, the main item is also not available for pick up, even if it is in inventory at the pick-up location.
- Commerce can provide inventory status for configurable SKUs only once the configurable SKUs are in the shopping cart. See *Using Oracle CPQ Cloud Features with Oracle Commerce* for details about configurable SKUs and asset-based orders.
- In-store pick up is available for account-based shoppers without any specific configuration for accounts. For orders that require approval, once the order has been approved, the shopper cannot change in-store pick up to a different shipping method. For more information about account-based shoppers, see Configure Business Accounts.
- If your Commerce store lets shoppers pay for purchases with loyalty points, your store should be configured to display currencies other than loyalty points and the site's

`payShippingInSecondaryCurrency` property should be set to TRUE. Otherwise, when shoppers choose to pay in store (or with any other payment type except points), the order state is set to INCOMPLETE and the order never progresses to fulfillment. To learn more about paying with loyalty points, see Work with Loyalty Programs.

Understand in-store pick up shipping groups

This section describes the shipping group that contains items a shopper will pick up in a store, `inStorePickupShippingGroup`. Commerce automatically creates this shipping group when a shopper selects in-store pick up for an item they add to the cart. To allow a shopper to select in-store pick up along with other shipping methods, you must update the Shipping Options tab in the Checkout Layout. See Configure layouts and widgets for in-store pick up for more information.

The following excerpt from a sample request includes an in-store pick up shipping group.


```
"shippingGroups": [
    {
      "lastName": "Smith",
      "shippingMethod": "inStorePickupShippingGroup",
      "description": "sg60415",
      "submittedDate": null,
      "firstName": "John",
      "priceInfo": {
        "secondaryCurrencyTaxAmount": 0,
        "discounted": false,
        "shippingTax": 0,
        "secondaryCurrencyShippingAmount": 0,
        "amount": 0,
        "rawShipping": 0,
        "amountIsFinal": false,
        "currencyCode": "USD"
      },
      "phoneNumber": "987654321",
      "shipOnDate": null,
      "actualShipDate": null,
      "trackingInfo": [],
      "locationId": "Boston138",
      "specialInstructions": {},
      "middleName": null,
      "commerceItemRelationships": [
        {
          "availablePickupDate": "2018-07-23T12:12:58.000Z",
          "commerceItemId": "ci5000413",
          "inventoryLocationId": "Boston138",
          "amount": 0,
          "quantity": 1,
          "relationshipType": "SHIPPINGQUANTITY",
          "returnedQuantity": 0,
          "preferredPickupDate": "2018-07-23T12:12:58.000Z",
          "range": {
            "lowBound": 0,
            "highBound": 0,
```

```
            "size": 1
          },
          "commerceItemExternalId": null,
          "state": "INITIAL",
          "id": "r60390"
        }
      ],
      "state": "INITIAL",
      "id": "sg60415",
      "stateDetail": null,
      "email": "testTest",
      "handlingInstructions": [],
      "shippingGroupClassType": "inStorePickupShippingGroup"
    }
  ],
```

Manage inventory for in-store pick up

By default, Commerce maintains one set of inventory values for each product or SKU, however, to configure in-store pick up you must maintain multiple inventory locations so that you can provide inventory information to shoppers.

This section describes changes to the endpoints that allow you to implement in-store pick up. For detailed information about the procedures in this section, as well as examples for updating and working with inventory for a specific location, see Manage Multiple Inventory Locations.

To maintain inventory for individual physical stores or web sites, you must represent them in Commerce by creating new locations. Use the `createLocation` endpoint to create a new physical location. You specify information about the location in the body of the request. To allow in-store pick up at a physical location, set its `pickUp` property to true. You must set pickup to true even if you do not plan to have goods in stock at this location, but will have them shipped there from an online inventory location or warehouse.

The following example creates a brick-and-mortar store where shoppers can pick up their purchases:

```
POST /ccadmin/v1/locations HTTP 1.1
Authorization: Bearer <access_token>

{
  "externalLocationId": "107",
  "locationId": "Cambridge02",
  "pickup": true,
  "address1": "221 Third Street",
  "country": "USA",
  "city": "Cambridge",
  "faxNumber": "(617) 386-1200",
  "postalCode": "02141",
  "phoneNumber": "(617) 386-1200",
  "email": "cmb02@example.com",
  "stateAddress": "MA",
  "county": "Middlesex",
  "name": "Cambridge, MA -- 02141",
```

```
  "longitude": -71.0901,
  "latitude": 42.3629
}
```

See Create locations for more details about creating and working with physical locations, including information about how to configure location inventory in an environment that supports multiple sites.

Before you can create an inventory location for your site, you must create or update a site to use the `inventoryLocationId` property. Use the site endpoint to set a default inventory location ID for a custom site that contains the `inventoryLocationId`. For example:

When you create a new inventory location, it initially has no inventory data associated with it. You can use the `createInventory` endpoint to set inventory values for a specific SKU at the location. For example, the following request specifies inventory data for the location created in the previous section:

```
POST /ccadmin/v1/inventories HTTP 1.1
Authorization: Bearer <access_token>

{
  "locationId": "Cambridge02",
  "id": "xsku5014",
  "stockThreshold": 10,
  "stockLevel": 75
}
```

See Create inventory data for locations for more details about adding and working with inventory for physical locations

You can also import inventory data for a location, specified by its `locationId`. See Export and import inventory data for more information.

Configure layouts and widgets for in-store pick up

You must make changes to a number of the storefront layouts to add in-store pick up to your storefront.

This section describes adding new widgets to page layouts and also making sure the latest versions are used for some widgets that are included in the page layouts out of the box. To replace a widget with the latest version, see Upgrade deployed widgets in Customize your store layouts.

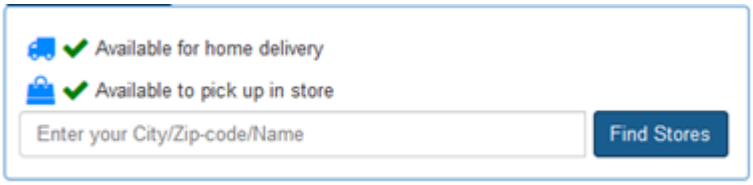
To enable in-store pick up, you must make the following changes to your storefront:

- Add the Store Pick Up element to the Product Details widget. See Update the Product layout for in-store pick up.
- Make Checkout Layout for Multi Ship the default checkout layout. See Update the default checkout layout for in-store pick up.
- Configure the Enable Store Pickup setting for the Shopping Cart Summary and Shipping Options widgets. See Update the default checkout layout for in-store pick up and Update the Cart layout for in-store pick up.

- Update to the latest versions of the Order Confirmation and Order Details widgets.
- If your environment supports scheduled orders, update to the latest version of the Scheduled Order widget. See Configure page layouts for scheduled orders for more information.
- (Optional) Configure widgets to show when an item will be available for pick up. See Display the date and time an item will be available for pick up.

Update the Product layout for in-store pick up

Add the Store Pick Up element to the Product Details widget on the Product layout. (Make sure you are using the latest version of the Product layout and Product Details widget.) This element specifies that an item can be picked up (based on the value of the SKU's onlineOnly property) and lets the shopper search for pick-up locations by postal code or city and state.



When the shopper clicks the Find Stores button, Commerce displays a list of nearby stores where the item can be picked up. For stores where the item is in stock, The number of stores listed depends on the .Enter the Number of Matching Stores to Display value you entered on the Product Details widget's Settings tab. (By default, the widget displays a maximum of ten stores.) The shopper can click a Select Store button for a store that has the item in stock to put the item in the shopping cart.

Note: If you customize the query that finds stores, note that Commerce does not support finding a store by geolocation, that is, by its longitude and latitude coordinates.

Update the Cart layout for in-store pick up

Make sure you are using the latest version of the Shopping Cart widget so you can enable in-store pick up.

To configure the Shopping Cart widget to support in-store pick up:

1. From the **Design** page, select the **Layout** tab.
2. From the **Layout** tab menu, select the **Cart** layout.
3. Click the **Shopping Cart** widget's setting icon to view the widget information.
4. On the **Settings** tab, select the **Enable Store Pickup** check box.
5. Enter the maximum number of stores to display when a shopper searches for a pick-up location from a product's details page.
   By default, the maximum number of stores listed is ten.
6. Click **Save**.

Update the default checkout layout for in-store pick up

Make Checkout Layout for Multi Ship the default checkout layout. This new layout includes a Progress Tracker stack with the following tabs and widgets:

- Login
- Shipping Options

- Payment Methods
- Review Order

Login tab

The Login tab includes a new version of the Login-Checkout widget.

Shipping Options tab

The Shipping Options tab includes a new Shipping Options widget that lets registered and anonymous shoppers select a shipping option on the cart or checkout page. If Enable Store Pickup is selected on the widgets Settings tab, shoppers can search for a store that has the item in stock.

Payment Methods tab

The Payment Methods tab includes the Split Payments widget. The latest version of this widget displays the Pay in Store option if it has been configured and if all items in the order will be picked up at the same location. See Configure payment processing for in-store pick up for more information.

Even if an order qualifies for in-store payment, the Pay in Store option will not appear if gift cards have already been applied to the order. The shopper can remove gift cards from the order and then select Pay in Store. The shopper can pay with the gift cards when they pick up the items at the store.

In the new version of this widget, each payment method includes its own Billing Address section. A payment's billing address overrides the default billing address.

This tab also includes a new version of the Promotions widget.

Review Order tab

The Review Order tab includes the Review Order widget, which has not been updated in this release.

Note: A shopper can view the details of their selected pick-up location when they review their order. However, after they view the pick-up location details in Review Order and then navigate back to Shipping Options to update the selected store, the updated store is not reflected within the Review Order widget.

Display the date and time an item will be available for pick up

You can optionally configure widgets to display the date and time an item will be available for pick up at a store. For example, you could display the availability dates on the product details page, in the shopping cart, during checkout, and in the order history.

Every in-store pick up shipping group in an order contains the following properties:

- `availablePickUpDate` specifies the date and time the items in the shipping group will be at the location and ready to be picked up by the shopper. This data likely originates in your order management system.
- `preferredPickUpDate` specifies the date and time the shopper specifies would like to pick up their items. The date specified by `preferredPickUpDate` should be valid in relation to inventory availability, for example, it should occur after the date specified by `availablePickUpDate` or the `availabilityDate`, but Commerce does not automatically provide validation. The integration you write will need to include code that validates the dates.

Configure products and SKUs for in-store pick up

The `onlineOnly` property is a Boolean that specifies whether an item is allowed to be picked up.

If the property is set to false (default), the item can be picked up; if set to true, the item cannot be picked up, but can be shipped or downloaded.

You can set the property on both products and SKUs. If you set `onlineOnly` to true on a product, all its SKUs automatically inherit the setting; you cannot override the setting for individual SKUs.

For SKU bundles and configurable SKUs, Commerce validates the onlineOnly value only for the main SKU, not for any of the SKUs added to the bundle or .configurable SKU. See Integrate with an External Product Configurator and Create and work with SKUs for more information about creating these types of SKUs.

To set properties on a product, issue a `PUT` request to the `/ccadmin/v1/products/{id}` endpoint on the administration server.

The following sample request sets the `onlineOnly` property for a product:

```
PUT /ccadmin/v1/products/x0215 HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
  "onlineOnly": true

}
```

To set properties on a SKU, issue a `PUT` request to the `/ccadmin/v1/skuProperties/{id}` endpoint on the administration server.

The following sample request sets the `onlineOnly` property for a SKU whose parent product's `onlineOnly` property is set to false:

```
PUT /ccadmin/v1/skuProperties/sku0215a HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
  "onlineOnly": true

}
```

Customize email templates for in-store pick up

You can configure and automatically send the following types of email when a shopper chooses to pick up a purchase in a store:

- Items Ready For Pickup emails are sent when shipping Group Type is `inStorePickupShippingGroup` and the Shipping Group Status for the order is “This email is triggered when the shipping group state is `PENDING_SHIPMENT`.

Commerce sends a separate email for each item in an order that a shopper will pick up.

- Items Picked Up emails are sent when the shopper picks up an item up at a store. This email is triggered when the Shipping Group Status for the order is `NO_PENDING_ACTION`.

When sending these emails to a registered shopper, Commerce uses the email address associated with the shopper profile. When sending emails to an anonymous shopper, Commerce uses the email address from the `inStorePickupShippingGroup`,

The following email templates can also include information about in-store pick up:

- Order Placed
- Order Approved
- Order Payment Initiated
- Order Pending Approval
- Order Quoted
- Order Rejected
- Quote Failed
- Quote Requested
- Scheduled Order Placed Failed
- Store Cancel Order
- Store Return Order

You do not need to download new versions of the templates in order to see in-store pick up details, as these properties are part of the shipping group, and so are automatically added to a template that includes the shipping group object. See Understand buy online pick up in store for details about the order properties that support in-store pick up.

See Configure Email Settings and Customize Email Templates for information about enabling and working with email notifications your store sends to shoppers.

Configure payment processing for in-store pick up

Commerce supports existing payment types and gateways for in-store pick up.

A new payment type, called In Store Payment, lets shoppers pay for items when they pick them up in a store instead of when they place the order online. Commerce displays the In Store Payment type only when all items in an order will be picked up at the same location – that is, when all items in an order are part of the same in-store pick up shipping group or multiple in-store pick up shipping groups for the same store. If an order includes items that will be picked up at multiple locations or items that will be shipped or downloaded, In Store Payment is not available and the shopper must pay for the entire order at checkout.

An order created with the In Store Payment type includes a single in store payment group. The in store payment group functions as a placeholder in the order until the

shopper picks up and pays for the order in store. After the shopper pays in the store, you can update the order with payment details using the `updateOrder` endpoint in the Admin API. (See Order Management REST APIs for more information.)

To update the order with appropriate payment details, issue a `PUT` request to the `/ccadmin/v1/orders` endpoint on the administration server. Replace the existing payment group with new payment groups that represent the actual payment at pick up. For example, the following excerpt from a sample `updateOrder` request body specifies that a shopper paid with cash when they picked up an order:

```
 ...
  "paymentGroups": [
    {
      "paymentGroupClassType": "cash",
      "amountAuthorized": 200,
      "amount": 200,
      "gatewayName": "merchantCashGateway",
      "paymentProps": {},
      "paymentMethod": "cash",
      "state": "PAYMENT_REQUEST_ACCEPTED",
      "id": "pg304412",
      "submittedDate": "2018-04-30T08:25:35.000Z",
      "debitStatus": [],
      "authorizationStatus": [
{
          "amount": 200,
          "statusProps": {
            "sample-addnl-property-key2": "sample-payment-property-value2",
            "responseReason": "1001",
            "sample-addnl-property-key1": "sample-payment-property-value1",
            "merchantTransactionId": "MERCHANT-TRANSACTION-ID",
            "currencyCode": "USD",
            "occs_tx_id": "o30430-pg30441-1525076734070",
            "occs_tx_timestamp": "2018-04-30T08:25:34+0000",
            "merchantTransactionTimestamp": "1447807667046",
            "responseCode": "1000",
            "token": "token-success"
 },
          "transactionSuccess": true,
          "errorMessage": null,
          "externalStatusProps": [],
          "transactionId": "HOST-TRANSACTION-ID",
          "transactionTimestamp": "2015-11-18T00:47:47.000Z"
        }
      ],
      "currencyCode": "USD"
    }
  ],
  "relationships": [
    {
      "paymentGroupId": "pg304412",
      "amount": 200,
      "relationshipType": "ORDERAMOUNTREMAINING",
      "id": "r7i0471"
    }
```

```
  ],
  ...
  "id": "o50444",
  "state": "SUBMITTED",
}
```

To configure the In Store Payment settings in the Commerce administration interface:

1. Click the **Settings** icon.
2. Select **Payment Processing** from the **Settings** list
3. On the **Payment Gateways** tab, select **In Store Payment** from the **Service Type** list.
4. Select one of the following environments to configure.
   Preview: Your store's preview environment

   Agent: The Commerce Agent Console

   Storefront: Your production storefront
5. (Optional) If your Commerce environment is configured for scheduled orders or order approvals, select **Enable For Scheduled Order** or **Enable For Order Approval** to let shoppers pay for those orders if they pick them up in stores.
6. Click **Save**.
7. Publish your changes. See Publish Changes for more information.

See Configure Payment Processing for more information about configuring payment settings in Commerce.

Note: Make sure you are using the latest version of the layouts and widgets that allow shoppers to select in-store pick up and payment. See Configure layouts and widgets for in-store pick up for more information.

Understand tax processing and in-store pick up

When you configure tax processor integrations, you must provide Ship from Warehouse Location properties that specify an address from which items sold on your store are shipped.

The tax processor uses this information when calculating taxes.

In previous releases, Commerce set the Ship from Warehouse Location property per order. To support in-store pick up, this property is now set at the line item level, that is, per shipping group:

- When all items in an order are shipped from a warehouse to the shopper, the Ship from Warehouse Location is the warehouse you specified when you configured your tax processor integration.
- When a shopper picks up all items in the order from a store, the Ship from Warehouse Location is the store address.
- When an order includes a mix of items being shipped and picked up, or items that are being picked up at multiple locations, each shipping group is assigned its own Ship from Warehouse Location.

Commerce automatically sets the ship-to address for in-store pick up shipping groups as the pick-up location. You do not have to configure your tax processor integrations to do this. Commerce uses the address that you specified when you created the location. See Create physical locations for more information.

To learn about configuring the built-in Avalara and Vertex tax processor integrations, see Configure Ship from Warehouse Locations. To learn about configuring an integration with an external tax processor, see Configure Tax Processors.

Configure the Picked Up Items webhook

The Picked Up Items event webhook sends information to one or more external systems when .a shopper completes an in-store pick up.

Commerce invokes the webhook when an order's `InStorePickupShippingGroup` state is changed to `NO_PENDING_ACTION`.

The following properties are sent in the JSON request body of the webhook:

- Meta-data for the `shippingGroups` object that contains the items
- All components of the order object. See Order Submit request example for a sample JSON representation of an order in a webhook body.

To send this data to the external system, you configure the webhook by specifying the URL, username, and password for accessing the system. (See Use Webhooks for more information.)

16

Create Scheduled Orders

Commerce provides a set of widgets and page layouts that allow shoppers to create scheduled orders.

The shopper adds items to her cart and, instead of checking out, creates a schedule that determines the frequency used to fill the order in the future. A service audits the order repository periodically and triggers orders that are scheduled to run. This section describes how to implement the scheduled orders feature.

Configure an invoice payment gateway for scheduled orders

Because scheduled orders are submitted automatically at a future time, they must be paid for using either a stored credit card or by sending an invoice to the shopper.

For information about paying for scheduled orders using a stored credit card, see Support stored credit cards. If, instead, you want to send invoices to shoppers for scheduled orders, you can set up an invoice payment gateway, as described in Integrate with an Invoice Payment Gateway. In addition to those instructions, you must add the `enabledForScheduledOrder` property to the `config.json` file for the gateway. An example of a `config.json` file with the `enabledForScheduledOrder` property is provided below.

```
{
  "configType": "payment",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "instances" : [
    {
      "id": "agent",
      "instanceName": "agent",
      "labelResourceId": "agentInstanceLabel"
    },
    {
      "id": "preview",
      "instanceName": "preview",
      "labelResourceId": "previewInstanceLabel"
    },
    {
      "id": "storefront",
      "instanceName": "storefront",
      "labelResourceId": "storefrontInstanceLabel"
    }
  ],
  "properties": [
  {
      "id": "enabledForScheduledOrder",
      "type": "booleanType",
      "name": "enabledForScheduledOrder",
      "helpTextResourceId": "enabledForScheduledOrderHelpText",
```

```
        "labelResourceId": "enabledForScheduledOrderLabel",
        "defaultValue": true,
        "public" : true
    },

    {
        "id": "paymentMethodTypes",
        "type": "multiSelectOptionType",
        "name": "paymentMethodTypes",
        "required": true,
        "helpTextResourceId": "paymentMethodsHelpText",
        "labelResourceId": "paymentMethodsLabel",
        "defaultValue": "invoice",
        "displayAsCheckboxes": true,
        "options": [
           {
             "id": "invoice",
             "value": "invoice",
             "labelResourceId": "invoiceLabel"
           },
           {
             "id": "card",
             "value": "card",
             "labelResourceId": "cardLabel"
           }
         ]
      }
    ]
}
```

Configure the scheduled order service

A scheduled order service runs periodically to review the order repository and determine if any scheduled orders are due to be filled.

Out of the box, this service is not set to run, so you must provide a frequency for it before scheduled orders can work.

To set the initial frequency of the scheduled order service, you issue a `POST` request to the `scheduledJobs` endpoint, with a payload that defines the schedule, an example of which is provided below. To update the schedule, you issue a `PUT` request to the same endpoint.

```
POST /ccadmin/v1/merchant/scheduledJobs

{
"componentPath": "ScheduledOrderService",
"scheduleType": "periodic",
"schedule":
  {
   "period" : 1000000
  }
}
```

The `scheduleType` and schedule properties determine the frequency used when running the scheduled order service. The `scheduleType` property may be either periodic (time-based) or calendar (calendar-based).

If you specify periodic for the `scheduleType` property, you must also provide a period property in milliseconds.

If you specify calendar, you must provide additional properties that further define the frequency. These properties include:

- `occurrenceInDay`: This field can be set to 1 or 2, representing once a day or twice a day, respectively. The default is 1, in other words, if `occurenceInDay` is not specified, the order will be run once on the specified days.
- `daysOfWeek`: This field can be set to 1, 2, 3, 4, 5, 6, or 7, which correspond to (and can be represented in the UI as) Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday.
- `weeksInMonth`: This field can be set to 1, 2, 3, 4, or 5, which correspond to (and can be represented in the UI as) the first, second, third, fourth, or last week of the month.
- `monthsInYear`: This field can be set to 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, which correspond to (and can be represented in the UI as) January, February, March, April, May, June, July, August, September, October, November, and December.
- `daysInMonth` : This field corresponds to the day of the month and it can be set to a number between 1 and 31 (depending on the number of days in the month).

The following combinations of these properties are allowed:

- `occurenceInDay`, `daysOfWeek`, `weeksInMonth`, `monthsInYear`
- `monthsInYear`, `daysInMonth`, `occurenceInDay`

Note: If both `daysInMonth` and `daysOfWeek` are sent in the payload, then `daysInMonth` takes priority.

To help you better understand how to combine these properties, some examples are provided below.

This schedule runs twice on Sunday, Monday, and Tuesday of the first, second and third weeks of February and April.

```
"schedule":{    "daysOfWeek":[1,2,3],
           "weeksInMonth":[1,2,3],    "monthsInYear":[1, 3],
           "occurrenceInDay":2 }
```

This schedule runs twice a day:

```
"schedule":{    "occurrenceInDay":2 }
```

This schedule runs once every Sunday:

```
"schedule":{    "daysOfWeek":[1] }
```

This schedule runs twice on Monday in the second and third weeks of every month:

```
"schedule":{    "daysOfWeek":[2],    "weeksInMonth":[2,3],
           "occurrenceInDay":2 }
```

This schedule runs once on the first day of every month:

```
"schedule":{    "daysInMonth ":[1] }
```

This schedule runs once on the fifth day of June:

```
"schedule":{    "daysInMonth ":[5],    "monthsInYear": [5]
        }
```

This schedule runs once on the first day of January, March, May, July, September, and November:

```
"schedule":{    "daysInMonth ":[1]
           "monthsInYear":[0, 2, 4, 6, 8, 10] }
```

This schedule runs on the 10th, 15th, 20th, and 21st days of February and December:

```
"schedule":{    "daysInMonth":[10,15,20,21],
           "monthsInYear":[1, 11] }
```

Configure page layouts for scheduled orders

This section describes the page layouts that are affected by the scheduled orders feature, along with the changes and verifications you need to make to implement scheduled orders.

Several of the sections below require you to ensure that you are using the latest version of a widget. To replace a widget with the latest version, see Customize your store layouts.

Checkout Layout for Scheduled Orders

To implement scheduled orders, you must add the Scheduled Order – Checkout widget to the checkout page you are using (either Checkout Layout or Checkout Layout with GiftCard). This widget presents UI controls to the shopper that capture the schedule information for a scheduled order, including a name for the scheduled order, the start and end dates, the frequency, and the option to suspend the scheduled order.

Also on the checkout layout, make sure you are using the latest version of these widgets:

- Order Summary – Checkout widget. The latest version of this widget toggles the Place Order button to a Schedule Order button when the shopper selects the option to create a scheduled order.
- Payment Gateway Options widget. The latest version of this widget disables the Cash payment option when the scheduled order checkbox is selected.

- Gift Card widget. The latest version of this widget disables the gift card payment option when the scheduled order checkbox is selected.

Profile Layout for Scheduled Orders

The Scheduled Order List widget is available for use on the Profile Layout. This widget displays a list of scheduled orders associated with the current shopper's profile. Clicking the link for one of the scheduled orders displays the Scheduled Order Layout with details for the selected order.

While it is not configured this way out of the box, you can create a vertical tab stack on your Profile Layout and place the Scheduled Order List widget on one of the tabs. For more information on vertical tab stacks, see Add vertical tabs in Design Your Store Layout.

Scheduled Order Layout for Scheduled Orders

The Scheduled Order Layout displays the details of the selected scheduled order. To access the Schedule Order Layout, a shopper must click a scheduled order link from the list displayed by the Scheduled Order List widget on the Profile Layout. See Profile Layout for Scheduled Orders for details.

Out of the box, the Scheduled Order Layout is populated with an instance of the Scheduled Order widget, which is the widget responsible for rendering the scheduled order's details, which include:

- The scheduled order's name and ID.
- The next order date.
- The schedule details, including schedule name, start and end date, frequency, and a suspend option. Note that the shopper can modify the schedule details but she cannot modify any other details about the scheduled order. See Modify the schedule of a scheduled order for more information.
- Previously fulfilled orders, including any reasons for failed orders.
- The contents of the scheduled order.
- The shipping address and method for the scheduled order.

This widget also provides an option to delete the scheduled order and to place a one-time order based on the scheduled order's content. See Delete a scheduled order and Place a one-time order based on a scheduled order for more information, respectively.

Order History Layout for Scheduled Orders

On the Order History Layout, you must make sure you are using the latest version of the Order History widget. This version includes orders placed as part of a scheduled order in the order history list. This version also provides the option to display a scheduled order ID column for orders generated from a scheduled order template.

Order Details Layout for Scheduled Orders

On the Order Details Layout, make sure you are using the latest version of the Order Details widget. This version includes the scheduled order name for any orders placed via a scheduled order.

Update prices in a scheduled order

It is possible that prices will change for items in a scheduled order over time.

The Scheduled Order Layout, which displays the details for a selected scheduled order, displays the latest prices for the scheduled order. Each time a schedule order is triggered, the most current prices are applied to the order's items. Also, any applicable promotions that are available at the time the scheduled order is triggered are applied.

Notify shoppers about scheduled order activity

The Order Placed email template sends an email to the shopper whenever a scheduled order is triggered.

In addition to the regular order details, this template is augmented with information relevant to scheduled orders such as the scheduled order name, the scheduled order ID, and the next time the schedule will run.

The Scheduled Order Error email template sends an email to the shopper when a scheduled order fails to run along with an error message that indicates why the order failed.

For more information on email templates, see Customize Email Templates.

Understand shopper tasks for scheduled orders

The configuration described in the preceding sections provides shoppers with the functionality described below.

Note that, when a contact is removed from an account, all scheduled orders associated with that contact expire.

Place a scheduled order

On the checkout page, the shopper can enable the Scheduled Order option to create a scheduled order from his cart contents instead of placing the order immediately. When the shopper enables this option, he is presented with a scheduled order form that allows him to specify a name for the scheduled order, define the start and end dates, and set the frequency. Out of the box, frequencies include:

- Once a day
- Weekly. When this option is selected, the shopper can specify the day of the week (Sunday through Saturday) or the weeks of the month (First, second, third, fourth, last) for when an order should be run.
- Monthly. When this option is selected, the shopper can select once a month, every two months, or quarterly for when the order should be run.

The shopper also has the option to suspend the order. Suspended orders are not run until they are re-activated. See Suspend a scheduled order for details.

Modify the schedule of a scheduled order

A shopper can modify the schedule of an existing scheduled order but the contents of the order cannot be modified, nor can the shipping method or address. Instead, the shopper would have to create a new scheduled order with the modified contents and shipping details and then delete the old one.

Suspend a scheduled order

A shopper can suspend a scheduled order, either while they are first creating the scheduled order on the checkout page, or while they are viewing the scheduled order's details. A suspended order will not run until it is re-activated from the scheduled order's details page.

Place a one-time order based on a scheduled order

When viewing a schedule order's details, a shopper can place an immediate, one-time order using the scheduled order's contents by clicking the **Place Order** button. Clicking this button places the contents of the scheduled order into the shopping cart. If any other products exist in the cart already, the contents of the scheduled order are added to them. The shopper then checks out as normal.

Delete a scheduled order

To delete a scheduled order, the shopper must view the scheduled order's details, then click the **Delete** button.

Manage a failed scheduled order

Scheduled orders can fail for a variety of reasons, for example, a product in the scheduled order is out of stock or discontinued, or the price list group associated with the scheduled order has been removed so the order can no longer be priced.

Depending on the failure type, the shopper may be able to quickly create a new scheduled order that does not have the issue (for example, a new order that removes a discontinued product). To do this, he can view the details page for the problem scheduled order, click the **Place Order** button to add the order's contents to the cart, modify the cart contents (for example, to remove a discontinued product), go to the checkout page and create a new scheduled order. The shopper will also have to delete the problem scheduled order as described in Delete a scheduled order.

If the shopper cannot resolve the issue, he will have to contact an agent and provide the error information to get assistance.

17

Notify Shoppers When Items are Back in Stock

You can use the Admin API to enable shoppers to receive email notifications for out of stock products when they arrive back in stock.

This section describes how to create the notification extension, configure the scheduler, and add the Notify Me element to the Product Details widget.

If your Oracle Commerce environment integrates with an external system to notify shoppers when items are back in stock, configure the Back in Stock event webhook to notify the system that handles your shopper notifications. See Understand event webhooks for more information.

Understand back in stock notifications

Shoppers on your store can choose to receive a back in stock email notification for out of stock products.

To do so, they click the **Notify Me** link within the **Product Details** page on the storefront. After clicking the link, shoppers are prompted to enter their email address. Once confirmed, they receive a confirmation message informing them that they will be notified when the product becomes available. When back in stock, the shopper receives an email notification containing a link to the previously out of stock product.

In order for the Notify Me link to appear to shoppers when products are out of stock, you must have the Notify Me email template enabled. See Enable the types of email your store sends for more information.

Create and upload the notification extension

You can use an extension to add a Notify Me button to your Product Details widget, which is then displayed to shoppers when they view a product that is out of stock.

You must create and upload the notification extension in order to create the Notify Me element. To do so:

1. Click the **Settings** icon.
2. Click **Extensions** and display the **Developer** tab.
   Before you develop an extension, you must generate an ID that you will include in your extension file.
3. Click **Generate ID**, and you are prompted to name the extension.
4. Name the extension **Notify Me**, and click **Save**.
   Your extension ID is now generated and must be used in the extension's `ext.json` file.
5. Edit the `ext.json` file and set the `extensionID` property to the value generated in the previous steps.

6. Package all the files within your <extension-name> directory in a ZIP file.
7. Open the **Installed** tab, and click **Upload Extension** to upload the extension to the administration interface.
8. Select the extension ZIP file from your local file system.
9. Publish your changes.

Create a Notify Me element

The Product Details widget is separated into elements, (see Fragment a Widget into Elements for more information) including the Notify Me element. To create an element to display the Notify Me button, you must create a `template.txt` file providing the HTML rendering code for the element. The contents of the `template.txt` file look similar to the following:

```
<!-- Notify Me Element -->
<!-- ko if: initialized() && $data.elements.hasOwnProperty('product-
notify-me') -->
  <div class="notify-me" id="cc-notify-me-container" data-
bind="visible: $data.elements['product-notify-me'].showNotifyMe">
    <a href="#" data-bind="click : $data.elements['product-notify-
me'].notifyMe.bind(this)" id="CC-notifyMe-link">
      <span id="CC-notifyMe-label" data-bind="widgetLocaleText:
'notifyMeText'"></span>
    </a>
  </div>

  <!-- Popup -->
  <div class="modal fade" id="CC-notify-me-dialog" tabindex="-1"
role="dialog">
    <div class="modal-dialog cc-modal-dialog" id="CC-notify-me-dialog-
content" role="document">
      <div class="modal-content">
        <div class="modal-header CC-header-modal-heading">
          <h3 data-bind="widgetLocaleText:'notifyDialogTitle'"></h3>
        </div>
        <div class="modal-body cc-modal-body">

          <p data-bind="widgetLocaleText: 'notifyOutOfStock'"></p>
          <p data-bind="widgetLocaleText: 'notifyInstructions'"></p>

          <div class="form-group row">
            <div class="controls col-md-12">
              <label class="control-label inline" for="CC-login-input"
data-bind="widgetLocaleText:'emailAddressText'"></label>
              <span role="alert" class="text-danger" id="CC-notify-
email-input-error" data-bind="validationMessage:
user().emailAddress"></span>
              <input type="email" class="col-md-5 form-control" id="CC-
notify-email-input" aria-required="true"
                     data-bind="validatableValue:
user().emailAddress, widgetLocaleText: {value: 'emailAddressText',
attr:'placeholder'},
                     event:
{ blur: $data.user().emailAddressLostFocus.bind(this),
```

```
focus: $data.user().emailAddressFocused.bind(this) }" />
            </div>
          </div>

          <div id="CC-notify-me-footer" class="modal-footer CC-header-modal-
footer">
            <div class="center-block">
              <button class="cc-button-primary" data-bind="widgetLocaleText:
'confirmText', click: $data.elements['product-notify-
me'].confirm.bind(this)" />
              <button class="cc-button-secondary" data-
bind="widgetLocaleText: 'cancelText', click: $data.elements['product-notify-
me'].cancel.bind(),  event: {mousedown: $data.elements['product-notify-
me'].handleMouseDown.bind($data, $parent), mouseup: $data.elements['product-
notify-me'].handleMouseUp.bind($data, $parent)}" />
            </div>
          </div>

        </div>
      </div>
    </div>
  </div>

<!-- /ko -->
```

This file must be located in the **`NotifyMe/templates`** folder.

Define the Notify Me element meta-data

The Notify Me element requires a manifest file, called `element.json`, to define key properties. The contents of the `element.json` file look similar to the following:

```
{
  "inline" : true,
  "supportedWidgetType":["productDetails"],
  "translations" : [
    {
      "language" : "en",
      "title" : "Notify Me",
      "description" : "Element to allow setting of back in stock
notifications"
    }]
}
```

Define the Notify Me element.js

The Notify Me element requires an `element.js` file, the contents of the which look similar to the following:

```
define(
//-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
['jquery', 'knockout', 'navigation', 'ccConstants', 'ccRestClient',
```

```
'pubsub', 'notifier'],
  //
------------------------------------------------------------------------
  // MODULE DEFINITION
  //
------------------------------------------------------------------------
  function ($, ko, navigation, CCConstants, CCRestClient, pubsub,
notifier) {    "use strict";
    return {
      elementName: 'product-notify-me',
      showNotifyMe: ko.observable(false),

      onLoad: function(widget){
        var self = this;

       $.Topic(pubsub.topicNames.PRODUCT_VIEWED).subscribe(function(pro
duct){
  self.setVisible(widget);
});        $.Topic(pubsub.topicNames.SKU_SELECTED).subscribe(function(p
roduct, sku, variant){
  self.setVisible(widget);
});

widget.stockStatus.subscribe(function(newValue){
  self.setVisible(widget);
});

   $("[id^='CC-prodDetails-'").on("change", function(evt, data){
     if ($(evt.target).find("option:selected").index() == 0){
       //the "Select..." option has been selected, hide the notify-me
       self.showNotifyMe(false);
     }
   });

//Set up subscriptions to user product notification
events        $.Topic(pubsub.topicNames.USER_PRODUCT_NOTIFICATION_SUCCE
SS).subscribe
  (function(data){

notifier.sendSuccess(widget.WIDGET_ID,widget.translate('notifySuccess')
);        });

$.Topic(pubsub.topicNames.USER_PRODUCT_NOTIFICATION_FAILED).subscribe
  (function(data){

notifier.sendSuccess(widget.WIDGET_ID,widget.translate('notifyFailed'))
;        });

widget.confirmNotify = function(widget, email){
  var inputData = {}, skuId = '';

  inputData.siteId = widget.site().siteInfo.id;
  inputData.productId = widget.product().id();

  if (widget.selectedSku()){
```

```
    skuId = widget.selectedSku().repositoryId;
  }else{
    //SKU for top level product
    skuId = widget.product().stockStatus().catRefId || '';
  }
  inputData.skuId = skuId;

  inputData.profileId = widget.user().repositoryId();
  inputData.email = email;
  inputData.locale = widget.locale();
  inputData.expiryDate = "";

  widget.user().createProductNotification(inputData);

  $("#CC-notify-me-dialog").modal("hide");
 }
},
confirm: function(widget, event){
  if (widget.user().emailAddress.isValid()) {
    var email = $("#CC-notify-email-input").val();
    widget.confirmNotify(widget, email);
  }
},
notifyMe: function(widget){
  notifier.clearError(widget.WIDGET_ID);
        if (!widget.user().loggedIn()){
          widget.user().reset();
          $("#CC-notify-email-input").val("");
          $("#CC-notify-me-dialog").modal("show");
          $('#CC-notify-me-dialog').on('shown.bs.modal', function () {
            $("#CC-notify-email-input").focus();
          });
          $('#CC-notify-me-dialog').on('hidden.bs.modal', function() {
           $("#CC-notifyMe-link").focus();
          });
        }else{
          widget.confirmNotify(widget, widget.user().email());
        }
      },
      cancel: function(){
        $("#CC-notify-me-dialog").modal("hide");
      },
      setVisible: function(widget){
        var hide = false;

        //if a product has variants, but none are selected, don't show the
           notify me option
        if (widget.productVariantOptions &&
        widget.productVariantOptions()){
          if (!widget.selectedSku()){
            hide = true;
         }
         //always allow options to be selected
          if (widget.disableOptions){
            widget.disableOptions(false);
```

```
            }
          }
          this.showNotifyMe(!widget.stockStatus() && !hide);
        },
        /**
         * Ignores the blur function when mouse click is up
         */
        handleMouseUp: function(data) {
          this.ignoreBlur(false);
          data.user().ignoreEmailValidation(false);
          return true;
        },
        /**
         * Ignores the blur function when mouse click is down
         */
        handleMouseDown: function(data) {
          this.ignoreBlur(true);
          data.user().ignoreEmailValidation(true);
          return true;
        }
      };
    }
);
```

Add the Notify Me element to the Product Details widget

In order to enable shoppers to receive back in stock email notifications, you will need to add the Notify Me element to the Product Details widget on the Product layout within the Design page.

To do so:

1. Click the **Design** page, and open the **Layout** tab.
2. Open the **Product** layout.
3. Click the **Grid View** icon.
4. Open the **Product Details** widget settings.
5. From the **Layout** tab, open the **Element Library** located at the bottom of the page.
6. Locate the **Notify Me** element, and drag to the relevant position on the **Product Details** widget.
7. Click **Save** to confirm.

Configure the scheduler to send the back in stock emails

Oracle Commerce runs a scheduler that determines when to send out back in stock email notifications by performing periodic reviews of the inventory.

The scheduler must be configured with an endpoint, and have a frequency set as, out of the box, this service is not automatically set to run.

To set the initial frequency of the scheduled product notification service, you issue a `POST` request to the `scheduledJobs` endpoint, with a payload that defines the schedule, an example of which is provided below. To update the schedule, you issue a `PUT` request to the same endpoint.

```
POST /ccadmin/v1/merchant/scheduledJobs

{
"componentPath": "ProductNotificationService",
"scheduleType": "periodic",
"schedule":
  {
   "period" : 1000000
  }
}
```

See Configure the Scheduled Order Service to learn how to use these properties to set how frequently the service runs.

The frequency of the inventory reviews is dependent on the scheduler configurations. **Note:** If the product becomes unavailable, then the scheduler will delete the notification request. Likewise, if the notification request expires, the scheduler deletes the notification request. (The default notification expiration time is three months.)

18

Enable Purchase Lists

Commerce provides a set of widgets and page layouts that allow shoppers to create purchase lists, which are lists that shoppers can use to quickly access frequently purchased items.

Purchase lists can be enabled in both Oracle Commerce consumer and account-based sites.

This section describes how to manage purchase lists.

Understand the difference between wish lists and purchase lists

A wish list allows shoppers to maintain a list of items that they are considering purchasing, also allowing them to use social media to share and comment on the list.

Purchase lists allow shoppers to maintain a quickly accessible list of frequently purchased products that can be added to an order. While both of these lists are similar, there are some distinctions.

Understand wish list features

Wish lists allow shoppers to create a list of items that they may want to purchase, or have other shoppers purchase for them. These lists can be shared using social media, and allow for social experiences from others that are invited to view the list. Once an item from a wish list has been included in an order, the item does not remain in the wish list.

Wish lists are best used by shoppers that want to communicate and share the list with others.

Wish lists provide your shoppers with:

- A way to create and manage a list of products they are considering buying. Included is the ability to save items in wish lists and purchase easily from them later.
- A social experience where shoppers can share their lists and receive feedback and suggestions from friends and family members they invite to their wish lists.
- Email notifications for key wish list-related activities.
- Registered shoppers can create up to 50 wish lists. The number of product posts, text posts, and comments on an individual post is set to 100 per wish list.

For detailed information on wish lists, refer to Understand wish list features.

Understand purchase list features

Purchase lists are created by shoppers and include items that they have selected. These lists, which do not include pricing information or shipping and bill addresses, can be created from the shopper’s profile page, the product detail page or the order detail pages and include the following:

Property	Description
Name	A unique name for the purchase list. The purchase list name can be modified by the owner of the list.
Description	A description for the purchase list. This field is not exposed in the UI.
Items	The shopper adds SKUs and associated quantities. A shopper can save a purchase list that contains no SKUs. The item name is a link to the SKU's Quick View.

Purchase lists can be used as reminders for frequently purchased items or item combinations. These lists are also helpful when a buyer purchases items on a frequent basis, but does not want to create a scheduled order. For example, a buyer may purchase office supplies frequently, and would like to be able to access the items without searching through the catalog. By creating a list that contains a list of frequently purchased office supplies, the buyer can quickly add these items to the order.

Purchase lists are associated with a specific buyer. When a shopper creates a purchase list, it is available on all sites. If your environment is configured for account-based commerce, purchase lists are available on all sites and in all accounts.

Unlike a wish list, when an item is added to an order, the item remains on the purchase list.

Purchase lists provide shoppers with the following features:

- A way to create and manage a list of products they buy consistently, including the ability to save items in purchase lists and add them to orders later.
- The ability to create and manage multiple purchase lists.
- The ability to share purchase lists you have created. See Share purchase lists for more complete information.
- Once configured, buyers can search and view lists of purchase lists they created. The list views available are a view for purchase lists the shopper actually created as well as a view of purchase lists shared with/by the shopper.
- Registered shoppers can add items to the purchase list from the product details page, their profile, or from the order detail page.
- Registered users can add a product from a purchase list to a shopping cart for purchase.
- The ability to edit and delete purchase lists, including modifying purchase list names, adding and removing items from purchase lists and modifying the quantity of items within the purchase list. Note that the quantity of an item in a purchase list cannot be null.
- There is no limit to the number of purchase lists that a shopper can make. (Note that having a large number of purchase lists may affect server performance.)

Purchase lists can be viewed by shoppers when working on their profile, viewing product details or viewing order detail information. When a shopper views a list of purchase list available on their profile, they are presented with the following information:

Property	Description
Name	The name of the purchase lists that they created.
Number of Purchase List items	The total number of items included within each list.
Date Last Modified	The date when the list was last modified.
Delete Button	An icon that allows a shopper to delete an item.
Link to create a new purchase list	A link that allows a shopper to create a new purchase list.

When a shopper opens the list of purchase lists available to them, they can select a specific purchase list to review.

Work with purchase lists

The following instructions describe the steps that are needed to work with purchase lists.

Purchase lists are comprised of the following widgets and layouts:

Widget	Layout	Description
Purchase Lists	Shopper Profile Layout	From the Profile Layout, a shopper can view a list of their purchase list and select a purchase list to modify. **Note:** The Purchase List widget is included by default in account-based environments.
Purchase List Details	Purchase List Details Layout	The Purchase List Details Layout displays the Purchase List Details widget. The widget displays a list of purchase lists, and details of individual purchase lists, as well as allows a shopper to use a search box to search for products.
Product Details	Product Layout	The Product Details widget allows a shopper to add items to a new or existing purchase list using a checkbox element.
Order Details	Order Confirmation Layout and Order Details Layout	When a shopper accesses the Order Details widget, all items are selected for inclusion into a new or existing purchase list. The shopper can select specific items to add to a new or existing purchase list. By default, all of the items on the Order Details page are added to the purchase list, however the shopper can modify the selection as needed.

Widget	Layout	Description
Order Details with Additional Info	Order Confirmation Layout and Order Details Layout	When a shopper accesses the Order Details with Additional Info widget, as with the Order Details widget, all items are selected for inclusion into their purchase list. The shopper can select specific items to add to a new or existing purchase list. The Order Details with Additional Info widget includes, by default, the purchase list elements that are displayed to enable to shopper to add items to their purchase list.

For information on working with widgets and layouts, refer toUnderstand widgets .

Configure purchase list widgets

To display purchase lists, you must include the purchase list widgets in the appropriate layouts, as well as include the purchase list elements. To make purchase lists available to shoppers:

1. Place the Purchase List widget on the profile layout. Note that in account-based environments, the widget is added by default to the profile layout.
2. Once the Purchase List widget has been added to profile layout, configure the Order Details widget to include the purchase list elements. By default, the purchase list elements are disabled.
3. Ensure that the purchase list elements have been added to the Order Details with Additional Info widget. The elements are enabled by default on this widget.
4. Configure the Product Detail Page widget to include the Add to Purchase List elements.

The purchase list elements place an **Add to Purchase** button and checkboxes next to each item that can be added to the list.

Note: For additional info on configuring the Purchase List widget so that it supports the sharing of purchase lists, refer to Share purchase lists.

For detailed instructions on working with widgets and layouts, refer to Create a Widget. To replace a widget with the latest version, refer to Upgrade deployed widgets in Customize your store layouts.

Configure search for purchase lists

When populating a purchase list, you can configure the environment to allow a shopper to search for products and get a type-ahead dropdown of product names. The `TypeAhead` search interface, which is known in the Oracle Commerce Search administration UI as a searchable field ranking, displays a search box that is separate from the standard search. This search box presents a dropdown list of products that match the `TypeAhead` criteria.

To enable shoppers to search using the `TypeAhead` search box in a specific field of a purchase list, do the following:

1. Determine which field should be made searchable.
2. Log in to the Oracle Commerce administration interface.
3. Mark the field as Searchable as described in both the #unique_147/unique_147_Connect_42_TITLE_JVC_GN2_RHB section and Add fields to the searchable field ranking list.
4. Publish the changes that you made to the field.
5. Use the **Search** tab to access the **Searchable Field Ranking** page.
6. Open the `TypeAhead` ranking as described in the Understand the searchable field ranking list.
7. Once you have added the field, and positioned it in the desired position relative to the other fields, save your changes.

For additional information on configuring search settings, refer to the Manage Search Settings.

Work with the purchase list API

The purchase list API allows you to create purchase lists that are accessible only on specific sites, or for account-based shoppers, only on specific sites and/or accounts.

The API also allows you to create purchase lists that are accessible in all contexts, like the purchase lists that are created using the UI, as well as to create custom properties for line items.

The following tasks can be performed using the purchase list API. For additional information on working with APIs, refer to the Use the REST APIs.

Create a purchase list with the API

When a shopper creates a purchase list, it is available in all site and account contexts for consumer commerce or account based-commerce, however, you can specify specific sites and/or accounts.

Issue a `POST` request to `/ccstore/v1/purchaseLists`. For example:

```
{
  "name":"Purchase List 1",
  "description":"Purchase List",
  "siteId":"siteUS",
  "accountId":"or-100001",
    "items": [{
    "productId":"Product_18Cxi",
    "catRefId": "Sku_18Dxi",
    "quantityDesired" : 25
  },
{
    "productId":"Product_5Cx",
    "catRefId": "Sku_5Cxy",
    "quantityDesired" : 30
  }
```

```
]
}
```

Update a purchase list with the API

To update a purchase list issue a `PUT` request in the following format: `/ccstore/v1/purchaseLists/{id}` where you provide the purchase list ID. For example, you could use the following to update the name or description of the purchase list with the following:

```
PUT /ccstore/v1/purchaseLists/gl42244
```

To update a specific purchase list item, use the `/ccstore/v1/purchaseLists/{id}/updateItems`. The following example removes one of the products:

```
{
    "items": [{
    "productId":"Product_18Cxi",
    "catRefId": "Sku_18Dxi",
    "quantityDesired" : 25,
    "op": "update"
  },
{
    "productId":"Product_5Cx",
    "catRefId": "Sku_5Cxy",
    "quantityDesired" : 100,
    "op": "delete"

  }
]
}
```

Delete a purchase list with the API

To remove a purchase list, issue a `DELETE` request and the ID of the purchase list. For example:

```
DELETE /ccstore/v1/purchaseLists/gl42244
```

Once the purchase list has been removed, the shopper will get a confirmation prompt.

Use custom properties on purchase lists and their line items

You can use the API to add custom properties to a purchase list itself, or a purchase list's line items.

Use the `updateItemType` endpoint to create new custom properties. You can also use this API to update existing custom properties. Purchase lists are accessed from the `gift-list` item type. To create or update a custom property for the `gift-list`, issue a `PUT` command in the request body as follows:

```
//updateItemType
PUT /ccadmin/v1/itemTypes/gift-list
```

```
{
  "specifications": [
    {
      "id": "customPLProperty_1",
      "label": "Custom PL Property 1",
      "type": "shortText",
      "uiEditorType": "shortText",
      "required": false,
      "default": null,
      "searchable": true,
      "localizable": false
    }
  ]
}
```

To delete a custom property for a purchase list, issue a `DELETE` command:

```
//deleteitemTypeProperty
DELETE /ccadmin/v1/itemTypes/gift-list/properties/customPLProperty_1
```

You can review the metadata of the custom property by using the `getItemType` API:

```
//getItemType
GET /ccadmin/v1/itemTypes/gift-list
GET /ccagent/v1/itemTypes/gift-list
GET /ccstore/v1/itemTypes/gift-list
```

Create custom properties for line items

You can create custom properties for purchase list line items, which allows you to do such things as add notes to a purchase list item or provide additional product information about the item. Custom properties can be used by the purchase list widget, as described in Configure Purchase Lists. You can expose the property in your customized purchase list widget. You can customize the widget code using copies of an item from a purchase list to a cart such that when it copies the item, it also copies the content of the custom property to a custom property that has already been defined on the order line item.

Items in a purchase list are accessed from the `gift-item` item type. To create or update a custom property for the `gift-item` item type, issue a `PUT` command in the request body as follows:

```
//updateItemType
PUT /ccadmin/v1/itemTypes/gift-item

{
  "specifications": [
    {
      "id": "customPLIProperty_1",
      "label": "Custom PL Item Property 1",
      "type": "shortText",
      "uiEditorType": "shortText",
      "required": false,
```

```
      "default": null,
      "searchable": true,
      "localizable": false
    }
  ]
}
```

To delete a custom line item property, issue a `DELETE` command:

```
DELETE /ccadmin/v1/itemTypes/gift-item/properties/customPLIProperty_1
```

You can review the metadata of the custom property by using the `getItemType` API:

```
GET /ccadmin/v1/itemTypes/gift-item
GET /ccagent/v1/itemTypes/gift-item
GET /ccstore/v1/itemTypes/gift-item
```

Use the `updateItems` API to modify custom properties on purchase list line items. For example:

```
//updateItems
POST /ccstore/v1/purchaseLists/gl10001/updateItems
POST /ccagent/v1/purchaseLists/gl10001/updateItems
{
  "items": [
    {
      "op": "add",
      "productId": "Product_18Cxi",
      "quantityDesired": 25,
      "catRefId": "Sku_18Dxi",
      "customPLIProperty_1": "Value for customPLIProperty_1 property
for PurchaseList1's item Product_18Cxi"
    },
    {
      "op": "update",
      "productId": "Product_5Cx",
      "quantityDesired": 30,
      "catRefId": "Sku_5Cxy",
      "customPLIProperty_1": "Value for customPLIProperty_1 property
for PurchaseList1's item Product_5Cx"
    }
  ]
}
```

When you update or create a custom purchase list line item, the custom properties that you create can be included as part of the request payloads from either the storefront or the agent APIs. You can also pass values for custom properties on the purchase list line items in the request payload when you are creating or updating a purchase list. For example:

```
//createPurchaseList
POST /ccstore/v1/purchaseLists
```

```
POST /ccagent/v1/purchaseLists

// updatePurchaseList
PUT /ccstore/v1/purchaseLists/gl10001
PUT /ccagent/v1/purchaseLists/gl10001

{
  "customPLStringProperty_1": "Value for customPLProperty_1 property for
PurchaseList1",
  "accountId": "or-100001",
  "name": "PurchaseList1",
  "description": "PurchaseList",
  "siteId": "siteUS",
  "items": [
    {
      "productId": "Product_18Cxi",
      "quantityDesired": 25,
      "catRefId": "Sku_18Dxi",
      "customPLIProperty_1": "Value for customPLIProperty_1 property for
PurchaseList1's item Product_18Cxi"
    },
    {
      "productId": "Product_5Cx",
      "quantityDesired": 30,
      "catRefId": "Sku_5Cxy",
      "customPLIProperty_1": "Value for customPLIProperty_1 property for
PurchaseList1's item Product_5Cx"
    }
  ]
}
```

Anyone who has edit access to a purchase list can update the value of a custom property on a purchase list line item, although only the owner of a purchase list can update the value of a custom property that resides on the purchase list itself.

Share purchase lists

In Oracle Commerce, you create and maintain purchase lists so that you do not have to look through the catalog for the same products each time you want to purchase them.

You create the purchase list once and, then, whenever you want to order those items, you select the purchase list and add all or some of the items in it to your order. See Understand purchase list features for more complete information on the feature

As either a shopper or an account-based shopper, you can share also purchase lists with other shoppers or account-based contacts. This section of the chapter provides information on understanding this feature and illustrates how to share purchase lists.

Understand the ways that purchase lists are shared

There are two ways that purchase lists can be shared. These are the following:

- As a non-account-based shopper, you share the list with other non-account-based shoppers - With this method, a non-account-based shopper who has created a purchase

list provides one or more email addresses of other non-account-based shoppers with whom they wish to share the list.

- As an account-based shopper, you share the list with other account-based shoppers within the context of the current account - With this method, an account-based shopper who has created a purchase list provides one or more email addresses of recipients within the current account with whom they wish to share the purchase list. The account information refers to an account to which both you and the owner(s) of the email address belong. You also can indicate whether the purchase list should be shared with all shoppers in the current account. If you share a purchase list with an account then all members of the account can see the purchase list. A purchase list is private to the owner unless you explicitly share it with other non-account-based shoppers or account-based shoppers.

Use the REST API to share a purchase list when creating or updating a purchase list. Use the `sharedWithOthers` property by setting it to `true` or `false`. This allows other shoppers to see the purchase list. Use the `siteId` to indicate the site that shares the purchase list. For detailed information on working with the Storefront REST API, refer to the Oracle Commerce REST API.

Understand purchase list owners vs. list recipients

Purchase list owners and the recipients that they share the list(s) with have different capabilities as to how they can use the list.

As a purchase list's owner (creator), you may:

- Share the purchase list with other shoppers, and grant edit permission to recipients.
- Edit the name and description of the purchase list.
- Edit any account/site viewing restrictions on the purchase list (via API only).
- Delete the purchase list.
- See and edit the list of purchase list recipients.
- View the purchase list's "last modified" information. A list owner can always see who last modified the purchase list and when the last modification occurred.

As the recipient of a purchase list you may:

- See the name and email address of the purchase list's owner.
- Modify the quantity of an item in the purchase list.
- If granted edit access, add items to the purchase list or remove items from the purchase list.

Note: Any shopper (non-account-based shopper or account-based shopper) who tries to add an item(s) to an existing purchase list from Product Details or Order Details can only choose from purchase lists that they have created.

- Copy the purchase list and become the owner of the copy so that you can share the copy with other shoppers. Refer to the user interface for a link to copy (and rename) the list.

Recipients of shared purchase lists are viewed and managed differently depending on the type of shopper who creates the list. If you are a non-account-based shopper who creates the purchase list being shared you can do the following with list recipients:

- See a list of all recipients (i.e., email addresses) with whom you shared the purchase list.
- Add a recipient.
- Remove a recipient.
- When adding one or more recipients, you can optionally add a comment that will be included in the email notification sent to recipients. This comment is not saved and is not subsequently viewable.

If you are an account-based shopper who creates the purchase list being shared you can do the following with list recipients:

- See a list of all individual recipients (i.e., email address/account information pairs) with whom you have shared the purchase list.
- In the shopper user interface, you can only see the email addresses of those with whom you have shared the purchase list in the current account context.
- Add an individual recipient.
- Remove an individual recipient.
- See a list of all account recipients with whom you have shared the purchase list.

In the shopper user interface, you can only see whether you have shared the purchase list with the current account.

- Add an account recipient (via API only, for accounts other than the current context).
- Remove an account recipient (via API only, for accounts other than the current context).
- When adding one or more recipients, you can optionally add a comment that will be included in the email notification sent to recipients. This comment is not saved and is not subsequently viewable.

Understand validation of shoppers chosen to receive lists

Commerce validates the following before making a purchase list visible and accessible to a non-account-based shopper or account-based shopper:

- The shopper's email address exists in Commerce.
- The shopper's type (non-account-based shopper or account-based shopper) matches the type of the shopper who created the purchase list.
- The shopper is a recipient of the purchase list.
- With an account-based shopper, that the customer belongs to the account with which the purchase list was shared, or to the account in the email address/account information pair with which the purchase list was shared.
- The shopper's account and site context adhere to any account and/or site context viewing restrictions assigned to the purchase list.

Understand which purchase list types can be viewed

With the functionality of purchase lists being extended so that they can be shared, a shopper (non-account-based shopper or account-based shopper) can see two types of purchase lists:

- My Lists – This is a list of all purchase lists that a shopper owns (and that are available in the current account or site)
- Lists Shared with Me – This is a list of all purchase lists shared with the shopper (and that are available in the current account or site context) including:

  - Lists shared with the owner directly by email address
  - For account-based shoppers only, lists shared with owner at the account level (i.e., by virtue of the owner's membership in an account)

Information that can be seen in the My Lists list includes the following:

- Purchase list name (this is a link that opens the list details)
- An icon indicating whether the purchase list is shared or private
- The number of items contained in the list

Information that can be seen in the Lists Shared with Me list includes the following:

- Purchase list name (this is a link that opens the list details)
- The purchase list owner's name
- The purchase list owner's email address
- The number of items contained in the list

Each list of collected lists is paginated depending on the amount of lists.

Each list also has a filter control that lets you filter on the following fields:

- Purchase list name
- Purchase list owner's name (for Lists Shared with Me)
- Purchase list owner's email address (for Lists Shared with Me)
- Placeholder text for the filter control on My Lists: "Filter by name"
- Placeholder text for the filter control on Lists Shared with Me: "Filter by name, email, or owner"

Note: These are not typeahead filters.

You can also sort the list on the following fields (ascending and descending):

- Purchase list name
- Purchase list owner's name - sorted by last name (for Lists Shared with Me)
- Purchase list owner's email address (for Lists Shared with Me)

Finally, you can switch between the following views in the Lists Shared with Me list:

- All lists shared with you (the owner) in this account (the default)
- Lists shared with all members of this account
- Lists shared with you (the owner) individually

Assign edit access to shared purchase lists

When sharing purchase lists, you have the ability to assign edit access to the list recipients if you are the list creator. Edit permission allows a shared purchase list recipient to add items to the purchase list or remove items from the purchase list. Editing access permissions can differ depending on the type of shopper you are.

A list owner who is a non-account-based shopper can:

- Set a default edit permission (yes or no) for the purchase list. The default for this setting is no edit access.

- Set the edit permission (yes or no) for an individual recipient of the purchase list. The default for this setting is no edit access.

Note: An individual non-account-based shopper recipient's permission, if explicitly set, takes precedence over the purchase list's default permission.

A shared purchase list owner (creator) who is an account-based shopper can

- Set a default edit permission (yes or no) for the purchase list for an account recipient. The default for this setting is no edit access.
- Set the edit (yes or no) for an individual recipient (a member of the current account) of the purchase list – that is using an email address. The default for this setting is no edit access.

Note: The permission for the individual email address, if explicitly set, takes precedence over the purchase list's default permission for the account.

Understand email notifications for recipients of shared lists

Recipients of shared lists receive email notification when they are chosen to share a list. The following occurs in the email notification process for people who are chosen as recipients of shared purchase lists:

- When a purchase list owner adds an individual recipient to share the list, the recipient receives a notification email.
- When a purchase list owner adds an account-based recipient, a single email notification is sent with all current members of the account Bcc'd on the email.
- Emails are scheduled using the scheduler, rather than being sent immediately. See Notify Shoppers When Items are Back in Stock for more information on using the scheduler.

For more complete information on what occurs during the email notification process, the contents of the emails, and how to configure the email settings for shared purchase lists, refer to Configure Email Settings.

Make sure purchase list sharing is available for shoppers

To make sure that list sharing is available for both non-account-based shoppers and account-based shoppers, you just need to make sure that the correct purchase list widgets are used in the store. These widgets are already present in layouts for account-based shoppers, but need to be added for shoppers that are not account based.

Once the purchase list widgets are included in a store's pages, purchase list sharing is automatically available. There is no need actually configure list sharing – you just need to have the correct widgets present.

Note: For additional information on how to use purchase list widgets in your store layouts, refer to Work with purchase lists.

As mentioned, for non-account-based shoppers, you must add the necessary widget to the correct layout from the Design area of the administration user interface to expose the feature (these widgets are already present in layouts for account-based shoppers).

Use the following steps to add the widgets for non-account-based shoppers:

1. Login as the Administrator.
2. Open the **Design** page.
3. Search for the "Profile Layout" on the **Layout** page.

4. Open it for editing in the **Grid View**.
5. Find the “Profile Navigation (shared)” widget and open its **Settings**.
6. Add another row in it by clicking on “**Add More Rows**”.
7. In the first column add “`purchaseListsText`” and in the second column add “`/purchaselists`”.
8. Click on **Save** and publish these changes.

To check and make sure that the introduction of the widget to layout has made the creation of purchase lists available, do the following:

1. Login as to the store as a non-account-based shopper.
2. Click **My Account** at the top of the user interface.
3. On the **My Account Page**, click on **Purchase Lists**. At this page will see the options to:
   - Create a purchase list
   - Share a purchase list after you have created it by providing an email address.
   - Set edit access permissions for the shared list.
   - Copy an existing list for sharing, renaming, etc.
4. Be sure to save any changes or additions made to that page.

Add products to a list from the search box on the Purchase List details page

To have the ability to add products to your list from the search box on the Purchase List details page, use the following steps:

1. Login as the Administrator.
2. Open the **Catalogs** page.
3. Click **Manage Catalogs > Product Types > Base Product**
4. Click **SKU Properties** and click on edit the ID property.
5. Check **Allow** property to be searched.
6. Click on **Save** and **Publish** that change.
7. In the Search area of the administration user interface, go to **Searchable Field Ranking**.
8. Open the Typeahead ranking.
9. Add the SKU ID field.
10. Click on **Save** and **Publish** that change.

You should now be able to use the search box in the Purchase List details page to search for a product by its SKU ID.

Add products to your list from the Product details page

To have the ability to add Products to your purchase list from the Product details page, use the following steps:

1. Login as the Administrator.
2. Open the **Design** page.

3. Search for the **Product Layout**.
4. Open it for edit in the **Grid View**.
5. Look for the **Product Details Widget** and open its **Settings**.
6. From the pull-up menu for **Element Library**, pick the **Add To Purchase List** element (widget) and place it in the blank section above.
7. Click on **Save** and **Publish** these changes.

After having completed these steps, you can add to an existing purchase list or choose to create a new one when you open the product details page for a product. For additional information on how to enable purchase lists refer to Configure purchase lists.

Share a purchase list and specify edits permissions

To share purchase lists and specify edit permissions, required widgets need to conditionally display fields slightly differently for a non-account-based shopper as opposed to an account-based shopper who has created a purchase list. This process can be described as follows:

- If you are a non-account-based shopper who created the purchase list then you can
  - Indicate whether the purchase list should be shared with all shoppers. The default for this global sharing is False.
  - If you wish to share the purchase list with all shoppers, optionally specify a global edit permission (yes or no) for the recipients. The default for the global edit permission is No / False.
  - Provide one or more email addresses of shoppers with whom to share the purchase list.
  - For each email address of a recipient, optionally specify an edit permission (yes or no). This individual edit permission, if present, overrides the global edit permission.
  - When adding one or more recipients, optionally add a comment that will be included in the email notification sent to recipients. This comment is not saved and is not subsequently viewable.
- If you are an account-based shopper who created the purchase list, then you can
  - Indicate whether the purchase list should be shared with all shoppers in the current account. The default for this account-level sharing is False.
  - If you share the purchase list at the account level, optionally specify an account-level edit permission (yes or no). The default for the account-level edit permission is No / False.
  - Provide one or more email addresses of shoppers with whom to share the purchase list. The purchase list will be shared with each of them in the context of Monica's current account context.
  - For each email address of a recipient, optionally specify an edit permission (yes or no). This individual edit permission, if present, overrides the account-level edit permission.
  - When adding one or more recipients, optionally add a comment that will be included in the email notification sent to recipients. This comment is not saved and is not subsequently viewable.

Both non-account-based shoppers and account-based shoppers can see the list of email addresses (and accounts, in an account-based shopper's case) with which they have shared a purchase list.

Both non-account-based shoppers and account-based shoppers can remove an email address (or an account, in an account-based shopper's case) from the list of recipients.

When a non-account-based shopper or an account-based shopper delete a purchase list that is shared, both will get a confirmation dialog that announces that the purchase list is shared and will ask them if they really want to delete it. Cancel and Delete buttons allow them to make either choice.

Understand the purchase list cleanup process

Behind the scenes, Commerce will periodically run a cleanup process that:

- Updates recipient lists for purchase lists owned by shoppers whose account memberships have changed
- Customers can configure how often the process runs.

See Work with the purchase list API for more information on removing purchase lists.

19

Enable Order Approvals

The changes you make to a storefront for order approvals make it possible for an administrator to enable or disable order approvals, set a purchase limit, and designate approvers on the storefront itself.

They also add the necessary UI controls for shoppers and approvers that are working with orders that require approvals; for example, controls for viewing orders that need approval, approving or reject the orders, and providing payment for orders that have been approved.

This section describes the necessary changes you must make to a storefront to incorporate order approvals.

Note: This chapter has a companion chapter, Use Order Approvals, that provides an overview of order approvals and describes other non-developer tasks for the feature.

Allow a delegated administrator to control order approvals

If you want delegated administrators of an account to be able to enable and disable order approvals and set a purchase limit on the storefront, you must set a flag in the administration interface.

Once you set this flag, the corresponding settings in the administration interface become read-only and only the delegated administrator is allowed to manage these settings.

An account's delegated administrator approval management is site-specific. If an account has multiple contracts, you must select a site when setting the delegate approval management flag.

Note: If you integrate with an external system determine if orders require approval, the account's delegated administrators will not be able to enable and disable order approvals or set a purchase limit. See Integrate with an external system for order approvals for more information.

To allow delegated administrators to control order approvals:

1. In the administration interface, click the **Accounts** tab.
2. Select the account to be modified.
3. Click the **Approvals** tab.
4. If you are using multiple sites, select the site that will be associated with the approval.
5. Enable the Administrator at the account can manage approvals option and save your changes.

Configure a deferred payment gateway for order approvals

If you want your shoppers to be able to pay for orders requiring approval using a deferred payment method such as invoice or cash, you must set up the payment gateway for the method using the instructions provided in the Integrate with an Invoice Payment Gateway or Integrate with a Cash Payment Gateway sections.

In addition to these instructions, you must add an `enabledForApproval` property to the `config.json` file for the gateway. For example, the following code snippet shows the `config.json` file for an invoice payment gateway that includes the `enabledForApproval` property:

```
{
  "configType": "payment",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "instances" : [
    {
      "id": "agent",
      "instanceName": "agent",
      "labelResourceId": "agentInstanceLabel"
    },
    {
      "id": "preview",
      "instanceName": "preview",
      "labelResourceId": "previewInstanceLabel"
    },
    {
      "id": "storefront",
      "instanceName": "storefront",
      "labelResourceId": "storefrontInstanceLabel"
    }
  ],
  "properties": [
  {
      "id": "enabledForApproval",
      "type": "booleanType",
      "name": "enabledForApproval",
      "helpTextResourceId": "enabledForApprovalHelpText",
      "labelResourceId": "enabledForApprovalLabel",
      "public": true,
      "defaultValue": true
  },

  {
      "id": "paymentMethodTypes",
      "type": "multiSelectOptionType",
      "name": "paymentMethodTypes",
      "required": true,
      "helpTextResourceId": "paymentMethodsHelpText",
      "labelResourceId": "paymentMethodsLabel",
      "defaultValue": "invoice",
      "displayAsCheckboxes": true,
      "options": [
         {
           "id": "invoice",
           "value": "invoice",
           "labelResourceId": "invoiceLabel"
         },
         {
           "id": "card",
           "value": "card",
```

```
            "labelResourceId": "cardLabel"
          }
        ]
      }
  ]
}
```

You must also add a translation resource that provides the label for the `enabledForApproval` property in the administration interface. You should add the resource to each `<locale>.json` file you have for the languages supported in your administration interface. For example, the following code snippet shows the `en.json` file where the value for the `enableForApprovalLabel` is Enable for order approvals. This is the label that will appear in the administration interface when you view the gateway's properties.

```
{
  "resources" : {
    "paymentMethodsHelpText": "Select the payment methods.",
    "paymentMethodsLabel": "Payment Methods",
    "invoiceLabel": "Invoice",
    "cardLabel": "CCCard",
    "title": "Invoice Payment Gateway Config",
    "description":"Invoice Payment Gateway configuration.",
    "agentInstanceLabel": "Agent Configuration",
    "previewInstanceLabel": "Preview Configuration",
    "storefrontInstanceLabel": "Storefront Configuration",
    "poRequiredHelpText":"Check if PO number is required",
    "poRequiredLabel":"PO number required",
    "enabledForApprovalLabel":"Enable for order approvals",
    }
}
```

When configuring the deferred payment gateway in the administration interface, be sure that the "Enable for order approvals" option is checked (if you follow the instructions above, it will be checked by default).

Set the frequency of canceled order clean up

A service runs periodically to review the order repository and remove any orders that have been marked for cancellation because they exceeded the price hold period time limit.

To set the initial frequency of the order cancellation service, you issue a POST request to the `scheduledJobs` endpoint, with a payload that defines the path to the `CancelOrderScheduledJob` component and the schedule, an example of which is provided below. To update the schedule, you issue a PUT request to the same endpoint.

```
POST /ccadmin/v1/merchant/scheduledJobs

{
"componentPath": "CancelOrderScheduledJob",
"scheduleType": "periodic",
"schedule":
  {
   "period" : 1000000
```

```
  }
}
```

The `scheduleType` and schedule properties determine the frequency used when running the service. Setting these properties is described in detail in the Configure the scheduled order service section.

See Set a price hold period for more information on the price hold period.

Configure page layouts for order approvals

You must make changes to a number of the storefront layouts to add order approvals to your storefront.

Oracle recommends that you clone the out-of-the-box layouts and then make your changes to the clones. If your site only supports account-based shoppers, you can mark the clones as the defaults and make the order approval changes to those pages. If your site must support both account-based shoppers and other, non-account affiliated shoppers, then you will need two versions of the pages, one marked as default for the non-account affiliated shoppers and the other marked as “Display layout to account shoppers only” for the account-based shoppers. In this scenario, you would make the order approval changes to the pages designed for the account-based shoppers.

The modifications described in the sections below involve adding new widgets to page layouts and also making sure the latest versions are used for some widgets that are included in the page layouts out of the box. To replace a widget with the latest version, see Upgrade deployed widgets in Customize your store layouts.

Profile layout for order approvals

To add the order approval feature to your storefront, you must add these two widgets the Profile layout:

- The Order Approval Settings widget provides the delegated administrator with an interface for enabling and disabling order approvals and setting the purchase limit.

If you integrate with an external system determine if orders require approval, the account’s delegated administrators will not be able to enable and disable order approvals or set a purchase limit. See Integrate with an external system for order approvals for more information.

- The Orders Pending Approval widget allows an approver to view a list of orders pending approval.

To add the order approval widgets to the Profile layout, you can create a vertical tab stack and place the order approval widgets on individual tabs within the stack. To restrict the display of the Order Approval Settings widget to contacts with administrator privileges and the display of the Orders Pending Approval tab to contacts with approver privileges, you can add something similar to the following code snippet in the vertical tab stack’s template:

```
<!-- ko foreach: regions -->
<!--ko if:($data.displayName() == 'Profile') ||
((($data.displayName() == 'Account Contacts') ||
($data.displayName() == 'Account Addresses') ||
($data.displayName() == 'Order Approval Settings')) &&
```

```
($masterViewModel.data.global.user.roles.map(function(data)
{return data.function; }).indexOf("admin")!== -1)) ||
(($data.displayName() == 'Orders Pending Approval') &&
($masterViewModel.data.global.user.roles.map(function(data)
{return data.function; }).indexOf("approver")!== -1))-->
   <li role="presentation" data-bind="css: {active: $index() === 0},
        attr: { id: 'verticalTabs-'+$parent.id()+'-tab-'+$index() }">
      <a data-toggle="tab" data-bind="attr: { 'href': '#verticalTabs-' +
           $parent.id() + '-content-' + $index()}">
         <span data-bind="text: displayName"></span>
      </a>
   </li>
<!-- /ko -->
<!-- /ko -->
```

This code snippet shows the My Profile tab to all contacts but restricts the display of the Orders Pending Approval tab to approvers and the display of the other tabs (Account Contacts, Account Addresses, and Order Approval Settings) to administrators. (It assumes you used “My Profile”, “Account Contacts”, “Account Addresses”, “Order Approval Settings”, and “Orders Pending Approval” as the display names for the tabs that hold the Customer Profile, Account Contacts, Account Addresses, Order Approval Settings, and Orders Pending Approval widgets, respectively.)

In addition to adding the Order Approval Settings and Orders Pending Approval widgets to the Profile layout, you must also make sure you are using the latest version of the Account Contacts widget. This version allows the delegated administrator to assign the Approver role to contacts.

Note: The Account Addresses and Account Contacts widgets are described in the Add delegated administration to your storefront section. For more information on vertical tab stacks, see Add Vertical Tabs in Customize your store layouts and Use Stacks for Increased Widget Layout Control.

Order Details layout and Scheduled Order layout for order approvals

These two layouts provide detailed information about orders, both regular (Order Details layout) and scheduled (Scheduled Order layout). The modifications you have to make to them are similar and are described below.

You have to add the Order Approvals widget to each layout. This widget allows an approver to approve or reject an order and provide comments when viewing an order’s details. Note that Order Approvals widget only appears in the storefront when an approver is viewing an order’s details. Otherwise, it is hidden.

You also must use the latest version of these widgets:

- The Order Details widget or the Order Details with Additional Info widget on the Order Details layout.
- The Scheduled Order widget on the Scheduled Order layout.

The updated versions of these widgets provide:

- The reasons an order requires approval. If you are using the Order Approvals webhook to integrate with an external system, all the reasons returned by the webhook response are listed.

- The approver name and comments, along with the rest of an order's details, after the order has been approved or rejected.
- A Complete Payment button for the shopper if an order has been approved but still requires payment.
- An Add Items to Cart button for orders that have been rejected. This button allows the shopper to quickly add the items in the rejected order to a new cart, forming the basis for a modified order that will pass approval.

Order Confirmation layout for order approvals

To incorporate order approvals on the Order Confirmation layout, make sure you are using the latest version of either the Order Confirmation widget or the Order Confirmation with Additional Info widget. The latest versions of these widgets include conditional text for two cases:

- If the order is pending approval, a message describing the reasons the order requires approval is provided. If you are using the Order Approvals webhook to integrate with an external system, all the reasons returned by the webhook response are listed.
- If the order will need payment after approval (in other words, the payment method used for the order is not a deferred payment method like invoice or cash), this message is provided: "After approval, you will need to provide payment information."

Order History layout for order approvals

To incorporate order approvals on the Order History layout, make sure you are using the latest version of the Order History widget. This version allows a contact to see orders that are pending approval, approved, rejected, or canceled, along with orders that did not require approval. Note that the text strings used to communicate the order statuses can be customized via the Text Snippets tool.

Checkout layout for order approvals

The checkout layout requires more customization to support order approvals than the other page layouts. As such, it is described in its own section, Manage the checkout flow for orders requiring approval.

Manage the checkout flow for orders requiring approval

Commerce cannot store payment details such as credit card information in between when an order is placed and when the order is approved.

As such, storefronts that use the order approvals feature must implement two checkout layouts, one for the initial checkout flow and a second for when payment is provided after approval has been given. The initial checkout flow layout handles orders in the following way:

- If an order does not require approval, the shopper can pay for it immediately using any payment method she chooses to provide.
- If the order requires approval, the shopper can either:
  - Pay for it immediately with a deferred payment method like invoice or cash that does not require storing payment details. Orders that fall into this category

are immediately processed after approval is given and require no further interaction by the shopper.

- Opt to pay for the order after approval is given, using a non-deferred payment method like a credit card or gift card. The shopper is notified via email after order approval is given and must return to the store to provide payment information. This is when the shopper is presented with the post-approval checkout layout.

Because order approval is based on aspects of the order (such as the order total or items the order contains) at the time the shopper submitted the order, the post-approval checkout layout must restrict the shopper from editing the order in any way other than providing payment information. To pay for an approved order, the shopper must view the order's details, either by clicking a link in the Order Approved notification email or by viewing her order history and clicking the order that is pending payment. When a shopper is viewing order details for an approved order that is pending payment, a Complete Payment button is provided. Clicking this button sends the shopper to the post-approval checkout layout, where she can provide the payment information.

The following sections describe how to create the two checkout layouts that support order approvals.

Note: The sections below provide the minimum version number for each of the widgets you will be placing on the checkout layout. In order for the order approvals feature to work as described in this section, you must use these minimum versions or later. To tell which version of a widget you are using, view the widget's settings and click the About tab to see the widget's version number.

Initial checkout flow for order approvals

Follow the instructions below to create the initial checkout flow for storefronts that use order approvals.

To create the initial checkout flow:

1. On the **Design** tab, clone the **Checkout Layout** and give it a descriptive name like **Checkout**, **Order Approval**, **Immediate Payment**.
2. Enable the **Display** layout to account shoppers only option and save the clone.
3. Go to **Grid View** for the **Checkout**, **Order Approval**, **Immediate Payment** layout.
4. The rows containing the Notifications Widget, Header – Basic Widget, and Footer Widget widget instances should remain as is. Remove all widgets from the row in between them (Login – Checkout, Customer Address Book, Payment Details, and so on) and drag the column separator to the right to reconfigure the row to have a single column.
5. Add a **Progress Tracker** stack to the empty row you just created.
6. Edit the **Progress Tracker** so that it has the following tabs:
   - **Login**
   - **Schedule Order** (this tab is not required if your storefront does not include the scheduled orders feature)
   - **Shipping and Promotions**
   - **Billing and Payments**
7. On the **Login** tab, add the Login – Checkout widget (version 2 or later).
8. If you created a **Schedule Order** tab, add the Scheduled Order – Checkout widget to it (version 2 or later).

9. On the **Shipping and Promotions** tab, add new instances of the following widgets and modify them as described:
    - Promotion (version 1 or later).
    - Managed Account Address Book (version 3 or later). Name the instance Managed Account Address Book, Shipping Only. View the widget's settings and disable the Include Billing Details option.
    - Cart Summary (version 5 or later).
    - Order Summary - Checkout (version 9 or later). Follow the instructions in Modify the Order Summary – Checkout widget to edit the widget to hide the Place Order button and enable the Shipping Method menu.
    - Check for Approval Required. This is a custom widget that you have to create yourself. See Create the Check for Approval Required widget for details on how to do so.
10. On the **Billing and Payments** tab, add new instances of the following widgets and modify them as described:
    - Managed Account Address Book (version 3 or later). Name the instance Managed Account Address Book, Billing Only. View the widget's settings and disable the Include Shipping Details option.
    - Payment Gateway Options (version 1 or later). Modify the instance to include elements for any deferred payment methods your storefront supports for orders that require approval, such as Invoice Payment and Cash Payment. Make sure to configure the payment gateway for any payment methods you add here. For more information, see Configure a deferred payment gateway for order approvals.

    Note: If an order requires approval and a payment gateway is not configured for order approvals (that is, the `enableForApproval` flag has not been set to true for the gateway) then the payment method associated with that gateway will be hidden and disabled.
    - Pay After Approval (version 1 or later). This widget provides shoppers with a checkbox that allows them to specify that they will pay for the order after approval has been given.
    - Payment Details (version 6 or later). This version of the widget is hidden if the order requires approval.
    - Gift Card Widget (version 3 or later). This version of the widget is hidden if the order requires approval.
    - Cart Summary (version 5 or later).
    - Order Summary - Checkout (version 9 or later). Follow the instructions in Modify the Order Summary – Checkout widget to edit the widget to display the Place Order button and disable the Shipping Method menu.

    In addition to the new widget instances you just added to the Billing and Payments tab, you must also add an instance of the Cart Summary widget (version 5 or later). For this widget, use the same instance you created and placed on the Shipping and Promotions tab.

Checkout flow for payment after approval

Follow the instructions below to create a layout for the checkout flow that is used when orders are paid for after approval has been given.

To create the delayed payment checkout flow:

1. On the **Design** tab, clone the **Checkout Layout** and give it a descriptive name like **Checkout**, **Order Approval**, **Delayed Payment**.
2. Enable the **Display** layout to account shoppers only option.
3. Set the layout to be displayed when the **Order Status** is **PENDING_PAYMENT**. If your storefront uses the scheduled orders feature, also set the layout to be displayed with the **Order Status** is **PENDING_PAYMENT_TEMPLATE**.
4. Save the clone.
5. Go to **Grid View** for the Checkout, Order Approval, Delayed Payment layout.
6. The rows containing the Notifications Widget, Header – Basic Widget, and Footer Widget widget instances should remain as is. Remove all widgets from the row in between them (Login – Checkout, Customer Address Book, Payment Details, and so on).
7. Add a new instance of the Managed Account Address Book widget (version 3 or later) to the empty row you just created. Name the instance Managed Account Address Book, Delayed Payment. View the widget's settings and ensure that both Include Billing Details and Include Shipping Details are enabled. Note that this version of the widget will disable editing of the shipping address if the order state is PENDING_PAYMENT.
8. Add the next four widgets to the row. Use the same instances you created for the Billing and Payments tab of the immediate payment flow.
   - Payment Gateway Options (version 1 or later).
   - Payment Details (version 6 or later)
   - Gift Card Widget (version 3 or later)
   - Cart Summary (version 5 or later).
9. Add a new instance of the Order Summary - Checkout widget (version 9 or later) to the row.

Modify the Order Summary – Checkout widget

The HTML for the Order Summary – Checkout widget instances on both the Shipping and Promotions tab and the Billing and Payments tab must be modified to support the two order approval-related checkout layouts.

For the Shipping and Promotions tab

Edit the instance of the Order Summary – Checkout widget that resides on the Shipping and Promotions tab to remove the following code from the widget's HTML template:

```
<!-- ko ifnot : (order().approvalRequired()) -->
<div class="paymentoptions hidden-xs">
  <h3 data-bind="widgetLocaleText:'paymentOptionsText'"></h3>
  <div class="row-payments">
    <!-- ko foreach: payment().cards -->
    <span data-bind="css : ($index() % 4) == 0 ? 'row-first' : '' ,
       attr:{id: 'CC-checkoutOrderSummary-payment'+$parents[1].id()+value}">
    <img data-bind="attr:{src: img}" alt=""/>
    </span>
    <!-- /ko -->
  </div>
</div>
```

```
<!-- /ko -->
<!-- ko ifnot : (order().showSchedule) -->
<div id="CC-checkoutOrderSummary-placeOrder" class="checkout row">
<button class="cc-button-primary col-xs-12" data-bind="click:
handleCreateOrder,
enable: order().enableOrderButton">
<span data-bind="widgetLocaleText:'placeOrderText'"></span></button></
div>
<!-- /ko -->
<!-- ko if : (order().showSchedule) -->
<div id="CC-checkoutOrderSummary-placeOrder" class="checkout row">
<button class="cc-button-primary col-xs-12" data-bind="click:
handleCreateOrder,
enable: order().enableOrderButton">
<span data-bind="widgetLocaleText:'scheduleOrderText'"></span></
button></div>
<!-- /ko -->
<p><span data-bind="widgetLocaleText:'paymentMessage'"></span></p>
<!-- ko if : $data.payment().gateways.paypalGateway.enabled -->
<!-- ko ifnot : (order().approvalRequired()) -->
<!-- ko ifnot : (order().isPaypalVerified()) -->
<div id="CC-checkoutOrderSummary-paypal" class="checkout row">
  <!-- ko if: (order().showSchedule() &&
       !order().paymentDetails().isPaypalEnabledForScheduledOrder()) --
>
  <span id="CC-checkoutOrderSummary-paymentAvailablability"
    data-bind="widgetLocaleText: 'paymentMethodNotAvilable'"></
span><br>
  <img class="img-responsive center-block" alt="checkoutWithPayPal"
    data-bind="attr: {src: paypalImageSrc}">
  <!-- /ko -->
  <!-- ko ifnot: (order().showSchedule() &&
       !order().paymentDetails().isPaypalEnabledForScheduledOrder()) --
>
  <a data-bind="attr : { id: 'CC-checkoutOrderSummary-
checkoutWithPaypal'},
    disabled: {condition: cart().items().length == 0,
    click: order().handleCheckoutWithPaypal.bind(order()) }" href="#">
  <img class="img-responsive center-block" alt="checkoutWithPayPal"
    data-bind="attr: {src: paypalImageSrc}">
  </a>
  <!-- /ko -->
</div>
<!-- /ko -->
<!-- /ko -->
<!-- /ko -->
```

For the Billing and Payments tab

Edit the instance of the Order Summary – Checkout widget that resides on the Billing and Payments tab to replace this portion of the widget's HTML template:

```
<button id="cc-shippingOptions-dropDown" class="btn dropdown-toggle
col-xs-12"
data-toggle="dropdown" tabindex="0" data-bind="click:
```

```
displayShippingMethodsDropdown,disable: !order().isOrderEditable(),
attr: {'aria-label': ''}" style="border-color:#ddd;background-color:white;">
```

With the following code:

```
<button id="cc-shippingOptions-dropDown" class="btn dropdown-toggle col-
xs-12"
        data-toggle="dropdown" tabindex="0" data-bind="click:
        displayShippingMethodsDropdown,disable: true, attr: {'aria-label':
''}"
        style="border-color:#ddd;background-color:white;">
```

Create the Check for Approval Required widget

The checkout flow for order approvals requires a custom widget that determines whether or not an order requires approval. This widget manages what the shopper sees for billing and payment options, depending on whether or not an order requires approval. For example, if the order requires approval, the shopper will not be able to pay for the order using a credit card. This widget does not have any UI associated with it, only the logic for determining if the order requires approval.

For general information on creating and uploading a custom widget, refer to Create a Widget. The code snippets below show you the custom code that must exist in the widget.

The following example shows the JavaScript for the Check for Approval Required widget:

```
/**
 * @fileoverview Check for Approval Require Widget.
 *
 * @author
 */
define(
  //-------------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------------
  ['knockout', 'pubsub', 'notifier', 'CCi18n', 'ccConstants', 'navigation',
   'ccRestClient'],
  //-------------------------------------------------------------------------
  // MODULE DEFINITION
  //-------------------------------------------------------------------------
  function(ko, pubsub, notifier, CCi18n, CCConstants, navigation,
ccRestClient) {
    "use strict";
  return {

    onLoad: function(widget){
      var pageId=widget.pageContext().pageType.id;
    },
     validate : function() {
         var orderId=this.user().orderId();
         var data = {
             "orderId":orderId
         };
         this.order().checkOrderForApproval(data);
       return true;
```

```
    }
  }
});
```

The following example shows the content of the HTML template for the Check for Approval Required widget. Note that, because there is no UI associated with this widget, it contains only a placeholder element:

```
<div style="display:none"></div>
```

Display a contact's purchase limit in a widget

A contact's purchase limit is available from the User view model; however, it is not included in the out-of-the-box widgets.

If you want to add a contact's purchase limit to a widget, you can do so using code similar to the sample below:

```
widget.user().derivedOrderPriceLimit()
```

Integrate with an external system for order approvals

The built-in Commerce approval functionality determines if an order requires approval based on a purchase limit you specify.

If you want to create more complex rules than a simple purchase limit, you can integrate with an external system that determines if an order requires approval. For example, you might want to require approval for all orders that include specific items or that are shipped to certain addresses. To integrate with an external system, you must enable approvals for an account, and configure the Order Approvals webhook.

Enable or disable order approvals

Order approval settings are defined at the account level and apply to all contacts within the account. When you integrate with an external system, the order approval feature can be enabled or disabled only in the administration interface, not on the storefront.

To enable order approval and specify that an external system should determine if orders require approval:

1. Click the **Accounts** icon.
2. Select the account to modify and click its **Approvals** tab.
3. If you are using multiple sites, select the name of the site. (Approvals are site-specific.)
4. Under **Approval Settings**, select **Require Approval**, then select Use external service to determine approval settings.
5. Click Save.

Configure the Order Approvals webhook

When a contact places an order and their account is configured to use an external service to determine whether approval is required, the server first invokes the Order Approvals function webhook. The webhook sends the following data to the external system:

- Details about the order. The request does not include certain payment details, such as credit card information. See Understand webhooks and PCI DSS compliance for information about payment details that are excluded from the request. See Order Submit request example for a sample JSON representation of an order in a webhook body.
- Shopper profile details for the contact who placed the order.
- Details about the account for which the order was placed.

To send this data to the external system, you configure the webhook by specifying the URL, username, and password for accessing the system. (See Configure webhooks for details.) You must also configure the external system to read the request data, determine whether the order requires approval, and send a response that includes the following items:

- The key `approvalAction`, whose value must be either true (the order requires approval) or false (the order does not require approval).
- The key `approvalActionReason`, whose value is a string that describes the reason approval is required if `approvalAction` is true. Commerce adds this string to the order's properties and displays it in layouts and emails related to orders and approvals. See Configure page layouts for order approvals for details about layouts where this string can appear. See Notify users of order approval-related events for details about emails where this string can appear.

If `approvalAction` is true but `approvalActionReason` is missing, empty, or contains a null string, Commerce uses the string Reason unavailable.

For example, if the order requires approval because some of the products ordered are part of a specific collection, the response body might be:

```
{
    "approvalAction": true,
    "approvalActionReason": "Contains restricted items"
}
```

If Commerce cannot connect to the external system, for example in the event of an outage, or if the `approvalAction` key is missing, null, or contains an invalid value, the order is sent for approval. Approvers are notified, via the Order Pending For Approval email, that the order requires approval because the external system could not be reached.

20

Assign Catalogs and Price Groups to Shoppers

By default, Commerce assigns catalogs and price groups to sites or, for account-based commerce, to accounts.

However, you might want to override these default assignments with different catalogs and price groups for each registered shopper. For example, you can personalize the catalog and prices a shopper sees based on geographic location or level in a loyalty program.

To enable this, Oracle Commerce includes tools that you can use to build a custom integration with an external system that determines which catalog and price groups to assign to a shopper. You create a custom widget that makes a call to the external system to obtain the catalog and price lists to use. You configure a webhook that the widget invokes, sending information about the shopper's context to the external system. The webhook also calls the external system to validate the order when it is placed.

Configure the External Price Group and Catalog webhook

The External Price Group and Catalog webhook sends information to an external system which determines which catalog and price groups to display to a shopper and sends that information in a response.

If Commerce cannot connect to the external system (for example in the event of an outage), or if the webhook response contains incorrect information, Commerce uses the catalog and price groups assigned to the current site or account.

The custom widget you create invokes the webhook to send shopper-context information to the external system, which responds with the catalog and price groups to display. Then, when the shopper places the order (or when a scheduled order is triggered, or an approver opens an order for approval), the server invokes the webhook, which sends the context data to the external system to verify that the catalog and price group used to place the order are still valid. This step is used to ensure that items in the cart are still available and that the prices in the cart have not been modified.

If the webhook response returns a different catalog and price group than the ones that were assigned when the shopper created the order, or if the default catalog and price group are used because the webhook fails, the server verifies that all items in the order are still available at the same price. If they are, the order progresses to the next step.

If all the items in the order are not available at the same prices, an error message is displayed and the shopper can change items in the cart before submitting the order again. For scheduled orders, the order fails. For an order that has been opened by an approver, an error message is displayed and the order is returned to the Pending Approval list.

Note: If your store supports scheduled orders and order approvals, make sure you are using the latest versions of the layouts and widgets described in Create Scheduled Orders and Enable Order Approvals.

The following properties are sent in the JSON request body of the of the External Price Group and Catalog webhook:

- Profile details for the shopper. The webhook sends all the properties (including custom properties) that are sent with the Shopper Account Update webhook.

Note: The webhook does not send any information about the shopper's audience membership.

- If the store is configured for account-based commerce, details about the current account. The webhook sends all the properties (including custom properties) that are sent with the Account Update webhook.
- If the webhook is triggered because an order is placed, a scheduled order is instantiated, or an order is approved, the webhook body also includes details about the order. See Order Submit request example for a sample JSON representation of an order in a webhook body.

The webhook body does not include certain payment details, such as credit card information. See Understand webhooks and PCI DSS compliance for information about payment details that are excluded from the request.

- If any custom order properties have their `externalShopperContext` attribute set to true, the webhook body also includes those order properties as a map of name/value pairs. See Create the custom order properties for more information.

For example, the following portion of a sample request body sends the values for the custom properties eventProp1Key and eventProp2Key.

```
"contextData":{

"userContext":"{\"eventProp1Key\":\"eventasas1\",\"eventProp2Key\":\"tr
ue\"}"
    }
```

To send this data to the external system, you configure the webhook by specifying the URL, username, and password for accessing the system. (See Configure webhooks for details.) You must also configure the external system to read the request data, determine which catalog and price groups Commerce should display, and send a response. The following table describes the properties that should be returned in the JSON response body of the External Price Group and Catalog webhook.

Property	Description
responseCode	An integer that specifies which catalog and price groups to use. When the webhook is triggered during a context change, 0 specifies that Commerce displays the catalog and price groups assigned to the current site or, for account-based commerce, to the current account. If the value is 1, Commerce displays the catalog and price groups whose IDs are returned in the body of the response. When an order is validated, 0 specifies that the catalog and price groups assigned to the current site or account are valid for the order. If the value is 1, the catalog and price groups assigned to the shopper are valid for the order. If the value is 2, the items or prices in the order are not valid. If the response code is 0 or 1, the order is submitted or opened for approval. If the response code is 2, the order is not submitted, and an error is displayed.
Message	A string that describes the responseCode.
defaultPriceListGroup	A string that is the ID of the price group whose prices are displayed by default.
defaultAdditionalPriceListGroups	An array of strings that are the IDs of additional price groups. Additional price groups let a shopper select from a list of currencies your store supports and see those prices on your store.
defaultCatalog	A string that is the ID of the catalog to use.

For example, if the external system determines that a shopper should see the North American Parts (NAParts) catalog priced with the `defaultPriceGroup` with two additional price groups (plgCAD and plgMXN), the response body might be:

```
{
  "defaultAdditionalPriceListGroups": [
    "plgInr",
    "plgEuro"
  ],
  "defaultPriceListGroup": "defaultPriceGroup",
  "defaultCatalog": "NAParts",
  "message": "use this data",
  "responseCode": 1
}
```

Create a custom shopper context widget

This section describes how to create a sample widget that lets a logged-in shopper select a different catalog and price groups set on a storefront.

The widget makes an endpoint call to get a set of custom order properties. The values of these properties will be used to display the UI for a context selector that a shopper uses to select a catalog and price list groups. The custom widget's JavaScript file extends the storefront's `ContextViewModel` class by implementing a callback function. When a shopper successfully switches context, this function invokes the External Price Group and Catalog

webhook, which makes a call to the external system to obtain the catalog and price groups to use.

Create the custom order properties

A settable attribute of order type properties, `externalShopperContext` lets you specify if the properties and their values are sent as name/value pairs in the webhook request body. To add custom properties to an order type, issue a `PUT` request to the `/ccadmin/v1/orderTypes/{id}` endpoint on the administration server. (See Add custom properties to an order type for more information.)

The following example shows a sample request that adds the `defaultPriceListGroup`, `defaultCatalog`, `defaultAdditionalPriceListGroups`, `responseCode`, and message properties to the order type. Note that the `externalShopperContext` attribute is set to true for these properties.

```
{
   "properties":{
      "defaultPriceListGroup":{
         "dimension":true,
         "multiSelect":true,
         "textSearchable":false,
         "default":"",
         "internalOnly":false,
         "localizable":false,
         "label":"defaultPriceListGroup",
         "type":"shortText",
         "uiEditorType":"shortText",
         "required":false,
         "searchable":false,
         "audienceVisibility":false,
         "externalShopperContext":true
      },
      "defaultCatalog":{
         "dimension":true,
         "multiSelect":true,
         "textSearchable":false,
         "default":"",
         "internalOnly":false,
         "localizable":false,
         "label":"defaultCatalog",
         "type":"shortText",
         "uiEditorType":"shortText",
         "required":false,
         "searchable":false,
         "audienceVisibility":false,
         "externalShopperContext":true
      },
      "responseCode":{
         "dimension":true,
         "multiSelect":true,
         "textSearchable":false,
         "default":"",
         "internalOnly":false,
         "localizable":false,
```

```
            "label":"responseCode",
            "type":"shortText",
            "uiEditorType":"shortText",
            "required":false,
            "searchable":false,
            "audienceVisibility":false,
            "externalShopperContext":true
         },
         "message":{
            "dimension":true,
            "multiSelect":true,
            "textSearchable":false,
            "default":"",
            "internalOnly":false,
            "localizable":false,
            "label":"message",
            "type":"shortText",
            "uiEditorType":"shortText",
            "required":false,
            "searchable":false,
            "audienceVisibility":false,
            "externalShopperContext":true
         },
         "defaultAdditionalPriceListGroups":{
            "dimension":true,
            "multiSelect":true,
            "textSearchable":false,
            "default":"",
            "internalOnly":false,
            "localizable":false,
            "label":"defaultAdditionalPriceListGroups",
            "type":"shortText",
            "uiEditorType":"shortText",
            "required":false,
            "searchable":false,
            "audienceVisibility":false,
            "externalShopperContext":true
         }
      }
}
```

Create the shopper context widget extension

The `ext.json` file contains metadata for the extension. For example:

```
{
  "extensionID": "c2e6a60e-579a-4190-af3e-5edc0cd8a725",
  "developerID": "999999",
  "createdBy": "Demo Corp.",
  "name": "ShopperContextSelectorDemoWidget",
  "version": 1,
  "timeCreated": "2017-10-10",
  "description": "Demo Shopper Context Widget"
}
```

Note that the `extensionID` must match the value generated on the Extensions page in the administration interface. See Install the widget for more information.

Write the widget JavaScript

The following example shows the JavaScript for the sample Shopper Context Selector widget.

For general information on creating and uploading a custom widget, refer to Create a Widget.

```
/**
 * @fileoverview Shopper Context Selector.
 */
define(

  //-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
  ['knockout', 'pubsub', 'ccConstants','notifier', 'CCi18n',
'storageApi',
 'viewModels/shopperContext'],

  //-------------------------------------------------------------------
  // MODULE DEFINITION
  //-------------------------------------------------------------------

  function(ko, pubsub, CCConstants, notifier, CCi18n, storageApi,
ShopperContext) {
    "use strict";
    return {
      WIDGET_ID:
        "shopperContextSelector",
      isReady: ko.observable(false),
      onLoad: function(widget) {
        var self = this;
        widget.shopperContextViewModel = ko.observable();
        widget.shopperContextViewModel(ShopperContext.getInstance());
        widget.shopperContextViewModel().
           getOrderDynamicPropertiesWithDefaultValues();
        $.Topic(pubsub.topicNames.USER_LOGIN_SUCCESSFUL).subscribe(func
tion(){
          widget.shopperContextViewModel().populatePLGandCatalogData();
             });
        $.Topic(pubsub.topicNames.USER_LOGOUT_SUCCESSFUL).subscribe(fun
ction(){
          widget.isReady(false);
          window.location.reload();
        });
      },
      beforeAppear:function(page) {
        var widget = this;
        if (widget.user().loggedIn() != false) {
          widget.isReady(true);
        }else{
```

```
          widget.isReady(false);
        }
      },
   // Click handler for the Load Context button
      handleLoadContext: function (viewModel, event) {
        var widget = this;
        widget.shopperContextViewModel().populatePLGandCatalogData();
      },
     };
   }
);
```

Display the context switcher

The custom widget's `display.template` file displays the context switcher that a shopper uses to pick a new context that will invoke the webhook.

```
<!-- ko if: (isReady) -->
<div class="container-fluid">
  <!-- ko if: $data.shopperContextViewModel().dynamicProperties().length > 0
-->
  <div class="row"  data-bind="foreach:
$data.shopperContextViewModel().dynamicProperties">
    <!-- ko if: (type === 'shortText' || type==='richText' ||
type==='number') -->
    <div class="col-md-12">
      <label data-bind="text: label"></label>
      <input type="text" data-
bind="value:$parent.shopperContextViewModel().shopperContext[id],
attr:{'id':id}"/>
    </div>
    <!-- /ko -->
    <!-- ko if: (type === 'checkbox') -->
    <div class="col-md-12">
      <label data-bind="text: label"></label>
      <input type="checkbox" data-
bind="checked:$parent.shopperContextViewModel().shopperContext[id],
attr:{'id':id}"/>
    </div>
    <!-- /ko -->
    <!-- ko if: (type == 'date') -->
    <div class="col-md-12">
      <label data-bind="text: label"></label>
      <input type="date" data-
bind="checked:$parent.shopperContextViewModel().shopperContext[id],
attr:{'id':id}"/>
    </div>
    <!-- /ko -->
  </div>
  <button data-bind="click: handleLoadContext"> <span data-bind="text:'Load
Context'"></span></button>
  <!-- /ko -->
</div>
<!-- /ko -->
```

Install the widget

To install the widget, perform the following tasks in the administration interface:

1. Click the **Settings** icon.
2. Click **Extensions** and display the **Developer** tab.
3. Click **Generate ID** to generate an extension ID for the widget.
4. Edit the widget's `ext.json` file and set the `extensionID` property to the value generated in the previous step.
5. Package the widget as a ZIP file. Use the structure described in Create the widget structure.
6. Display the Installed tab and click **Upload Extension**. Select the ZIP file.
7. Publish your changes.

21

Implement Storefront Single Sign-On

Oracle Commerce enables you to integrate customer logins on your storefront with an external customer data store or identity management tool.

For example, suppose you have an existing informational website with a large number of customer accounts. When you create a new Commerce site, you may want to provide existing customers with accounts on the commerce site.

Storefront Single Sign-On (SSO) is implemented using SAML (Security Assertion Markup Language) 2.0, which is an open-standard XML-based data-exchange format. Before setting up storefront SSO, you should be familiar with SAML 2.0. For information about SAML 2.0, see:

`https://en.wikipedia.org/wiki/SAML_2.0`

The SAML 2.0 specification is available at:

`http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`

Storefront Single Sign-On (SSO) provides two main benefits:

- Your Commerce environment can share logins with another site or system, so that logging into one environment automatically logs a shopper into the other.
- If an unregistered shopper is logged into the external system, then the first time the shopper accesses the Commerce site, a shopper profile is automatically created.

There are two ways you can use SSO on your storefront:

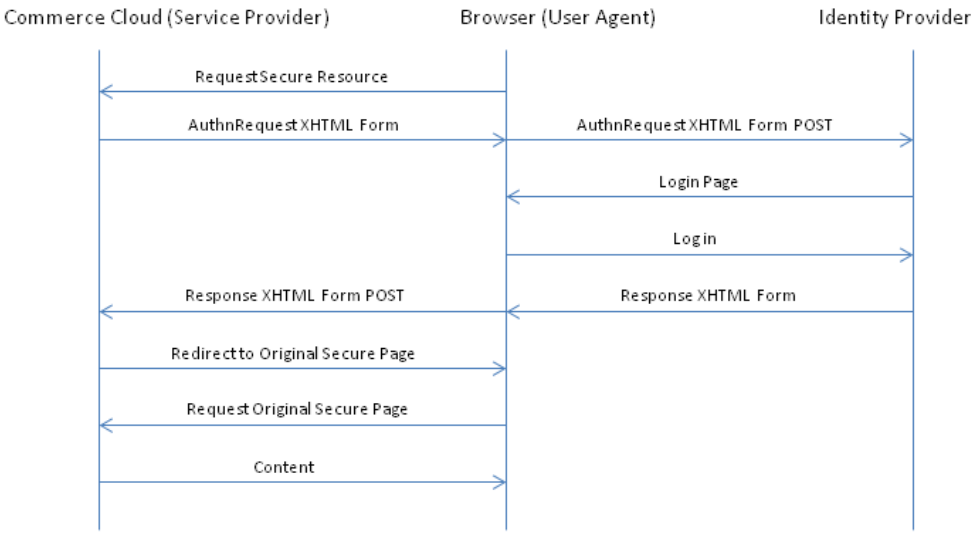
- You can configure your storefront to use SSO exclusively. In this case, all shopper logins are maintained in the external system.
- You can configure your storefront to support both SSO and standard logins. In this case, the logins for shoppers using SSO are maintained in the external system, and the logins for other shoppers are maintained in Commerce.

This section describes both of these configurations and how to set them.

Understand storefront SSO message flow

SAML 2.0 supports a variety of different message flows for authentication and authorization.

The following diagram illustrates the approach used by Commerce. It shows the flow of messages when a shopper logs into a Commerce storefront using storefront SSO. Note that in SAML terminology, Commerce is referred to as the service provider, while the external system that provides authentication is called the identity provider:



Configure storefront SSO

Setting up storefront SSO involves the following steps:

1. Configure Commerce storefront SSO settings.
2. Download the service provider entity descriptor from Commerce.
3. Upload the service provider entity descriptor to the identity provider, then download the corresponding identity provider entity descriptor.
4. Upload the identity provider entity descriptor to Commerce.
5. Configure CORS to enable the identity provider to access Commerce resources.
6. Modify the storefront so that the links for logging in and accessing an account direct the shopper to either the storefront or the identity provider, as appropriate.
7. If your Commerce environment is running multiple sites, repeat this process for each new site you create.

These steps are described in the sections that follow.

Note that if you configure your storefront to use SSO exclusively and your identity provider allows multiple accounts to share the same email address, you should enable sharing of email addresses in Commerce as well. See Allow profiles to share an email address for information about how to do this. If you configure your storefront to support both SSO and standard logins, neither your identity provider nor Commerce should support sharing of email addresses.

Configure Commerce storefront SSO settings

Use the `PUT /ccadmin/v1/merchant/samlSettings` endpoint in the Admin API to configure Commerce to use storefront SSO. The endpoint request body includes the following properties that are used to create the service provider entity descriptor:

- `enabled` – If `true`, support for SSO is enabled. Default is `false`.
- `nameIdPolicyFormat` – The SAML name ID policy to use. Default is `urn:oasis:names:tc:SAML:2.0:nameid-format:persistent`.

In addition, the request body can include several properties that control the SAML security policies that Commerce enforces. The values of these properties are used to create settings in the service provider entity descriptor:

- `signAuthnRequest` – If `true`, the SAML request message will be signed. Default is `true`.
- `nameIdPolicyAllowCreate` – If `true`, Commerce allows the identity provider to create persistent name identifiers for sessions. Default is `true`.
- `requireEncryptedAssertions` – If `true`, Commerce accepts SAML assertions from the identity provider only if they are encrypted. Default is `true`. For security reasons, this should be set to `true` in your production environment.
- `requireSignedResponse` – If `true`, Commerce accepts authorization responses from the identity provider only if they include a signature. Default is `true`. For security reasons, this should be set to `true` in your production environment.

The following call enables and configures SSO on a Commerce instance:

```
PUT /ccadmin/v1/merchant/samlSettings  HTTP/1.1
Authorization: Bearer <access_token>

{
  "enabled": true,
  "nameIdPolicyFormat": "urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent",
  "requireEncryptedAssertions": true,
  "requireSignedResponse": true,
  "signAuthnRequest": true,
  "nameIdPolicyAllowCreate": true
}
```

Note that it may take several minutes for the changes to propagate to the storefront server.

Download the service provider entity descriptors

Once you have configured SSO on your Commerce instance, you can use the `GET /ccstore/v1/merchant/samlSettings` endpoint in the Store API to return service provider entity descriptors. For example, if you send the following request:

```
GET /ccstore/v1/merchant/samlSettings  HTTP/1.1
Authorization: Bearer <access_token>
```

The response will be similar to this:

```
{
  "spEntityDescriptor": "<service provider entity descriptor>",
  "links": [
    {
      "rel": "self",
      "href": "http://myserver.example.com:7002/ccstore/v1/merchant/
samlSettings"
    }
  ]
}
```

The entity descriptor is returned as the value of the `spEntityDescriptor` property. This value is an XML document using a standard SAML 2.0 format for describing the configuration of the service provider.

To return the entity descriptor in Base64 encoding, call the endpoint with the `encode` query parameter set to `true`:

```
GET /ccstore/v1/merchant/samlSettings?encode=true  HTTP/1.1
Authorization: Bearer <access_token>
```

Save the value of `spEntityDescriptor` as a standalone document. Note that you need to create separate service provider entity descriptors for the preview context and the live context, as they will need to be registered with the identity provider as separate service providers.

Register the service providers with the identity provider

Register the two service providers (the preview context and the live context) with the identity provider. (Register both service providers with the same identity provider; do not create multiple identity providers.) Depending on the API or other tools the identity provider supplies for this purpose, you register the service providers either by uploading the service provider entity descriptor documents that you downloaded from Commerce, or by manually configuring the identity provider with the data from the documents.

Once you have registered the service providers, download the identity provider entity descriptor. There should be only one identity provider entity descriptor, which applies to both the preview context and the live context. The identity provider entity descriptor should be a Base64-encoded XML file. If the generated file is not Base64-encoded, encode it before you upload it to Commerce.

Upload the identity provider entity descriptor to Commerce

Use the `PUT /ccadmin/v1/samlIdentityProviders/default` endpoint to upload the identity provider entity descriptor to Commerce, and to map assertion attributes to profile properties. This mapping enables automatic creation of shopper profiles in Commerce.

The request body includes the following properties:

- `encodedIdpMetadata` -- The Base64-encoded identity provider entity descriptor.
- `loginAttributeName` – The identity provider attribute that stores the shopper's login name.
- `emailAttributeName` -- The identity provider attribute that stores the shopper's email address.
- `requiredAttributeToPropertyMap` – A map in which the keys are identify provider attributes and the values are the names of the corresponding required Commerce profile properties.
- `optionalAttributeToPropertyMap` – A map in which the keys are identify provider attributes and the values are the names of additional (non-required) Commerce profile properties. (Optional.)

For example:

```
PUT /ccadmin/v1/samlIdentityProviders/default  HTTP/1.1
Authorization: Bearer <access_token>

{
  "loginAttributeName": "uid",
  "emailAttributeName": "email",
  "encodedIdpMetadata": "<identity provider entity descriptor>",
  "requiredAttributeToPropertyMap": {
    "uid": "login",
    "email": "email"
  },
  "optionalAttributeToPropertyMap": {
    "fName": "firstName",
    "lName": "lastName"
  }
}
```

Configure CORS

To allow the identity provider to make requests to a site running on your Commerce instance, you must add the identity provider's domain to the list of domains for which the site supports CORS. To do this, add the domain to the `allowedOriginMethods` property of the corresponding site object and specify GET (for redirect requests) and POST. For example:

```
PUT /ccadmin/v1/sites/siteUS  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
    "properties": {
        "allowedOriginMethods": {
            "http://www.myIdentityProvider.com": "GET,POST"
         }
    }
}
```

See CORS support for more information about CORS.

Modify the storefront

To enable shoppers to access SSO on your storefront, you must modify any widgets that handle login or registration. The necessary modifications depend on whether your storefront supports SSO exclusively or supports both SSO and standard logins:

- If your storefront supports SSO exclusively, provide login and registration links that redirect the shopper to the identity provider.
- If your storefront supports both standard logins and SSO, provide two sets of links, one set that directs the shopper to the standard login and registration and one set that directs the shopper to the identity provider.

By default, the Header Widget in Commerce includes a Login/Registration element with links to the standard login and registration. In addition, Commerce provides an SSO-specific SAML Login/Registration element. If your storefront supports SSO exclusively, you can replace the

standard Login/Registration element with the SAML Login/Registration element. If your storefront supports both standard logins and SSO, you can include both login/registration elements in the widget. Note that the text and styling are identical in both elements, so if you do include both, you should modify one or both of them to make it clear which kind of login each supports.

Implement storefront SSO for multiple sites

If your Commerce environment is running multiple sites, all sites must be configured with the same identity provider. However, in the identity provider, each site should be registered as a separate service provider. (There should actually be two service providers registered for each site, one for the preview context and one for the live context.)

For each site, follow the instructions above to generate service provider entity descriptors for the preview context and live context. If your site URLs are differentiated by context root, pass the site ID in the `x-ccsite` header when calling the `GET /ccstore/v1/merchant/samlSettings` endpoint:

```
GET /ccstore/v1/merchant/samlSettings  HTTP/1.1
Authorization: Bearer <access_token>
x-ccsite: <siteId>
```

If your site URLs are differentiated by domain or subdomain, you do not need to pass the site ID in the header. Specifying the domain in the URL when calling the `GET /ccstore/v1/merchant/samlSettings` endpoint will identify the site.

The entity ID and assertion consumer POST location in each site's service provider entity descriptor should match the site's base URL. Register the service providers either by uploading the service provider entity descriptor documents that you downloaded from Commerce, or by manually configuring the identity provider with the data from the documents. Note that only a single identity provider is currently supported, so all service providers must be configured in the same identify provider.

After registering the service providers, download the identity provider entity descriptor, and then use the `PUT /ccadmin/v1/samlIdentityProviders/default` endpoint to upload it to Commerce. (There should be only one identity provider entity descriptor for a Commerce instance, regardless of the number of sites. If you have already performed this step for one site, you can skip it for other sites you add.) The identity provider entity descriptor should be a Base64-encoded XML file. If the generated file is not Base64-encoded, encode it before you upload it to Commerce.

In addition to adding each site as a service provider, you must update the CORS `allowedOriginMethods` property for each site to include the identity provider.

Understand storefront SSO limitations

The storefront single sign-on implementation has some limitations you should be aware of.

Storefront SSO does not provide single log-out (SLO). If a shopper signs out of Commerce or if the shopper's session times out, the shopper may continue to be logged into the identity provider until the identity provider session times out. Similarly, if the shopper signs out or is timed out of the identity provider, the shopper may remain logged into Commerce until the Commerce session times out.

If a Commerce storefront uses SSO exclusively, customer service agents cannot reset shopper passwords in the Agent Console; shopper passwords can be reset only in the identity provider. If a Commerce storefront supports both SSO and standard logins, passwords of shoppers using standard logins can be reset in the Agent Console, but passwords of shoppers using SSO can be reset only in the identity provider.

Implement storefront SSO for account-based shoppers

If your Commerce store is configured to support account-based commerce, storefront SSO lets you easily create business-account contacts for shoppers who have profiles in an external customer data store or identity management tool.

You can add authenticated shoppers to any active Commerce business account. See Configure Business Accounts for details about accounts, roles, and contacts.

You implement storefront SSO for account-based stores in much the same way as you would for stores that support only individual shoppers. This section includes only the information related to account-based stores. Before you read this section, make sure you have read all the sections that precede it in Implement Storefront Single Sign-On.

Note: The account-based commerce feature may not be enabled in your environment. Contact your Oracle account manager for more details on how to activate this functionality.

Understand how storefront SSO determines a shopper's account and role

Commerce uses the account ID supplied during login to determine how to associate the authenticated shopper with an account and an account-based role.

- If the shopper is a contact for the target account, and the account is active, the shopper is logged into the account with their assigned role.
- If the shopper does not have a profile on your store and the target account is active, a new contact is created for the shopper in that account and the shopper is logged in. If a role is specified in the identity provider entity descriptors, the new contact is assigned that role. If no role is specified, the new contact is assigned the default Buyer role.
- If the shopper is not a contact for the target account (or if that account is not active) but is a contact for one or more other active accounts, the shopper is denied access to the store.
- If the shopper has a profile on your store but is not a contact for the target account or any other active account, the shopper is denied access to the store.
- If your store supports both account-based shoppers and individual shoppers, the way you configure your store determines whether the shopper is logged in:
  - If the shopper does not have a profile on your store and you set the `fallbackToB2cUserCreation` property to true when you upload the identity provider entity descriptors, a new individual shopper profile is created. If `fallbackToB2cUserCreation` is set to false, the shopper is denied access to the store.
  - If your storefront includes SSO elements for both types of shoppers, when a shopper who does not have a profile on your store logs in with the regular (that is, not account-based) SSO link, Commerce creates a profile for a new individual shopper.

This shopper cannot later log into an account via the link for account-based SSO. If the original SSO login was accidental and the shopper needs access to an account, they must

contact the merchant, who can either manually add the shopper as a contact in the target account or simply delete the existing shopper profile.

Important: If you remove a shopper from your external identity management system, you must also deactivate or remove their associated contact from the Commerce business account they log into with SSO. Leaving the contact active allows the shopper to continue accessing your store and account features by logging in with their Commerce login credentials.

Configure account-based storefront SSO

Setting up storefront SSO for an account-based store involves the following steps:

1. Configure Commerce to use storefront SSO. See Configure Commerce storefront SSO settings for more information.
2. Once you have configured SSO on your Commerce instance, you can return service provider entity descriptors. See Download the service provider entity descriptor from Commerce for more information.
3. Upload the service provider entity descriptor to the identity provider, then download the corresponding identity provider entity descriptor. See Register the service providers with the identity provider for more information.
4. Upload the identity provider entity descriptors to Commerce, and map assertion attributes to profile properties. This mapping enables automatic creation of shopper profiles in Commerce. See Upload the identity provider entity descriptor to Commerce for more information, and see Identity provider entity descriptors for account based stores for additional properties specific to account-based shoppers.
5. Configure CORS to enable the identity provider to access Commerce resources. See Configure CORS for more information.
6. Modify the storefront so that the links for logging in and accessing an account direct the shopper to either the storefront or the identity provider, as appropriate. See Modify login layouts for account based shoppers for more information.

Identity provider entity descriptors for account based stores

Use the `PUT /ccadmin/v1/samlIdentityProviders/default` endpoint to upload the identity provider entity descriptors to Commerce, and to map assertion attributes to profile properties. This mapping enables automatic creation of shopper profiles and contacts in Commerce.

In addition to the properties described in Upload the identity provider entity descriptors to Commerce, the request body includes the following properties specifically for account-based stores:

- `organizationAttributeName` -- The identity provider attribute that stores the contact's account ID.
- `roleAttributeName` – The identity provider attribute that stores the contact's role in the account specified by `organizationAttributeName`.
- `fallbackToB2cUserCreation` – If true, Commerce creates an individual shopper profile if the contact logs in with invalid account credentials. See Create Page Layouts that Support Different Types of Shoppers for more information.

For example:

```
PUT /ccadmin/v1/samlIdentityProviders/default  HTTP/1.1
Authorization: Bearer <access_token>

{

  "loginAttributeName": "uid",
  "emailAttributeName": "email",
  "organizationAttributeName": "organizationId",
  "roleAttributeName": "Role",
  "fallbackToB2cUserCreation": true,
  "encodedIdpMetadata": "<identity provider entity descriptor>",
  "requiredAttributeToPropertyMap": {
    "uid": "login",
    "email": "email",
    "firstName": "firstName",
    "lastName": "lastName"
  },
  "optionalAttributeToPropertyMap": {
    "addressFirstName": "address.firstName",
    "addressLastName": "address.lastName",
    "address1": "address.address1",
    "postalCode": "address.postalCode",
    "city": "address.city",
    "country": "address.country",
    "state": "address.state"
  }
}
```

Modify login layouts for account based shoppers

This section describes a sample Header widget that lets account-based shoppers access SSO on your storefront. See Modify the storefront for overview information about updating the storefront to enable SSO.

In this sample, the out-of-the-box Header widget has been modified to include a customized version of the Login/Register element that includes an SSO login link. When the contact clicks the link, they see a login modal where they enter an account ID. If the ID matches

For details about how to create widgets, see Create a Widget.

The element template file provides the HTML rendering code for the element:

```
<div id="CC-header-sso-login" class="col-md-6">
   <a href="#CC-headermodalpane" id="CC-linkSsoLogin"
data-original-title="ssoLogin"
     data-bind="click: $parent.showSsoLoginSection.bind($parent),
       widgetLocaleText: 'ssoLoginLinkText',
       event:
{ mousedown: $parent.handleMouseDown.bind($parent, $parents[1]),
         mouseup: $parent.handleMouseUp.bind($parent, $parents[1])}">
   </a>
</div>
```

The following code is for the SSO pane that appears when the shopper clicks the SSO login link:

```
<!--Pane for SSO Login-->
   <div id="CC-ssoLoginPane">
    <div class="modal-header CC-header-modal-heading">
     <h3 class="modal-title"
id="CC-sso-login-text-title" data-bind="widgetLocaleText:
'ssoLoginText'"></h3>
     </div>
     <div class="modal-body cc-modal-body">
      <div id="CC-sso-login-section" data-bind="with: $parent.user">
      <div class="form-group row">
        <div class="controls col-md-12">
          <label class="control-label inline" for="CC-sso-login-
account-input"
                 data-bind="widgetLocaleText:'accountIdText'">
          </label>
          <input type="email" class="col-md-5 form-control"
id="CC-sso-login-account-input" aria-required="true"
                 data-bind="validatableValue: ssoLoginAccountName,
                   widgetLocaleText : { value:'accountIdText',
attr:'placeholder' }"/>
        </div>
      </div>
    </div>
   </div>
  <div class="modal-footer CC-header-modal-footer">
     <div class="center-block">
      <button type="button" id="CC-sso-login"
class="cc-button-primary" data-bind="widgetLocaleText: 'buttonLogin',
click: function(data, event) { doSsoLogin.bind($data, $parent, event)
() },
event: { mousedown: handleMouseDown.bind($data, $parent),
mouseup: handleMouseUp.bind($data, $parent) }"></button>
      <button type="button" id="CC-sso-login-cancel"
class="cc-button-secondary" data-dismiss="modal"
data-bind="widgetLocaleText: 'buttonCancel', click: function(data,
event)
{ handleCancelSsoLogin.bind(data, $parent, event)() },
event: { mousedown: handleMouseDown.bind($data, $parent),
mouseup: handleMouseUp.bind($data, $parent) }"></button>
    </div>
   </div>
</div>
```

The element's JavaScript file includes a click handler for cancelling the SSO login modal.

```
/**
     * Click handler to cancel the SSO login modal.
     * @param data Data that is passed on the click event.
     * @param event jQuery event of the click event on the cancel
button.
```

```
     */
    handleCancelSsoLogin: function(data, event) {
      if('click' === event.type ||
(('keydown' === event.type ||
 'keypress' === event.type) && event.keyCode === 13)) {
        notifier.clearError(this.WIDGET_ID);
        navigation.doLogin(navigation.getPath(), data.links().home.route);
      }
      return true;
    },
```

The following sample is an event handler for the log in with SSO link.

```
    /**
     * Event handler for the Log In With SSO link on the B2B login modal.
     * It shows the SSO login modal.
     * @param data Data passed when the link is clicked.
     */
    showSsoLoginSection: function(data) {
      this.hideAllSections();
      $('#CC-ssoLoginPane').show();
      $('#CC-sso-login-account-input').focus();
      data.ssoLoginAccountName('');
    },
```

The following sample is the click event handler for the Login button on the SSO Login modal.

```
    /**
     * Click event handler for the Login button in SSO Login modal.
     * @param data Data passed when the login button is clicked.
     * @param event jQuery event for the click event.
     */
    doSsoLogin: function(data, event) {
      if ('click' === event.type ||
(('keydown' === event.type || 'keypress' === event.type)
&& event.keyCode === 13)) {
        data.user().handleSamlLogin();
      }
      return true;
    },

    createOrganizationRequestSuccess: function(){
      this.hideAllSections();
      $('#CC-headermodalpane').children(".modal-dialog").css('top', '20%');
      $('#CC-organizationRequestSuccessPane').show();
    },

    createOrganizationRequestFailure: function(pResponse){
      this.modalMessageText(pResponse.message);
      this.showErrorMessage(true);
    },
```

During a SAML login, the `ssoLoginAccountName` variable is sent automatically as the `relay_state`. This variable is used for organization validation. If the `relay_state` is not passed in, no validation occurs when the customer logs into their parent organization. However, if the `relay_state` is passed in, the system validates that the customer has access to the account.

This is the event handler for the login with SSO link on the login modal.

```
    /**
     * Event handler for the Log In With SSO link on the B2B login
modal.
     * It shows the SSO login modal.
     * @param data Data passed when the link is clicked.
     */
    showSsoLoginSection: function(data) {
      this.hideAllSections();
      $('#CC-ssoLoginPane').show();
      $('#CC-sso-login-account-input').focus();
      data.ssoLoginAccountName('');
    },
```

Renew your SAML 2.0 SSO certificate

Your SAML certificate is valid for a period of time determined by your Identity Provider. An administrator at your store should keep track of when your SAML certificate is due to expire, and renew the certificate in advance.

To check the expiration date on your certificate, use the `getSamlSettings` endpoint in the Admin API:

```
GET /ccadmin/v1/merchant/samlSettings  HTTP/1.1
Authorization: Bearer <access_token>
```

The response includes a `certificate` object with `startDate` and `endDate` properties. The `endDate` property indicates when the certificate will expire:

```
{
    "nameIdPolicyFormat": "urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent",
    "requireEncryptedAssertions": false,
    "requireSignedResponse": false,
    "certificate": {
        "endDate": "2021-12-14T18:16:00.000Z",
        "startDate": "2020-12-15T18:16:00.000Z"
    },
    "signAuthnRequest": true,
    "spMetadata": "<service provider entity descriptor>",
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/merchant/
samlSettings"
        }
```

```
    ],
    "nameIdPolicyAllowCreate": true,
    "enabled": false
}
```

If your certificate will be expiring soon, use the `renewSamlCertificate` endpoint to renew it, Note that the request body should be empty:

```
POST /ccadmin/v1/merchant/renewSamlCertificate  HTTP/1.1
Authorization: Bearer <access_token>
```

If the request succeeds, the response has a status code of 200 to indicate the certificate has been renewed. The response payload includes the start date and end date for the new certificate:

```
{
    "endDate": "2022-12-10T17:41:09.000Z",
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/merchant/
renewSamlCertificate"
        }
    ],
    "startDate": "2021-12-11T17:41:09.000Z"
}
```

If the request is accepted but certificate renewal is still in progress, the response has a status code of 202. You can use the `getSamlSettings` endpoint later to check whether the certificate has been successfully renewed.

22

Implement Single Sign-On for Internal Users

You can configure Oracle Identity Cloud Service (IDCS) to enable the Oracle Commerce administration interface and Agent Console to support single sign-on (SSO) with other Oracle Cloud applications.

Two single sign-on implementations are supported, OpenID Connect and SAML 2.0. Note that OpenID Connect SSO supports the use of IDCS OAuth 2 application keys with Oracle Commerce, to simplify integration with other Oracle applications. SAML 2.0 SSO does not support this.

Configure SSO with OpenID Connect

You can configure Oracle Identity Cloud Service to provide single sign-on (SSO) for Oracle Commerce applications using OpenID Connect.

Before you begin, you will need the following:

- An Oracle Commerce account with authorization rights to configure federated authentication.
- An Oracle Identity Cloud Service account with authorization rights to manage applications and users (Identity Domain Administrator or Application Administrator).

IDCS must be configured to require multi-factor authentication (MFA) logins for users that can access the Oracle Commerce administration interface, to meet the requirements of PCI.

Configure Oracle Commerce in Oracle Identity Cloud Service

This section describes how to register and activate the Oracle Commerce administration and agent applications in Oracle Identity Cloud Service. You can then assign users or groups to these Oracle Commerce applications.

Register and activate the Oracle Commerce administration application

1. In the Oracle Identity Cloud Service administration console, select **Applications**, and then click **Add**.
2. Click **Confidential Application**.
3. Enter the name: Oracle CX Commerce Admin
4. Verify that the **Display in My Apps** checkbox is selected, and then click **Next**.
5. Click **Configure this application as a client now**.
6. For **Allowed Grant Types**, check **Resource Owner**, **Client Credentials**, **Refresh Token**, and **Authorization Code**.
7. For **Redirect URL**, enter: `https://<admin-server>/occs-admin/sso-login.jsp`
8. For **Logout URL**, enter: `https://<admin-server>/occs-admin/sso-logout.jsp`
9. For **Post Logout Redirect URL**, enter: `https://<admin-server>/occs-admin`
10. In the **Token Issuance Policy** section**,** under **Authorized Resources**, select **Specific**.

11. Under **Grant the client access to Identity Cloud Service Admin APIs**, click **Add**, and add **Identity Domain Administrator**.
12. Click **Next**.
13. Under **Expose APIs to Other Applications,** select **Configure this application as a resource server now**.
14. For **Primary Audience**, enter: `https://<admin-server>/occs-admin`
15. Click **Next**.
16. Under **Authorization**, check **Enforce Grants as Authorization**.
17. Click **Finish**. Oracle Identity Cloud Service should display a confirmation message.

Register and activate the Oracle Commerce agent application

1. In the Oracle Identity Cloud Service administration console, select **Applications**, and then click **Add**.
2. Click **Confidential Application**.
3. Enter the name: Oracle CX Commerce Agent
4. Verify that the **Display in My Apps** checkbox is selected, and then click **Next**.
5. Click **Configure this application as a client now**.
6. For **Allowed Grant Types**, check **Resource Owner**, **Client Credentials**, **Refresh Token**, and **Authorization Code**.
7. For **Redirect URL**, enter: `https://<agent-server>/occs-agent/sso-login.jsp`
8. For **Logout URL**, enter: `https://<agent-server>/occs-agent/sso-logout.jsp`
9. For **Post Logout Redirect URL**, enter: `https://<agent-server>/occs-agent`
10. In the **Token Issuance Policy** section**,** under **Authorized Resources**, select **Specific**.
11. Under **Grant the client access to Identity Cloud Service Admin APIs**, click **Add**, and add **Identity Domain Administrator**.
12. Click **Next**.
13. Click **Next**.
14. Under **Authorization**, check **Enforce Grants as Authorization**.
15. Click **Finish**. Oracle Identity Cloud Service should display a confirmation message.

Configure OpenID Connect SSO for Oracle Commerce

This section describes how to configure SSO in Oracle Commerce applications with Oracle Identity Cloud Service.

Configure an identity provider

1. Log in as an administrator at: `https://<commerce-admin-domain>/occs-admin/#/adminLogin`
   This is a special login path that allows your primary administrator direct access to the Oracle Commerce administration interface even when SSO is enabled, so that edits can be made to the SSO settings. This login requires multi-factor authentication.

2. Click the menu icon and select **Settings**.
3. On the **Settings** page, click **Oracle Integrations** section.
4. Select **IDCS** from the popup menu.
   If **IDCS** is not available as an option on this menu, contact your Oracle representative.
5. For **IDP Base URL**, enter the URL of your IDCS instance.
6. For **Admin App Client ID** , enter the **Client ID** of the Oracle Commerce administration application you set up in IDCS. (You can find this value on the **Configuration Page**, under **General Information**.)
7. For **Admin App Client Secret**, enter the **Client Secret** for the Oracle Commerce administration application from IDCS. (Click **Show Secret** to reveal this value.)
8. For **Agent App Client ID**, enter the **Client ID** of the Oracle Commerce agent application you set up in IDCS. (You can find this value on the **Configuration Page**, under **General Information**.)
9. For **Agent App Client Secret**, enter the **Client Secret** for the Oracle Commerce agent application from IDCS. (Click **Show Secret** to reveal this value.)
10. Click **Save** to save your changes, then logout.

Use IDCS OAuth 2 application keys with Oracle Commerce

OpenID Connect SSO supports the use of IDCS OAuth 2 application keys with Oracle Commerce, to simplify integration with other Oracle applications. To set up an OAuth 2 application key:

- Create a **Confidential Client**.
- Under **Allowed Grant Types**, select **Client Credentials**.
- Under **Authorized Resources**, select **Specific**.
- Under **Add Scope**, select Oracle CX Commerce Admin. For the scope, enter: `https://<commerce-URL>/occs-admin/auth/appid.full_control`

An application with this scope will have access to both the Admin and Agent APIs.

Verify the integration

This section describes how to verify that SSO and single log-out (SLO) work when initiated from Oracle Identity Cloud Service (identity provide initiated SSO and SLO) and from Oracle Commerce (service provider initiated SSO and SLO).

Verify identity provider initiated SSO

1. Access the Oracle Identity Cloud Service My Console at: `https://<IDCS-Service-Instance>.identity.oraclecloud.com/ui/v1/myconsole`
2. Log in using credentials for a user that is assigned to the Oracle Commerce agent and administration applications. (Oracle Identity Cloud Service displays a shortcut to Oracle Commerce applications under **My Apps**.)
3. Click the Oracle Commerce agent application. The Oracle Commerce agent home page appears.
4. On the home page, verify that the logged-in user is the same for both Oracle Commerce and Oracle Identity Cloud Service. This confirms that SSO that is initiated from Oracle Identity Cloud Service is working.

Verify service provider initiated SSO

1. Access Oracle Commerce at: `<siteurl>/occs-admin`
   You will redirected to the Oracle Identity Cloud Service Sign In page.
2. Log in using credentials for a user that is assigned to the Oracle Commerce administration application. The Oracle Commerce administration home page appears.
3. On the Oracle Commerce administration home page, verify that the logged-in user is the same for both Oracle Commerce and Oracle Identity Cloud Service. This confirms that SSO initiated from Oracle Commerce administration is working.

If the user can access only the dashboard page in Oracle Commerce administration after logging in, your Commerce Administrator will need to add the appropriate roles in the administration interface. By default, new users have dashboard access only.

Verify identity provider initiated SLO

1. On the Oracle Identity Cloud Service home page, click the user name in the upper-right corner, and then select **Sign Out** from the drop-down list.
2. Access the user profile in Oracle Commerce, and verify that the login page appears. This confirms that SLO is working and that the user is no longer logged in to Oracle Commerce and Oracle Identity Cloud Service.

Verify service provider initiated SLO

1. On the Oracle Commerce administration interface or agent console, click the user icon in the upper-right corner, and then select **Logout** from the drop-down list.
2. Click **OK** at the confirmation message that displays.
3. Access the Oracle Identity Cloud Service My Console, and then confirm that the login page appears. This confirms that SLO is working and that the user is no longer logged in to Oracle Commerce and Oracle Identity Cloud Service.

Troubleshoot the integration

Oracle Identity Cloud Service may display the following message:

"You are not authorized to access the app. Contact your system administrator."

The two most likely causes are:

- The administrator revokes access for the user at the same time as the user tries to access Oracle Commerce using Oracle Identity Cloud Service. If this happens, access the Oracle Identity Cloud Service administration console, select **Applications**, **Oracle CX Commerce Admin** (or **Oracle CX Commerce Agent**), **Users**, and then click **Assign** to re-assign the user.
- The OpenID Connect integration between the Oracle Identity Cloud Service and Oracle Commerce has been deactivated. In this case, access the Oracle Identity Cloud Service administration console, select **Applications**, **Oracle CX Commerce Admin**, click **Activate**, and then click **Activate Application**. Oracle Identity Cloud Service displays a confirmation message.

For other issues, contact your Oracle representative.

Configure SSO with SAML 2.0

You can configure Oracle Identity Cloud Service to provide single sign-on (SSO) for Oracle Commerce applications using SAML 2.0.

Before you begin, you will need the following:

- An Oracle Commerce account with authorization rights to configure federated authentication.
- An Oracle Identity Cloud Service account with authorization rights to manage applications and users (Identity Domain Administrator or Application Administrator).
- Identity provider metadata. Use the following URL to access the metadata: `https://<IDCS-Service-Instance>.identity.oraclecloud.com/fed/v1/metadata`

IDCS must be configured to require multi-factor authentication (MFA) logins for users that can access the Oracle Commerce administration interface, to meet the requirements of PCI.

Note: SAML 2.0 SSO does not support using IDCS OAuth 2 application keys with Oracle Commerce. If you want to use IDCS OAuth 2 application keys, use OpenID Connect SSO instead.

Configure SAML 2.0 SSO for Oracle Commerce

This section describes how to configure SSO in Oracle Commerce apps with Oracle Identity Cloud Service.

Configure an identity provider

1. Log in as an administrator at: `https://<commerce-admin-domain>/occs-admin/#/adminLogin`
   This is a special login path that allows your primary administrator direct access to the Oracle Commerce administration interface even when SSO is enabled, so that edits can be made to the SSO settings. This login requires multi-factor authentication.
2. Click the menu icon and select **Settings**.
3. On the **Settings** page, click **Oracle Integrations** section.
4. Select **IDCS** from the popup menu.
   If **IDCS** is not available as an option on this menu, contact your Oracle representative.
5. Upload the identity provider metadata file (see above).
6. Logout.

Configure Oracle Commerce in Oracle Identity Cloud Service

This section describes how to register and activate the Oracle Commerce applications. You can then assign users or groups to these applications.

Register and activate the Oracle Commerce administration application

1. Access the Oracle Identity Cloud Service administration console, select **Applications**, and then click **Add**.
2. Click **SAML Application**.
3. Enter the name: Oracle CX Commerce Admin
4. Verify that the **Display in My Apps** checkbox is selected, and then click **Next**.
5. For **Entity ID**, enter: `https://<commerce-admin-domain>/occs-admin`
6. For **Assertion Consumer URL**, enter: `https://<commerce-admin-domain>/occs-admin/sso-login.jsp`
7. For **NameID Format**, use: Persistent

8. For **NameID Value**, use: User Name
9. Open **Advanced Settings**.
10. For **Signed SSO**, use: Assertion
11. For **Signature Hashing Algorithm**, use: SHA-256
12. Select **Enable Single Logout**.
13. For **Logout Binding**, use: POST
14. For **Single Logout URL**, enter: `https://<commerce-admin-domain>/occs-admin/sso-logout.jsp`
15. For **Logout Response URL**, enter: `https://<commerce-admin-domain>/occs-admin`
16. Open **Attribute Configuration**.
17. Add the following attributes:

Name	Format	Entry	Value
uid	Basic	User Attribute	User Name
email	Basic	User Attribute	Primary Email
firstName	Basic	User Attribute	First Name
lastName	Basic	User Attribute	Last Name

Now click **Activate**, and then click **Activate Application**. Oracle Identity Cloud Service displays a confirmation message.

Register and activate the Oracle Commerce agent application

1. In the Oracle Identity Cloud Service administration console, select **Applications**, and then click **Add**.
2. Click **SAML Application**.
3. Enter the name: Oracle CX Commerce Agent
4. Verify that the **Display in My Apps** checkbox is selected, and then click **Next**.
5. For **Entity ID**, enter: `https://<commerce-agent-domain>/occs-agent`
6. For **Assertion Consumer URL**, enter: `https://<commerce-agent-domain>/occs-agent/sso-login.jsp`
7. For **NameID Format**, use: Persistent
8. For **NameID Value**, use: User Name
9. Open **Advanced Settings**.
10. For **Signed SSO**, use: Assertion
11. For **Signature Hashing Algorithm**, use: SHA-256
12. Select **Enable Single Logout**.
13. For **Logout Binding**, use: POST
14. For **Single Logout URL**, enter: `https://<commerce-agent-domain>/occs-agent/sso-logout.jsp`

15. For **Logout Response URL**, enter: `https://<commerce-agent-domain>/occs-agent`
16. Open **Attribute Configuration**.
17. Add the following attributes:

Name	Format	Type	Value
uid	Basic	User Attribute	User Name
email	Basic	User Attribute	Primary Email
firstName	Basic	User Attribute	First Name
lastName	Basic	User Attribute	Last Name

Now click **Activate**, and then click **Activate Application**. Oracle Identity Cloud Service displays a confirmation message.

Verify the integration

This section describes how to verify that SSO and single log-out (SLO) work when initiated from Oracle Identity Cloud Service (identity provide initiated SSO and SLO) and from Oracle Commerce (service provider initiated SSO and SLO).

Verify identity provider initiated SSO

1. Access the Oracle Identity Cloud Service My Console at: `https://<IDCS-Service-Instance>.identity.oraclecloud.com/ui/v1/myconsole`
2. Log in using credentials for a user that is assigned to the Oracle Commerce agent and administration applications. (Oracle Identity Cloud Service displays a shortcut to Oracle Commerce applications under **My Apps**).
3. Click the Oracle Commerce agent application. The Oracle Commerce agent home page appears.
4. On the home page, verify that the logged-in user is the same for both Oracle Commerce and Oracle Identity Cloud Service. This confirms that SSO that is initiated from Oracle Identity Cloud Service is working.

Verify service provider initiated SSO

1. Access Oracle Commerce at: `https://<commerce-admin-domain>/occs-admin`
   You will redirected to the Oracle Identity Cloud Service Sign In page
2. Log in using credentials for a user that is assigned to the Oracle Commerce administration application. The Oracle Commerce administration home page appears.
3. On the Oracle Commerce administration home page, verify that the logged-in user is the same for both Oracle Commerce and Oracle Identity Cloud Service. This confirms that SSO initiated from Oracle Commerce administration is working.

If the user can access only the dashboard page in Oracle Commerce administration after logging in, your Commerce Administrator will need to add the appropriate roles in the administration interface. By default, new users have dashboard access only.

Verifying identity provider initiated SLO

1. On the Oracle Identity Cloud Service home page, click the user name in the upper-right corner, and then select **Sign Out** from the drop-down list.
2. Access the user profile in Oracle Commerce, and verify that the login page appears. This confirms that SLO is working and that the user is no longer logged in to Oracle Commerce and Oracle Identity Cloud Service.

Verify service provider initiated SLO

1. On the Oracle Commerce administration interface or agent console, click the user icon in the upper-right corner, and then select **Logout** from the drop-down list.
2. Click **OK** at the confirmation message that displays.
3. Access the Oracle Identity Cloud Service My Console, and then confirm that the login page appears. This confirms that SLO is working and that the user is no longer logged in to Oracle Commerce and Oracle Identity Cloud Service.

Troubleshooting

Oracle Identity Cloud Service may display the following message:

"You are not authorized to access the app. Contact your system administrator."

The two most likely causes are:

- The administrator revokes access for the user at the same time as the user tries to access Oracle Commerce using Oracle Identity Cloud Service. If this happens, access the Oracle Identity Cloud Service administration console, select **Applications**, **Oracle CX Commerce Admin** (or **Oracle CX Commerce Agent**), **Users**, and then click **Assign** to re-assign the user.
- The SAML 2.0 integration between the Oracle Identity Cloud Service and Oracle Commerce has been deactivated. In this case, access the Oracle Identity Cloud Service administration console, select **Applications**, **Oracle CX Commerce Admin** (or **Oracle CX Commerce Agent**), click **Activate**, and then click **Activate Application**. Oracle Identity Cloud Service displays a confirmation message.

For other issues, contact your Oracle representative.

23
Configure Sites

Your Oracle Commerce instance initially has a single site. If you want to create other sites, you can do so by using the Admin REST API to create additional site objects.

Note that in order for requests to be routed to the correct site on your instance, each site must have a unique domain name. The topics in this section describe how to manage sites by creating, modifying, and deleting site objects.

Understand site objects

Each site is represented in Commerce by an object whose properties store configuration data for the site.

To set up a new site, you create a site object and set the values of the properties. You can also update an existing site by modifying the values of the site object properties, or delete a site by deleting its site object. Note that you must log into the Admin API on the administration server using a profile that has the Administrator role in order to create, modify, or delete sites.

To simplify site creation, Commerce designates one site as the default site. When you create a new site, the values for properties that you do not explicitly supply are copied to the new site from the default site, with the following exceptions:

- `productionURL` and `additionalProductionURLs` are left null if you do not supply values.
- `requireGDPRCookieConsent` and `requireGDPRP13nConsent` default to false.
- `priceListGroupList` defaults to a single JSON object containing the value of `defaultPriceListGroup`.
- id and name values are supplied automatically by Commerce.

In addition to serving as a template for site creation, the default site is used as the destination for requests to your Commerce instance when the site cannot be otherwise resolved (for example, a request whose URL does not match any of the URLs specified in the `productionURL` or `additionalProductionURLs` properties described below).

There is always a single default site in an individual instance of Commerce. Initially, this is the site whose `id` is `siteUS` (the site included with each Commerce instance). Once you have created other sites, you can change which site is the default. However, there are several things you should be aware of when setting the default site:

- The `enabled` property of the default site must be true. If you want to make a currently disabled site the default, first set `enabled` to true.
- You cannot set the `enabled` property to false on the default site. If you want to disable the site that is currently the default, first make a different site the default.
- If you set the `defaultSite` property to true on a site that is not currently the default, the `defaultSite` property is automatically set to false on the site that was previously the default.

- If you attempt to set the `defaultSite` property to false on the site that is currently the default, the call is ignored. To set the `defaultSite` property to false on a site, you must set it to true on a different site.
- You cannot delete the default site. If you want to delete the site that is currently the default, first make a different site the default.
- You cannot set the value of a site's `id` property to `defaultSite`.
- You can use `GET /ccadmin/v1/site/defaultSite` to return the current default site.

Most of the properties of site objects are site-specific – their values can be different for each site. A few properties are global – their values must be identical for all sites. When you set a site-specific property on a site object, the value you supply is applied only to that site object. When you set a global property on a site object, the value you supply is applied to **all** of the site objects in your Commerce instance.

The following site properties are global:

Property	Description
`recommendationsHost`	The hostname of the server for product recommendations. For example, `pt-recs-app1.us.example.com`.
`recommendationsPort`	The port number for accessing the server for product recommendations.

The remaining site properties are site-specific, including the following:

Property	Description
`productionURL`	The primary URL for accessing the site, without the protocol. If you have multiple sites running on your Commerce instance, their URLs can be differentiated by domain, subdomain, or context root.
`additionalProductionURLs`	Alternate URLs that can be used to access the site. Use this property to indicate other versions of the production URL, such as adding `www.example.com` to the `productionURL` entry of `example.com`. Note that you must also configure additional domain names using the `neverAuthenticateTargetHostNames` property in the Merchant Basic Auth endpoint. For additional information on this endpoint, refer to the REST API documentation.
`allowedOriginMethods`	An array that specifies external domains that are allowed to use CORS to make requests to the site. See CORS Support for more information.
`priceListGroupList`	An array of JSON objects that specify all of the price list groups associated with the site. Must include the price list group specified in `defaultPriceListGroup`.
`defaultPriceListGroup`	A JSON object that specifies the default price list group associated with the site.

Property	Description
`name`	Name of the site (maximum 254 characters).
`description`	Text describing the site.
`longDescription`	Text describing the site in greater detail.
`defaultCatalog`	The product catalog used for the site. Each site is associated with a single catalog.
`defaultBillingCountryId`	The country code of the default billing country for the site.
`defaultShippingCountryId`	The country code of the default shipping country for the site.
`defaultLocaleId`	A string containing an integer that identifies the default locale for the site. You can view the mapping of IDs to locales using the `listLocales` endpoint.`enabled`.
`additionalLocaleIds`	An array of IDs that specify other locales the site supports.
`useDefaultSiteLocale`	A Boolean specifying whether the locale specified by `defaultLocaleId` is displayed, even if it is not the browser's default locale.
`enabled`	A Boolean specifying whether the site is currently enabled.
`defaultSite`	A Boolean specifying whether the site is currently the default site.
`siteTypes`	An array listing the site types this site supports. For sites running consumer-based commerce, set this property to `["commerce"]`. If you have sites running account-based commerce or sites running both account-based and consumer-based commerce, this setting must be configured for you by Oracle.
`timeToLive`	Amount of time (in milliseconds) that an approved order or a partially paid order can remain available before payment is completed. If payment is not completed after this amount of time, the order is marked for cancellation.
`secondaryCurrency`	For sites that support loyalty points, the monetary currency that points can be converted to for calculating taxes and shipping costs. See Work with Loyalty Programs for more information.
`requireGDPRCookieConsent`	A Boolean specifying whether a shopper on the site must give consent before any cookies containing personal data are created on the shopper's machine. See Manage the Use of Personal Data for more information.
`requireGDPRP13nConsent`	A Boolean specifying whether a shopper on the site must give consent to be considered a member of any audience that uses shopper profile data. See Manage the Use of Personal Data for more information.

Create a site

You can use the `createSite` endpoint to create a new site, using the default site as a template.

For example:

```
POST /ccadmin/v1/sites  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
"properties":
  {
    "productionURL": "www.example2.com"
  }
}
```

The response is similar to the following:

```
{
   "longDescription": null,
   "priceListGroupList": [
      {
         "deleted": false,
         "repositoryId": "defaultPriceGroup",
         "active": true,
         "id": "defaultPriceGroup"
      }
   ],
   "productionURL": "www.example2.com",
   "timezone": "etc_utc",
   "description": null,
   "secondaryCurrency": null,
   "defaultCatalog": {
      "displayName": "Product Catalog",
      "repositoryId": "cloudCatalog",
      "id": "cloudCatalog"
   },
   "requireGDPRP13nConsent": false,
   "type": "siteConfiguration",
   "defaultBillingCountryId": null,
   "defaultShippingCountryId": null,
   "enabled": false,
   "requireGDPRCookieConsent": false,
   "payTaxInSecondaryCurrency": false,
   "timeToLive": null,
   "defaultLocaleId": "1",
   "activeTheme": null,
   "loyaltyPrograms": [

   ],
   "paymentOption": "0",
```

```
   "additionalProductionURLs": [

   ],
   "links": [
      {
         "rel": "self",
         "href": "http://myserver.example.com:7002/ccadmin/v1/sites"
      }
   ],
   "id": "100002",
   "defaultSite": false,
   "additionalLocaleIds": [

   ],
   "recommendationsHost": "pt-recs-app1.us.example.com",
   "favicon": null,
   "allowedOriginMethods": {

   },
   "noimage": null,
   "defaultPriceListGroup": {
      "deleted": false,
      "repositoryId": "defaultPriceGroup",
      "active": true,
      "id": "defaultPriceGroup"
   },
   "payShippingInSecondaryCurrency": false,
   "siteTypes": [
      "commerce"
   ],
   "recommendationsPort": "8080",
   "shipFromAddress": {

   },
   "repositoryId": "100002",
   "name": "100002"
}
```


Note: If you want to implement wish lists on a new site that you create using the Admin API, you must also create a new wish list environment and associate it with the site. To do this, use the `PUT /swm/rs/v1/sites/cc/{ccSiteId}` endpoint in the Social Wish Lists API. If you subsequently modify the site using the Admin API, you should use this endpoint to make the equivalent changes to the associated wish list environment. If you delete the site, you should also delete the associated wish list environment, using the `DELETE /swm/rs/v1/sites/cc/{ccSiteId}` endpoint.

For more information about these endpoints, see the REST API documentation that is available through the Oracle Help Center:

`http://docs.oracle.com/cloud/latest/commercecs_gs/CXOCC/`

`createSiteFromForm` endpoint

In addition to the `createSite` endpoint, which creates a site based on the default site, Commerce includes a `createSiteFromForm` endpoint that creates a site based on a site you specify. For example, to create a new site using site 100002 as a template:

```
POST /ccadmin/v1/sites/100002  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
"properties":
  {
    "productionURL": "www.example3.com"
  }
}
```

Update a site

You can use the `updateSite` endpoint to modify settings on an existing site.

For example:

```
PUT /ccadmin/v1/sites/100002  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "properties":
  {
    "productionURL": "www.example7.com",
    "defaultLocaleId": "3"
   }
}
```

The response is similar to the following:

```
{
   "longDescription": null,
   "priceListGroupList": [
      {
         "deleted": false,
         "repositoryId": "defaultPriceGroup",
         "active": true,
         "id": "defaultPriceGroup"
      }
   ],
   "productionURL": "www.example7.com",
   "timezone": "etc_utc",
   "description": null,
   "secondaryCurrency": null,
   "defaultCatalog": {
      "displayName": "Product Catalog",
      "repositoryId": "cloudCatalog",
      "id": "cloudCatalog"
```

```
    },
    "requireGDPRP13nConsent": false,
    "type": "siteConfiguration",
    "defaultBillingCountryId": null,
    "defaultShippingCountryId": null,
    "enabled": false,
    "requireGDPRCookieConsent": false,
    "payTaxInSecondaryCurrency": false,
    "timeToLive": null,
    "defaultLocaleId": "3",
    "activeTheme": null,
    "loyaltyPrograms": [

    ],
    "paymentOption": "0",
    "additionalProductionURLs": [

    ],
    "links": [
       {
          "rel": "self",
          "href": "http://myserver.example.com:7002/ccadmin/v1/sites/100002"
       }
    ],
    "id": "100002",
    "defaultSite": false,
    "additionalLocaleIds": [

    ],
    "recommendationsHost": "pt-recs-app1.us.example.com",
    "favicon": null,
    "allowedOriginMethods": {

    },
    "noimage": null,
    "defaultPriceListGroup": {
       "deleted": false,
       "repositoryId": "defaultPriceGroup",
       "active": true,
       "id": "defaultPriceGroup"
    },
    "payShippingInSecondaryCurrency": false,
    "siteTypes": [
       "commerce"
    ],
    "recommendationsPort": "8080",
    "shipFromAddress": {

    },
    "repositoryId": "100002",
    "name": "100002"
}
```

Delete a site

You can use the `deleteSite` endpoint to delete an existing site.

For example:

```
DELETE /ccadmin/v1/sites/100002  HTTP/1.1
Authorization: Bearer <access_token>
```

If the site is deleted successfully, a 200-level HTTP status code is returned.

Note that you cannot delete the default site. If you want to delete the site that is currently the default, first make a different site the default.

24

Work with Carts that are Shared Across Multiple Sites

When your environment contains multiple sites, you can configure your instance to share carts among them.

Sharing carts across multiple OSF sites allows your customers to put items into a cart from Site A, as well as from Site B. Shoppers can navigate among sites using a single login and can check out an order that contains items from either site. Shopping carts can be shared across all sites, or a subset of sites.

Note: Shared carts apply only in OSF environments.

For example, if you have a site named MyCompanyOne and another site name MyCompanyTwo, you can allow your shoppers to access both sites without logging off of one site and onto the other site. The shopper can also put an item from MyCompanyOne and an item from MyCompanyTwo into their cart and check out, combining items from two separate sites into a single order.

Customer service representatives can use the Agent Console to assist shoppers with orders that contain items from multiple sites. This allows Agents to perform returns, exchanges and appeasements for these items. Note that to support this on the Agent Console, the Agent should use the On Behalf functionality to work with the order. Refer to the Understand shopping on behalf of a shopper section for additional information.

Understand how carts are shared

Carts can be shared across multiple sites, in both consumer and account-based environments, by using the Admin API to create *cart sharing groups*, which use the sites' ID to identify and include specific sites. When sites are included in a cart sharing group, shoppers can use a single cart among all included sites.

Carts that are shared in multiple site environments work for both registered and anonymous shoppers if the environment uses a single top-level domain. For example, if your domain is `www.example.com` and you also have another site with the same domain and context root, such as `www.example.com/MyCompany`, or `www.MyCompany.example.com`, both registered and anonymous shoppers' carts can be shared among sites. However, only registered shoppers' carts can be shared among sites that do not use the same domain or sub-domain, such as `www.mycompanyone.com` and `www.mycompanytwo.com`. For detailed information on configuring and using domains, refer to Configure Storefront SSO.

The site that the shopper is currently on determines the language, currency, price group and catalog that is displayed. This means that if the shopper changes from SiteA to SiteB, the language, currency, price group and catalogs that are displayed may change, based upon how they are set on SiteB.

The site on which the shopper checks out determines the tax configuration and the payment gateway used in the order. Tax configurations do not change based upon the shopper's current site, unless your site computes and displays tax values prior to checkout. They are set only when the shopper initiates the checkout process.

Note that there are two site properties that are used on the order. These are `createSiteId` and `siteId`. The `createSiteId` property is set to whichever site the shopper was on when the first item was added, and is not updated. The `siteId` property is set every time an item is added to the cart, and again when the order is submitted.

Understand how prices work with carts shared among multiple sites

When you add sites to a cart sharing group, you must ensure that each product for each catalog used on the sites contains, at the minimum, a list price. This means that for every site in the cart sharing group, there must be list prices for all of the products within all of the catalogs used by all of the sites. If a price does not exist, a pricing error will occur. Should you choose to, you can add a sale price across all of the catalogs for sites that share a cart, which allows your shoppers to receive the sales price during checkout. To prevent shopper confusion, it is best to use the same price for products sold across different sites; the price that the shopper pays is based upon where the order was last priced, not upon the site where the item was added to their cart. For example, if you have the same t-shirt on both SiteA and SiteB, but the t-shirt of SiteA costs $25 and the t-shirt on SiteB costs $50, if the shopper checks out on SiteB, the t-shirt will be priced at $50. If the shopper checks out on SiteA, the t-shirt will be $25. This could confuse and discourage your shoppers.

If a shopper adds the same item from two different sites to their cart, the products will show up as different line items within the order. For example, if a shopper puts White Shirt with Ruffles from MyCompanyOne into their cart, and then puts White Shirt with Ruffles from MyCompanyTwo into their cart, their order will display two different line items that contain White Shirt with Ruffles. Additionally, if each site has a different price for the shirt, the price of the White Shirt with Ruffles will come from the site where the order was last priced. For additional information on working with prices, refer to Assign Catalogs and Price Groups to Shoppers.

When a shopper uses shared carts within a multiple site environment, the currency matches the current site every time that the order is priced. For example, if a shopper adds an item to their cart on SiteA, which uses US Dollars, and then adds items to their cart from SiteB, which uses Pesos, both items will be priced in Pesos whenever the shopper views their cart on SiteB. The system does not retain currency context as the shopper moves from one site to the other.

Working with shared carts has several implications on different aspects of your storefront. The following are some things you should keep in mind when configuring shared carts.

Security and carts shared among multiple sites

Your shoppers do not need to re-enter credentials when moving from site to site or when adding items to their cart. For information on setting security for shoppers, refer to Implement Storefront Single Sign-On. Note for GDPR: Although it appears that shopper information is shared across multiple sites, there is no change to the physical location of a shopper's data nor the organizations that have access to that data. For information on configuring GDPR, refer to Manage the Use of Personal Data.

Understand inventory and carts shared among multiple sites

Inventory is decremented from the site that the shopper was on when the item was added to the cart. Buy Online Pay In Store inventory is obtained from locations set in the `locationId` property of the `InStorePickUpShipping` Group. For information on configuring inventory settings, refer to Manage Multiple Inventory Locations.

Work with promotions in carts shared among multiple sites

Site-specific promotions, as well as promotions that apply to any of the sites that share a cart are available to shoppers. You can create item, order and shipping promotions across all sites within a cart sharing group. Additionally, you can create an item promotion that is valid on only a single site within the cart sharing group.

Review order history in carts shared among multiple sites

Whenever a shopper views their order history for sites that are part of the cart sharing group, orders for all of the sites within the group display.

Understand webhooks in carts shared among multiple sites

All order and commerce item site IDs are exposed in order-related webhooks. For information on working with webhooks, refer to Use Webhooks.

Work with cart sharing groups

Cart sharing groups allow you to determine how shopping carts are shared across sites on a single instance of Commerce. Note that all aspects of creating, updating or removing cart sharing groups must be done through the API. For additional information, refer to Use the APIs on instances running multiple sites.

Cart sharing groups are site groups that have been designated as sharing a resource, specifically shopping carts. Site groups that are identified as cart sharing groups use a Boolean property in their response and request payloads. This property indicates if the site group is a cart sharing group. Note that a site must exist before it can be added to a site group or a cart sharing group. Sites can be a member of only one cart sharing group. Because sites are publishable assets, site groups are also publishable. However, a site group cannot be published without its member sites also being published.

To work with cart sharing groups, use the `siteGroups` API. For detailed information on this API, refer to the REST API documentation.

Create a cart sharing group

To create a cart sharing group issue a `POST` command in the `siteGroups` API. The following example creates a cart sharing group that contains three sites:

```
POST  /ccadmin/v1/siteGroups

{
  "id": "cartGroupUSSites",
  "displayName": "Cart sharing group for US Sites",
  "sites":[ {"id": "siteUSSouth"}, {"id": "siteUSWest"}, {"id":
siteUSEast} ],
  "sharedCart" : true
}
```

The request uses the following properties:

Property	Description
`sites` (required)	All elements in the site array must contain a site `id` property. Note that sites must exist before they can be added to a group. This property is an array of sites that make up the cart sharing group.

Property	Description
`id` (optional)	When creating a cart sharing group, the `id` property is optional.
`displayName` (optional)	If you do not provide a `displayName`, the group's `displayName` is set to the group's ID.
`sharedCart` (optional)	This property, when set to `true`, indicates that the site group is a cart sharing group. This property defaults to `false`.

The response to this request would be:

```
POST  /ccadmin/v1/siteGroups

{
  "id": "cartGroupUSSites",
  "displayName": "Cart sharing group for US Sites",
  "sites":[
    {"id": siteUSSouth, "name": "Southern US site"}
    {"id": siteUSWest, "name": "Western US site"}
    {"id": siteUSEast, "name": "Eastern US site"}
  ],
  "sharedCart" : true
}
```

Update a cart sharing group

You can update a cart sharing group by issuing a `PUT` command such as:

```
PUT  /ccadmin/v1/siteGroups/{id}
```

Use the site group's ID to indicate which group to update.

The request and the response are the same as when creating the cart sharing group. However, note the following:

- If you specify the `id` property in the request body, it is ignored.
- If you omit a property from the request body, the old value of the property is retained.
- When a sites array is presented in the request body, the new value totally replaces the old value. Incremental additions and deletions within the site array are not supported.
- If the `sharedCart` property in the request is different than the group's old settings, the group changes from a site group to a cart sharing group, and vice versa.

Delete a cart sharing group

You can delete a cart sharing group by issuing a `DELETE` command using the following syntax:

```
DELETE  /ccadmin/v1/siteGroups/{id}
```

Note that a group cannot be deleted if it contains any sites.

Get cart sharing groups

You can get a single cart sharing group by issuing a `GET` command using the following syntax:

```
GET  /ccadmin/v1/siteGroups/{id}
```

You can get a collection of cart sharing groups by issuing a `GET` command using the following syntax:

```
GET  /ccadmin/v1/siteGroups/
```

The get collection endpoint supports standard optional query parameters, such as `offset` and `limits`. It also supports the `q` query parameter that allows you to filter site groups by the `dislayName`, `id` and `sites` properties.

25

Work with Loyalty Programs

Commerce provides Admin APIs you can use to set up and manage a loyalty points program for your store.

The topics in this section describe how to configure loyalty program features.

Implement loyalty points

If your store supports a loyalty program that lets shoppers earn and spend points, use the following steps to set up a loyalty program using the Commerce Admin API:

1. Define new custom currency specifically for the loyalty program as defined in the Create a custom currency for loyalty points section.
2. Set loyalty program details at the site level as described in the Configure a site to use loyalty programs section. This includes setting the primary currency for the site and linking price list group to sites so that you may display a catalog.
3. Configure a payment integration to provide a payment method shoppers can use for points-based orders at checkout. Refer to the Integrate with a Loyalty Point Payment Gateway section.
4. If you are using Avalara AvaTax or Vertex O Series tax processors, you must configure an exchange rate between your loyalty program currency and the secondary monetary currency of your site. Exchange rates are described in the Create exchange rates section.
5. Define how taxes are calculated and configure the tax settings at the price list group level as described in the Understand tax and shipping calculations with loyalty programs section.
6. Determine if the loyalty program will allow a mixed currency for taxes and shipping s described in the Display tax and shipping in currency for points-based orders section.
7. Update your widgets to use the loyalty program specific widgets to display information to your shoppers, as described in the Use loyalty-specific widgets section.
8. Add loyalty program properties to shopper profiles that let you indicate if a shopper is enrolled in your loyalty program and specify their loyalty account ID. Refer to Redeem loyalty points for this information. See Manage Shopper Profiles for details about how to create and set custom properties for shopper profiles.
9. Update or create email templates that contain loyalty program properties.

Create a custom currency for loyalty points

You can create a custom currency and assign it to a price group so shoppers can see items priced in points and then use points to pay for purchases on your store.

This section describes how to create a custom currency and add it to a price group. You will also need to perform other tasks so shoppers can place orders paid for with points. For example:

You must use the Commerce REST API to create a custom currency for the points. Merchandisers then create a price group for the currency in the Commerce administration interface. Once the price group is active, shoppers can select the points-based currency to see the prices for all items in points. All items in an order must be priced in a single currency. Therefore, a shopper must use points to pay for an entire order. A shopper cannot use points to pay for only some items in an order.

The Avalara AvaTax and the Vertex O Series tax processor integrations will convert points-based orders into monetary currency to calculate tax. If you implement a points-based currency for your store and you use an external Commerce tax processor integration (other than Avalara AvaTax or Vertex O Series) to calculate taxes, you must convert the dollar amount into points and then store that conversion information in Oracle Commerce. See Configure Tax Processing for more information.

You can configure your site to display tax and shipping amounts in a monetary currency, separate from the loyalty points used for an order. For information, refer to the Understand tax and shipping calculations with loyalty programs section

To create a new currency for loyalty points, issue a `POST` request to the `/ccadmin/v1/currencies` endpoint on the administration server. Specify the values of the currency properties as a JSON map in the body of the request.

The following table describes the body of the request.

Property	Description
`id`	(Required) A string that specifies the ID for the currency. This value must be lowercase letters and cannot include underscores or spaces. This value cannot be a valid JDK locale, such as `"en_US"`.
`currencyCode`	(Required) A string that represents the pointType of the loyalty program. This value cannot be an ISO 4217 standard currency code, for example, `"USD"`.
`symbol`	(Required) A Unicode string that specifies the currency symbol. When shoppers select the points-based currency on your store, prices are displayed with this currency symbol.
`displayName`	(Required) A string that specifies the display name for the currency. Shoppers do not see the display name, but it appears internally, for example, in the Commerce administration interface and when you export products.
`currencyType`	(Required) `"loyaltyPoints"` is the only accepted value for `currencyType`. Any other value will cause the request to return an error.
`fractionalDigits`	A number that specifies the precision specifier (the number of decimal places) in the prices shoppers see on your store. The default value is 2.

The following example shows a sample request for creating a loyalty points currency:

```
POST /ccadmin/v1/currencies HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>
```

```
{
"id": "points",
"currencyCode": "PTS",
"symbol": "P",
"displayName": "Frequent Shopper Bucks",
"currencyType": "loyaltyPoints",
"fractionalDigits": 0
}
```

If the currency is created successfully, the response body returned includes the ID for the new currency and a link to the URL used in the request. The following is a sample response:

```
{
  "currencyType": "loyaltyPoints",
  "symbol": "P",
  "deleted": false,
  "displayName": "PTS",
  "repositoryId": "points",
  "fractionalDigits": 0,
  "links": [
    {
      "rel": "self",
      "href": "http://myserver.example.com:7002/ccadmin/v1/currencies"
    }
  ],
  "currencyCode": "PTS",
  "numericCode": null,
}
```

Configure a site to use loyalty programs

The following sections describe how to configure your sites to recognize and use loyalty programs.

Associate a loyalty program's secondary currency to a site

You can use Admin endpoints to define the secondary currency for a site by setting the `secondaryCurrency` property.

To associate secondary currency to a site, issue a `PUT` request to the `/ccadmin/v1/sites/{site_id}` endpoint on the administration server. The following example identifies the secondary currency of the Site US as US Dollars:

```
PUT/ccadmin/v1/sites/siteUS
{
  "properties": [
    {
      "secondaryCurrency": "USD",
    }
}
```

Associate loyalty programs with a site

Loyalty programs can be associated with specific sites. To see which loyalty programs are associated with a site, issue a `GET` request using the `/ccadmin/v1/sites/{site_id}` endpoint on the administration server. The response may be similar to the following:

```
{
  "...": "...",
  "loyaltyPrograms": [
    {
      "repositoryId": "siteUS_pg100001",
      "programId": "pg100001",
      "programName": "Premier",
      "associationDate": "2017-07-05T14:15:37.000Z"
    },
    {
      "repositoryId": "siteUS_pg100002",
      "programId": "pg100002",
      "programName": "Insider",
      "associationDate": "2017-07-05T14:15:37.000Z"
    }
  ]
}
```

To associate a loyalty program with a site, issue a `PUT` request to the `/ccadmin/v1/sites` endpoint on the administration server. For example:

```
{
  "properties": {
    "...": "...",
    "loyaltyPrograms": [
      {
        "programId": "pg100001",
        "programName": "Premier"
      }
    ]
  }
}
```

You can associate a loyalty program with a site at any time by issuing a `PUT` request to `/ccadmin/v1/sites/{site_id}`.

Associate a loyalty program price group with a site

Once your site has been configured to recognize loyalty programs, you can create price groups for the loyalty program and associate the price group with the site.

Create a price group for loyalty points

Once you create a currency for the loyalty points, merchandisers can use the Commerce administration interface to create and activate a price group for the currency. A price group is a set of price lists (list price, sale price, and shipping surcharge), in a specific currency, for the products, SKUs, and shipping surcharges in

a catalog. Creating a price group for the points-based currency lets you price catalog items in the points so a shopper can select the points-based currency from a list of supported currencies and see those prices on your store.

For details about creating and activating price groups, see Configure Price Groups.

Associate the loyalty points price group

To associate the loyalty program price group with a site, issue a `PUT` request using `/ccadmin/v1/sites/`

`{site_id} .`

For example:

```
PUT /ccadmin/v1/sites/siteUS
{
  "properties": {
  "priceListGroupList": [
    {
      "active": true,
      "id": "defaultPriceGroup"
    },
        {
      "active": true,
      "id": "loyaltyPoint"
    }
  ]
  }
}
```

Enable a loyalty payment gateway

If you are going to integrate with a gateway for paying with loyalty points., you must configure your loyalty point programs as described in the Integrate with a Loyalty Point Payment Gateway section.

Understand tax and shipping calculations with loyalty programs

Oracle Commerce can be configured to provide shipping and tax calculations in both loyalty points or in monetary currency.

Understand tax calculations with loyalty points

The taxCalculationType property of the loyalty points price list group determines how tax calculation is performed.

There are three different types of conversion for the Avalara and Vertex tax processor integrations. Depending on the configuration made in the price list group, an administrator can configure the price list group to perform the conversion, skip the tax call entirely, or make the tax call without performing any currency conversions.

Whenever an order contains a combination of loyalty points and monetary currency, the `secondaryCurrencyCode` and `ExchangeRate` are added to the order.

For details additional about creating and activating price groups, see Configure Price Groups for additional information on working with price list groups.

For information on configuring tax processor integrations, refer to Configure Tax Processing and Configure Tax Processors.

Understand shipping calculations with loyalty points

Shipping is calculated in the same currency as the order. In loyalty points-based orders, shipping is calculated in loyalty points. The `CloudShippingPriceInfo` stores the shipping value that has been converted to a loyalty-based order.

The shipping price is determined for each shipping method by calling pre-calculators, applying promotions and then applying any shipping surcharges. For detailed information on setting up shipping methods, refer to Configure Shipping. The final value is converted to the currency amount and set within the `SecondaryCurrencyShippingAmount` property.

How order totals are calculated

When a secondary currency is identified for tax and/or shipping, the `PriceInfo` stores the total shipping/tax amounts that are applied to the order in the `secondaryCurrencyAmount` property. The currency conversion amounts for shipping and tax are also set within the order using the `priceTaxForOrderTotal` and `priceShippingForOrderTotal` property.

Display tax and shipping in currency for points-based orders

When the `isCurrencyConversionEnabled` flag is enabled in the `PriceListGroupManager`, the order is processed in points, and the tax and shipping amounts are converted as per configuration.

When the `isCurrencyConversionEnabled` flag is enabled in the `PriceListGroupManager`, the order is processed in points, and the tax and shipping amounts are converted as per configuration. However, you can configure your site to display tax and shipping in standard currency and not in loyalty points.

To set tax and shipping in currency for a loyalty points-based order, issue a `PUT` request to the `/ccadmin/v1/sites/{site_id}` endpoint on the administration server. Specify if the tax or shipping amounts should be displayed in the currency properties in a JSON map in the body of the request.

Note: When setting the `payTaxInSecondaryCurrency` and `payShippingInSecondaryCurrency` properties, they must both be set either to true or false. One property cannot be true while the other is false, they must both have the same value.

The following table describes properties that determines how tax and shipping are displayed:

Property	Description
`payTaxInSecondaryCurrency`	A Boolean flag that determines if tax should be displayed in the secondary currency.
`payShippingInSecondaryCurrency`	A Boolean flat that determines if shipping should be displayed in the secondary currency.

For example:

```
PUT ccadmin/v1/sites/SiteUS
{
  "properties": {
    "payShippingInSecondaryCurrency": true,
    "payTaxInSecondaryCurrency": true
  }
}
An example response might be similar to the following:
"priceInfo": {
    "secondaryCurrencyTaxAmount": 1.5,
    "amount": 89.97,
    "total": 96.27,
    "secondaryCurrencyShippingAmount": 6,
    "shipping": 0,
    "secondaryCurrencyTotal": 7.5,
    "primaryCurrencyTotal": 89.97,
    "shippingSurchargeValue": 0,
    "tax": 6.3,
    "subTotal": 89.97,
    "currencyCode": "USLoyaltyPoints",
    "totalWithoutTax": 15
  }
```

Use webhooks for loyalty points with monetary currency

When working with webhooks and loyalty programs, it is important to know which webhooks contain secondary currency properties. The following list shows the webhooks that recognize and enable loyalty-points currency conversion.

- CheckOrderApprovalWebhook
- ExternalPricing
- ExternalTaxCalculation
- IdleCart
- Order
- RequestQuote
- Return

Use widgets with loyalty programs with monetary currency

The following widgets can be used to display currency-related properties.

Note: To incorporate loyalty information on your store's pages, make sure you are using the latest version of the widgets.

Specifically, you should update the following:

- shoppingCartSummary
- orderSummary
- orderHistory
- checkoutOrderSummary

- checkoutPaymentDetails
- giftCard
- confirmationSummary
- cartShippingDetails
- checkoutConfirmation
- productDetails
- loyaltyDetails
- loyaltyPayment
- CybersourcePaymentAuthentication
- orderDetails
- ordersPendingApproval
- splitPayments
- accountOrderDetails
- accountCheckoutConfirmation

Redeem loyalty points

Once you have set up a loyalty program on an external system, you allow your shoppers to redeem loyalty points against a program.

To do this, enroll the shoppers into the program. You can enroll members in a loyalty program by providing loyalty details in their profiles using the Commerce Web API.

Use webhooks to configure loyalty point redemption

The Order Submit webhook, which is described in detail in the Order Submit Webhook section, includes loyalty details in the payload. This webhook gets the secondary monetary currency and exchange rate properties. The following example is of a `POST` request that identifies the loyalty programs and membership ID of the profile associated with the order.

```
"loyaltyPrograms": [
{
  "programId": "program1",
  "membershipId": null,
  "programName":"XTRAMILES",
  "status":"RequestForEnrollment"
},
{
  "programId": "program2",
  "membershipId": null,
  "programName":"XTRAREWARDS",
  "status":"RequestForEnrollment"
}
```

The Profile webhook contains loyalty program nomination in the payload. Loyalty membership details are stored in the profile properties and include the program and membership ID.

Use endpoints to configure loyalty point redemption

Endpoints allow you to configure loyalty programs for a site and are used during nomination to loyalty flow for validation. During nomination, all programs that are linked to the profile are included in the payload of Register/Update shopper webhook.

The following properties are used by the profile endpoint to recognize and redeem loyalty point data.

Property	Description
`programId`	The ID of the loyalty program.
`programName`	The name of the loyalty program.
`membershipId`	The ID that indicates the shopper is a member of the loyalty program.
`status`	Indicates the status of the enrollment within the loyalty program. The status can be: Enrolled – The profile has been successfully enrolled and has a membership number. This membership number can be saved in Oracle Commerce along with the loyalty program ID. Unenrolled – Indicates that the profile has been successfully removed from the loyalty system. Failed – Indicates that the loyalty system failed to enroll a program.

To create a new profile with loyalty data, issue a `POST` request in `/ccadmin/v1/profile`. For example:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "profileType": "b2b_user",
  "roles": [
    {
      "function": "buyer",
      "relativeTo": {
        "id": "900004"
      }
    }
  ],
  "receiveEmail": "yes",
  "active": true,
  "parentOrganization": "900004",
  "email": "jdoe@example.com",
  "daytimeTelephoneNumber": "212-555-1977",
  "loyaltyPrograms": [
  {
    "programId": "program1",
    "membershipId": null,
    "programName":"XTRAMILES",
    "status":"RequestForEnrollment"
  },
  {
```

```
    "programId": "program2",
    "membershipId": null,
    "programName":"XTRAREWARDS",
    "status":"RequestForEnrollment"
  }
 ]
}
```

You can add loyalty programs at any time to an existing profile by issuing a `PUT` request in `/ccadmin/v1/profile/{profile_id}`.

Understand enrollment into loyalty programs

When shoppers are enrolled into loyalty programs, Oracle Commerce stores the details of the shopper's enrollment in their profile. There are two ways to configure enrollment. You can decide to enroll all new shoppers, so that when you create a profile it triggers the profile webhook sending the enrollment request to an integrated loyalty system with the program name associated with your site. Or you can create a dynamic property for a profile.

Create custom properties

For example, you could create custom properties for the order, which allows the order to determine accrued points, point types and the loyalty transaction ID:

```
PUT /ccadmin/v1/orderTypes/order
{
  "properties": {
    "loyalty_transaction_id": {
      "internalOnly": true,
      "label": "Loyalty transaction Id",
      "type": "shortText",
      "uiEditorType": "shortText"
    },
    "loyalty_points_accrued": {
      "internalOnly": false,
      "label": "Loyalty points accrued",
      "type": "number",
      "default":0,
      "uiEditorType": "number"
    },
    "accrued_point_type": {
      "internalOnly": false,
      "label": "Point type",
      "type": "shortText",
      "uiEditorType": "shortText"
    }
}
```

You could also create custom properties for the profile, which allows the shopper to enroll in a loyalty program. For example:

```
PUT /ccadmin/v1/shopperTypes/user
{
  "properties": {
```

```
    }
    "enroll_to_loyalty": {
      "internalOnly": false,
      "label": "Enroll for Loyalty program",
      "type": "checkbox",
      "uiEditorType": "checkbox"
    }
  }
}
```

Once the loyalty details widget is placed on the profile layout (storefront), a shopper has the ability to see their loyalty details by accessing My Account and seeing the display of their loyalty status. This information is stored on the shopper's profile in the properties you created.

Note: When you create custom properties, make sure that your property name contains an underscore (_). For example, you could create a custom property named `CustomerProfile_PropertyName`. Once you have created this property, you must customize the integration code and replace any references with the new custom property name.

Use loyalty-specific widgets

There are two loyalty-specific widgets associated with loyalty programs, the Loyalty Details widget and the Loyalty Payment widget.

The Loyalty Details widget, when added to a layout, displays the loyalty details of a registered shopper. This widget displays only one loyalty program detail per shopper, but can be modified to display all of a shopper's loyalty programs memberships.

When the Loyalty Details widget is placed on a page layout that is accessible by registered shoppers, it gets the shopper loyalty information and populates the loyalty view model. Additionally, you can add the logic within the Loyalty Details widget to a global widget so that the loyalty view model is populated with the program details.

For additional information on the Loyalty Details widget, refer to Appendix: Layout Widgets and Elements.

The Loyalty Payment widget displays the payment method as loyalty points when the gateway is enabled. For detailed information on the Loyalty Payment widget, refer to the Appendix: Layout Widgets and Elements. For information on gateways, refer to Integrate with a Loyalty Point Payment Gateway.

Understand currency exchange rates

In instances where you have configured currency for a loyalty program, you may need to convert the loyalty program's currency to a secondary monetary currency.

To do this, you must set up and maintain exchange rates between the loyalty program's points and the secondary monetary currency.

The primary reason for creating an exchange rate is to set conversion rates between loyalty programs and the secondary currency. The exchange rate can also be used during tax calculations with Avalara AvaTax or Vertex O Series. This allows you to send order information to the tax processor in a format that the processor can use to calculate the tax. The tax value is then converted back into loyalty points and displayed in the order.

For information on configuring tax gateways, refer to Configure Tax Processors and Configure Tax Processing.

The secondary monetary currency is configured with the `secondaryCurrency` property when setting up a site. This indicates the currency that the site uses, such as USD. For information on creating a base currency and assigning it to a site, refer to the Associate a loyalty program price group with a site section.

Note: Exchange rates do not convert between different monetary currencies. For example, you should not configure an exchange rate between US dollars and Euros.

Exchange rates are set using the `exchangerates` endpoint to identify a source currency and target currency. Exchange rates are global and not site specific. There should be at least one exchange rate configured with the target currency identified in the `secondaryCurrency` property. Note that exchange rate settings are used only with sites that have loyalty points indicated in their price list group. For information on working with price list groups, refer to the Configure Price Groups.

Create exchange rates

When you create a site that supports loyalty points, you must create an exchange rate. You can create multiple exchange rates, but the site will determine which configuration to use when converting currency.

To create an exchange rate between currency and loyalty points, issue a `POST` request to the `/ccadmin/v1/exchangerates` endpoint on the administration server. Specify the values of the currency properties as a JSON map in the body of the request.

The following table describes properties in the body of the request.

The following example shows a sample request for creating an exchange rate between a monetary currency and the loyalty program's currency that you created in the Implement loyalty points section:

```
POST /ccadmin/v1/exchangerates HTTP/1.1
Authorization: Bearer <access_token>

{
    "sourceCurrency": "PTS",
    "targetCurrency": "USD",
    "exchangeRate": 5
}
```

If the exchange rate is created successfully, the response body returned includes the ID for the new currency and a link to the URL used in the request. The following is a sample response.

```
{
  "sourceCurrency": "USD",
  "targetCurrency": "PTS",
  "exchangeRate": 5,
  "id": "10001"

}
```

The price of the points-based order is converted to the secondary monetary currency for the site, using the corresponding exchange rate. The value that the tax processor calculates is converted from monetary currency back into points using the same exchange rate.

If you are using the External Tax Calculation webhook to integrate with an external tax processor, the tax call is made with or without currency conversion depending on how price list group has been configured. This allows the external tax processor to use the correct logic.

Once the conversion is complete, the exchange rate is recorded and used for subsequent operations, such as returns and exchanges. Once the tax value in monetary currency is returned by the tax processor, it is also stored so that it may be passed to order management systems as part of a Submit and Return Request webhook.

Use custom properties in loyalty integration

The custom properties in loyalty need to follow a convention.

Custom properties used in Oracle Commerce need to follow a naming convention, for example `x_customProperty`). You should change any custom properties you had created in the past to follow this convention. For example:

Older format	New format
`enrollToLoyalty`	`occ_enrollToLoyalty`
`loyaltyTransactionId`	`occ_loyaltyTransactionId`
`loyaltyPointsAccrued`	`occ_loyaltyPointsAccrued`
`accruedPointType`	`occ_accruedPointType`

26

Integrate with External Shipping Calculators

You can integrate your store with external shipping calculators that return shipping methods and costs for an order.

With Oracle Commerce, three shipping method options are available:

- Internally priced shipping method - A shipping method created in the administration interface whose price and availability is determined during checkout using internal rules. Refer to Configure Shipping for information on using this method.
- Externally priced shipping method - A shipping method created in the administration interface whose price and availability are determined using a combination of internal rules and the shipping calculator service. This method is covered in the Work with externally priced shipping methods Work with external shipping methods section this chapter.
- External shipping method – A shipping method returned by the external shipping calculator service that does not have an internal representation in the administration interface.

The price and availability is determined entirely by the service. Refer to the Work with external shipping methods section of this chapter for detailed information on using this method. Refer also to Use Webhooks for additional information on using webhooks and specifically the shipping calculator service.

This chapter focuses solely on providing information related to the externally priced and external shipping methods. Externally priced shipping methods are covered in their own section in the first half of this chapter. This type of shipping method provides you with the best (and preferred) way to use the shipping calculator service and allows you to take advantage of Oracle Commerce's shipping promotion and tax calculation features. External shipping methods are covered in the second half of this chapter. This information is provided to assist customers who may have used this as the original way to integrate with the shipping calculator service. There is finally a section on enabling fallback shipping methods, which applies to both externally priced and external shipping methods.

Work with externally priced shipping methods

Externally priced shipping methods can be created to represent each shipping method that can be returned by your shipping calculator service.

Unlike external shipping methods, creating externally priced shipping methods lets you take advantage of Oracle Commerce's shipping promotion and tax calculation features.

Oracle Commerce determines which externally priced shipping methods are available based on internal rules. The available shipping methods and costs are sent to the shipping calculator service in the request. The shipping calculator service responds with some or all of the available methods and their prices and these available shipping methods are displayed to the shopper with the returned price. This section of this chapter provides detailed information on using this method. Refer also to Use Webhooks for additional information on using webhooks and the shipping calculator service.

With Oracle Commerce, you can create an externally priced shipping method through the administration interface Shipping Methods page available from the Settings list. This is done just like creating an internally priced shipping method except there is a drop-down list selection you can choose to indicate that you are specifying an externally priced shipping method. Also, you have availability to the “applies to shipping methods” picker in shipping promotions like you do with internally priced shipping methods.

Other things to note about this method include:

- You have the ability to indicate that a shipping method is externally priced using both the Admin API and administration interface.
- You have the ability to set all properties for external shipping methods in mostly the same way you can for internal methods. Some of the fallback behavior is different, however. See Enable fallback shipping methods for more information.
- You have the ability to allow externally priced shipping methods to be marked as fallback methods. Externally priced shipping methods marked as fallback require prices. Pricing information is optional if the shipping method is externally priced but not a fallback. See Enable fallback shipping methods for more information.

Externally priced shipping methods behave the same as internally priced shipping methods. Shipping promotions and taxes are applied to externally priced shipping methods the same way as internally priced shipping methods.

The store recognizes the shipping regions associated with externally priced shipping methods as it does with internally priced shipping methods. The store should recognize the shipping regions associated with an externally priced shipping method that is considered unavailable based on internal rules but is returned by the shipping calculator. The service overrides the internal rules in this case and the resulting shipping method/storefront interface behaves the same as all other internally priced shipping methods.

Create externally priced shipping methods

As mentioned, you have the ability to create an externally priced shipping method using the administration interface. For information on how to do this, refer to Configure Shipping.

Configure the Shipping Calculator service for externally priced methods

Oracle Commerce invokes the Shipping Calculator function API to determine the available shipping method list. The list is then populated with a combination of available internal shipping methods and shipping methods returned in the Shipping Calculator response.

When the order is submitted, Commerce invokes the service again to confirm that the selected shipping method and its associated cost are still valid. For an order with multiple shipping groups, Commerce invokes the Web API once for each shipping group.

To send this data to the external shipping calculator service, you configure the service by specifying the URL, username, and password for accessing the service. (See Configure webhooks for details.) You must also configure the external service to read the request data, determine the appropriate shipping methods, and send a response that includes the following items:

- The key `orderIdReceived`, whose value is a string that is the ID assigned to the order by Commerce.
- The `shippingMethods` array, which contains available shipping methods and their associated costs. A sample response is shown in this section of this chapter.

You can configure fallback shipping methods to display if Commerce cannot connect to the external system, for example in the event of an outage. See Enable fallback shipping methods for more information

Understand how the shipping calculator Web API works

The shipping calculator Web API sends the following data to the external shipping calculator service:

- Most details about the order, including all its shipping groups. The request does not include certain payment details, such as credit card information. See Order Submit request example for a sample JSON representation of an order object in a Shipping Calculator service request body.
  Previous versions of this service sent less order data in the payload of the request. Commerce still supports this version of the request by default. To send the detailed order object in the request, you must first use the Admin API to issue a PUT request to `/ccadmin/v1/merchant/clientConfiguration` that sets the `includeOrderDetailsInShippingMethodsPayload` property to true.
- Profile details for the registered shopper who placed the order.
- The service request will also contain the externally priced shipping methods (along with their IDs and other properties) that are available based on internal rules.

Upon receiving the request, the shipping calculator service sends back a response which includes the externally priced shipping methods that are available for the current shopper/cart.

Note: The externally priced shipping methods in the response are identified by the ID sent in the request. Based on this ID, the response provides pricing and additional method information. In this way, the response format is different from what you might be used to when using existing external shipping methods. For more information, refer to the next two sections which describe the externally priced shipping methods response and request formats in more detail.

Understand the externally priced shipping methods shipping calculator service request

In more detail, the shipping service request provides the following information to the shipping calculator service:

- The externally priced shipping methods eligible based on internal rules in the request. These will include the shipping method ID (see example) to identify them and other properties including the internal price if defined.
- The complete cart/order definition if the full order capability has been requested.
- The shopper profile
- Shipping related properties for each item in the order such as dimensions and weight.

The following presents an example of a shipping calculator service request sent for externally priced shipping methods. The shipping method ID is displayed in bold in the example.

```
"availableExternallyPricedShippingMethods" : [ {
      "eligibleForProductWithSurcharges" : false,
      "ranges" : [ ],
      "associatedPriceListGroups" : [ {
        "id" : "defaultPriceGroup"
      } ],
      "displayName" : "Example External Shipping Method",
      "description" : "Example External Shipping Method",
      "allSites" : true,
      "sites" : [ ],
      "shippingMethodId" : "100001",
      "excludedCategoriesShippingCharge" : [ ],
      "isFallback" : false,
      "taxCode" : "100",
      "shippingGroupType" : "hardgoodShippingGroup"
```

Understand the externally priced shipping methods shipping calculator service response

Upon receiving the request, the shipping calculator service sends back a response which includes the externally priced shipping methods that are available for the current shopper/cart. These are identified by the ID received in the request.

The following example shows a shipping calculator service response for externally priced shipping methods. Again, the same shipping method ID is displayed in bold in the example.

```
"shippingMethods": [{
         "eligibleForProductWithSurcharges": false,
         "estimatedDeliveryDateGuaranteed": false,
         "displayName": "Example External Shipping Method",
         "shippingTax": 2,
         "currency": "USD",
         "shippingCost": 12.95,
         "shippingMethodId":"100001"
         "internationalDutiesTaxesFees": 0,
         "estimatedDeliveryDate": "2018-02-02 14:48:45 -0400",
         "shippingTotal": 14.95,
         "deliveryDays": 2,
         "taxcode": "100",
         "carrierId": "ON",
     }]
```

If a shipping method is returned in the response whose ID matches the one in the request then:

- The properties returned in the response override the properties modeled internally. For example, the price and tax code returned by the shipping calculator will be used instead of the price and tax code modeled internally.

- Oracle Commerce applies shipping promotions and calculate taxes for these shipping method and that generally behave the same as internal shipping methods except that they use the returned price and properties.

If a shipping method is returned in the response has an ID that does NOT match that in the request, or that does not have an ID then:

- This shipping method is displayed with the name, price, tax code, and other properties as returned in the response.
- Oracle Commerce treats these shipping methods as external shipping methods. Shipping promotions will not be applied. Other limitations may apply as described in the next section concerning external shipping methods.

A response may include a combination of externally priced and external shipping methods. Also, the shipping calculator may return a 400 error to indicate that the shipping address is invalid.

The following example shows an error response that might be returned when there is an invalid shipping address:

```
{   "errorCode": "00000000",
          "message": "Invalid shipping address",   "errors": [
            {       "errorCode": "28128",
              "message": "Update unsuccessful. Shipping address is not
                a valid ship-to address.",       "status": "400"
            }   ],   "status": "400" }
```

Understand the contents of an order payload

If the shipping calculator service is triggered because an order is placed, a scheduled order is instantiated, or an order is approved, the order payload also includes details about the order and its shipping methods. See Order Submit request example for a sample JSON representation of an order in a webhook body.

The payload does not include certain payment details, such as credit card information. See Understand webhooks and PCI DSS compliance for information about payment details that are excluded from the request.

Some examples of the information that you find in the payload associated with this shipping method include

- Promotion information
- Shopper profile information
- Product level information (for example length, width, weight, etc.)
- Available externally priced shipping methods information

For more complete information and examples of the payload associated with this shipping method, refer to Learn about the APIs.

Upgrading from external shipping methods to externally priced shipping methods

If you have previously used external shipping methods and want to take advantage of externally priced shipping methods, then you should re-implement the functionality following

the procedure described in the previous Work with externally priced shipping methods section of this chapter.

You need to create an externally priced shipping method for each shipping method returned by your shipping calculator service. You must also update your shipping calculator service to receive the available externally priced shipping methods in the request and then return a subset of these methods as identified by the `shippingMethodId` in the response. You can continue to use existing fallback shipping methods or mark the externally priced shipping methods as fallback.

Work with external shipping methods

Unlike externally priced shipping methods, there is no need to create an internal representation for an external shipping method. This step can be omitted.

However, you may need to create a dummy internal shipping method if you plan to use external shipping methods only. This method is described in Configure Shipping. This step applies only to external shipping methods and is not needed for externally priced shipping methods.

Note: You can use external shipping methods if you do not need to take advantage of Oracle Commerce shipping promotion functionality. Using externally priced shipping methods, however, provides you with the best way to use the shipping calculator service and allows you to take full advantage of Oracle Commerce's shipping promotions and tax calculation features.

Configure the Shipping Calculator service for external shipping methods

When the shopper selects the shipping methods list in the shopping cart, Commerce invokes the Shipping Calculator service. The list is then populated with the shipping methods returned in the service response. Then, when the shopper submits the order, Commerce invokes the service again to confirm that the selected shipping method and its associated cost are still valid. For an order with multiple shipping groups, Commerce invokes the service once for each shipping group.

The service sends the following data to the external shipping calculator:

- Most details about the order, including all its shipping groups. The request does not include certain payment details, such as credit card information. See Understand webhooks and PCI DSS compliance for information about payment details that are excluded from the request. See Order Submit request example for a sample detailed representation of an order object in a Shipping Calculator service request body.
  Previous versions of this service sent less order data in the payload of the request. Commerce still supports this version of the request by default. To send the more complete order representation in the request, you must first use the Admin API to issue a `PUT` request to `/ccadmin/v1/merchant/clientConfiguration` that sets the `includeOrderDetailsInShippingMethodsPayload` property to true.
- Profile details for the registered shopper who placed the order. See Manage Shopper Profiles for a sample JSON representation of a shopper profile.

To send this data to the external shipping calculator service, you configure the service by specifying the URL, username, and password for accessing the service. (See Configure webhooks for details.) You must also configure the external service to read the request data, determine the appropriate shipping methods, and send a response that includes the following items:

- The key `orderIdReceived`, whose value is a string that is the ID assigned to the order by Commerce.
- The `shippingMethods` array, which contains available shipping methods and their associated costs.

The following sample shows a response with two available shipping methods:

```
{
  "orderIdReceived": "o460411",
  "shippingMethods": [
    {
      "shippingCost": 12.95,
      "shippingTax": 2.00,
      "shippingTotal": 14.95,
      "internationalDutiesTaxesFees": 0,
      "eligibleForProductWithSurcharges": true,
      "deliveryDays": 2,
      "estimatedDeliveryDateGuaranteed": false,
      "estimatedDeliveryDate": "2013-04-12 14:48:45 -0400",
      "displayName": "canadapost-overnight",
      "carrierId": "ON",
      "taxcode": "GT987",
      "currency": "USD"
    },
    {
      "shippingCost": 29.00,
      "shippingTax": 4.00,
      "shippingTotal": 33.00,
      "internationalDutiesTaxesFees": 0,
      "deliveryDays": 2,
      "estimatedDeliveryDateGuaranteed": false,
      "estimatedDeliveryDate": "2013-04-12 14:48:45 -0400",
      "displayName": "fedex-2dayground",
      "carrierId": "1D",
      "taxcode": "TD543",
      "currency": "USD"
    }
  ]
}
```

You can configure fallback shipping methods to display if Commerce cannot connect to the external system, for example in the event of an outage. See Enable fallback shipping methods for more information;

If no shipping methods are available based on the shipping address, the response returns an error. The following sample shows an error response that might be returned for an invalid shipping address.

```
{
  "errorCode": "00000000",
  "message": "Invalid shipping address",
  "errors": [
    {
      "errorCode": "28128",
```

```
      "message": "Update unsuccessful. Shipping address is not
        a valid ship-to address.",
      "status": "400"
    }
  ],
  "status": "400"
}
```

Implement the shipping calculator service for external shipping methods

In comparison to an externally priced shipping method, an external shipping method is a method returned by the external shipping calculator service that does not have an internal representation in the administration interface.

To set up the external shipping method integration with an external shipping calculator, perform the following steps:

1. Create shipping regions that specify where the carrier service will ship to. See Create shipping regions for more information.
2. Your store can offer a combination of internal and external shipping methods. If your store uses only shipping methods returned from an external service, create a shipping method and associate it with the shipping regions you created in the previous step. See Create a shipping method for more information.

   Keep the following points in mind as you create the shipping method:

   - Give the shipping method an appropriate name, such as the descriptive name of a specific Region.
   - By default, Commerce assumes that shipping prices received from the external service include all applicable taxes. Therefore, any tax value calculated by your integrated tax processor (based on the shipping method's Tax Code property) is automatically overridden with 0 when the shipping price is displayed to the shopper.

   However, you can configure Commerce so it uses your integrated tax processor (Avalara, Vertex, or an external tax processor) to calculate applicable shipping taxes for external shipping methods. This is the preferred method for calculating shipping taxes because shipping taxes are calculated the way you intended. See Calculate taxes for external shipping methods shipping calculator service for more information.
3. If you want to offer only shipping methods returned in the Shipping Calculator service response, customize the Order Summary widget JavaScript to hide the shipping method (based on its repositoryId property) you created in the previous step. See Developing Storefront Classic Widgets for Oracle Commerce for more information about customizing widgets.
   Note: There is no way to hide this shipping method on the Agent Console. You must instruct Agent Console users to not select this shipping method.
4. Configure the Shipping Calculator service as described in Configure the Shipping Calculator service for external shipping methods section of chapter.

Once you have performed the steps in this procedure, the service is ready to use.

Understand the external shipping methods shipping calculator service request

With external shipping methods, you need to know the following information about the service request:

- The request includes the complete cart/order definition
- The request includes the shopper profile
- The request includes shipping related properties for each item in the order such as dimensions and weight.

Understand the external shipping methods shipping calculator service response

With external shipping methods, you need to know the following information about the service request:

- The response includes external shipping methods identified by display name. The price, tax code, and all properties are displayed as returned by the shipping calculator (since there is no internal representation).
- Alternately the response may return a 400 error indicating an invalid shipping address (existing)

Oracle Commerce will not apply shipping promotions to these shipping methods.

Calculate taxes for external shipping methods shipping calculator service

By default, Commerce requires that shipping prices received from the external service include all applicable taxes. However, you can configure Commerce to use your integrated tax processor (Avalara, Vertex, or an external tax processor) to calculate applicable shipping taxes for shipping methods returned by the external shipping methods shipping calculator service. This is the preferred method for calculating shipping taxes because shipping taxes are calculated the way you intended.

You configure the Shipping Calculator webhook response to specify where shipping taxes are calculated by configuring the shipping system to include the appropriate value for the flag `taxIncluded` in the webhook response. The value should be one of the following:

- true: Shipping taxes must be calculated by the external shipping service. This is the default value.
- false: Shipping taxes are automatically calculated by the tax processor you integrated with Commerce.

The following sample webhook response body might be returned if `taxIncluded` is omitted:

```
orderIdReceived     "o30451"
shippingMethods
0
shippingCost     12.95
shippingTax     2
shippingTotal     14.95
internationalDutiesTaxesFees     0
eligibleForProductWithSurcharges     true
deliveryDays     2
estimatedDeliveryDateGuaranteed     false
estimatedDeliveryDate     "2018-09-12 14:48:45 -0400"
displayName     "canadapost-overnight"
carrierId     "ON"
taxcode     "GT987"
currency     "USD"
```

If `taxIncluded` is set to false, the response body might be:

```
orderIdReceived     "o30451"
shippingMethods
0
shippingCost     12.95
shippingTax     0
shippingTotal     12.95
internationalDutiesTaxesFees     0
eligibleForProductWithSurcharges     true
deliveryDays     2
estimatedDeliveryDateGuaranteed     false
estimatedDeliveryDate     "2018-09-12 14:48:45 -0400"
displayName     "canadapost-overnight"
carrierId     "ON"
taxcode     "GT987"
taxIncluded     "false"
currency     "USD"
```

See Configure Tax Processing for information about integrating with a tax processor.

Enable fallback shipping methods

You can designate internal shipping methods as fallback by selecting the fallback option.

You can also designate externally priced shipping methods as fallback. If the shipping calculator fails (no response or invalid response), then any internal and externally priced shipping methods that are eligible based on internal rules are shown to the shopper. In this case, the internal prices defined in the shipping methods will be used.

Keep in mind that internal shipping methods marked as fallback are only available to the shopper if the shipping calculator fails. Externally priced shipping methods can be available to the shopper if the shipping calculator fails (using internal prices) or if the shipping method is available based on internal rules and is returned by the shipping calculator (as identified by an ID). In case of the latter, the price and properties returned by the shipping calculator will be used.

Understand more about fallback shipping methods

As mentioned, Oracle Commerce lets you offer fallback shipping methods to shoppers when it cannot connect to your external shipping calculator's web service, for example, in the event of an outage. The fallback logic displays internally priced shipping methods (that you can specify either in the administration interface or with the Admin API) to all orders if a specified number of service calls to the external shipping calculator fail within a specified time span. This prevents errors at the shipping calculation step of order processing and allows orders to progress to the payment processing step.

During the initial configuration of your environment, Oracle sets certain settings that trigger the use of fallback shipping methods, including the number of consecutive failed service calls, the time span over which to count failed service calls, and the time period after which Commerce should try calling the shipping calculator's service again.

Commerce uses the fallback logic to calculate shipping only when calls to the external shipping calculator fail. That is, when the external shipping calculator's web service responds to a call with a 500-level status code. Fallback shipping calculation is not used when any other type of response is received.

The `shippingMethod` property on Commerce order objects specifies the shipping method selected by the shopper when they created the order. This property is also included in the data Commerce sends in the Order Submit webhook. You can use this information to identify orders submitted with a fallback shipping method so you can decide how to handle them in your order management system.

Specify a fallback shipping method

You can mark any number of internally priced shipping methods as fallbacks. You might want to create a fallback method for each type of shipping method you expected to receive back from the external shipping service. This section describes how to use the Admin REST API to specify fallback shipping methods. To learn how to use the administration interface for this task, see Configure Shipping.

To specify that a shipping method can be used as a fallback, set its `isFallback` property to true. The following example marks an existing shipping method as a fallback.

```
PUT /ccadmin/v1/shippingMethods/standardShippingMethod HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
  "isFallback": true
}
```

Disable fallback shipping

The ability to use fallback shipping methods is enabled by default, but you can disable it. If you disable fallback shipping and Commerce cannot reach your shipping calculator's web service, shoppers may see errors during checkout and may not be able to complete their orders.

You use the `Commerce` Admin API to set the `fallbackEnabled` property, a Boolean that specifies whether fallback shipping methods are used. The default value for `fallbackEnabled` is true.

To set the `fallbackEnabled` property, issue a `PUT` request to the `/ccadmin/v1/merchant/fallbackShippingConfiguration` endpoint.

The following example shows a `PUT` request that disables fallback shipping.

```
PUT /ccadmin/v1/merchant/fallbackShippingConfiguration HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
   "fallbackEnabled": false
}
```

27

Integrate with an External Pricing System

By default, Oracle Commerce stores prices for products and SKUs internally in price lists. When a shopper browses products on your storefront, the internal price is displayed for each product, and this price is applied to the product when it is added to the shopping cart.

Some stores, however, may need to access prices that are maintained in an external system. To enable this, Oracle Commerce includes tools that you can use to build a custom integration with an external pricing system. You create a custom widget that makes a call to the external pricing system to obtain the price of an item when it is added to the shopping cart. You also configure a webhook that calls the external pricing system to validate the prices when the shopper checks out.

Create the widget

To obtain prices from an external pricing system, you write a custom widget to include on your storefront's pages.

The custom widget's JavaScript file extends the storefront's `CartViewModel` class by implementing a prepricing callback function. When a shopper modifies the shopping cart by adding or removing products or changing quantities, this function makes a call to the external pricing system to obtain prices for the items in the cart.

In addition to an `externalPrice` property for storing the external price of a product, the `CartViewModel` has an `externalPriceQuantity` property that determines the maximum quantity that the external price can be applied to. For example, suppose the `list` price of a product is $10.00, but the external pricing system returns an `externalPrice` value of $7.00 and an `externalPriceQuantity` value of 3. If a shopper selects this product and specifies a quantity of 2, the price in the cart for the two items is $14.00, because the external price is applied to both items. However, if the shopper specifies a quantity of 5, the external price is applied to only three of the items, so the total is $41.00 (three items at $7.00 each, and two items at $10.00 each).

If the value of `externalPriceQuantity` for a specific product is -1, there is no maximum quantity, so the external price is applied to all of the items of the product.

Create the widget structure

Create your widget as a global widget. Global widgets are automatically loaded for all pages; you do not need to add them explicitly to page layouts. Global widgets cannot include any user interface elements, so you should omit any display templates. The following shows an example of the files and directories in a global widget:

```
External Pricing/
    ext.json
    widget/
        external-pricing/
            widget.json
```

```
            js/
                external-pricing.js
```

To make the widget global, set the global property in the `widget.json` file to true:

```
"global": true
```

For more information about the widget structure and the contents of the `ext.json` and `widget.json` files, see Create a Widget.

Write the JavaScript

The JavaScript code you write extends the `CartViewModel` class by implementing a callback function that executes during `prepricing`. The following example shows sample JavaScript that implements a prepricing function:

```
define(

  ['jquery', 'knockout', 'ccLogger'],

  function($, ko, CCLogger) {

    'use strict';
    return {

      onLoad: function(widget) {
      CCLogger.info("Loading external pricing widget");
      var callbackMap = new Object();
      var performPrepricing = function()
      {
      // sample code to invoke external system
        $.ajax({
          type: 'POST',
          dataType: 'json',
          url: EXTERNAL_SYSTEM_SERVICE_URL,
          data: widget.cart().items(),
          success: function(data) {
            // update the cart items with external price details,
            // assuming data has item details with external prices
            if (data.items && data.items.length > 0) {
              for (var i = 0; i < widget.cart().items().length; i++) {
                for (var j = 0; j < data.items.length; j++) {
                  if (widget.cart().items()[i].productId ==
                    data.items[j].productId &&
                    widget.cart().items()[i].catRefId ==
                    data.items[j].catRefId &&
                    data.items[i].externalPrice &&
                    data.items[j].externalPriceQuantity) {
                    widget.cart().items()[i].externalPrice
                      (data.items[j].externalPrice);
                    widget.cart().items()[i].externalPriceQuantity
                      (data.items[j].externalPriceQuantity);
                  }
                }
```

```
                }
              // invoke pricing in this success callback
              widget.cart().markDirty();
            }
          },
          error: function() {}
        });
     };
     callbackMap['prepricing'] = performPrepricing;
     widget.cart().setCallbackFunctions(callbackMap);
     }
    }
   }
);
```

Note that because the asynchronous call to the external pricing system may complete after a pricing operation has already taken place on the Oracle Commerce server, the code explicitly marks the cart as having been modified after applying the external price information. This forces another pricing operation to be invoked using the external prices.

Some other considerations to take into account when you create your widget:

- To avoid making unnecessary calls, it is a good idea for your code to check whether external prices have already been applied to the cart items, and if so, skip calling the external system.
- If the external system does not impose a quantity limit on a specific product, your code should set `externalPriceQuantity` to -1, so that the external price is applied to all of the items of the product.

Install the widget

To install the widget, do the following in the administration interface:

1. Click the **menu** icon, then click **Settings**.
2. Click **Extensions** and display the **Developer** tab.
3. Click **Generate ID** to generate an extension ID for the widget.
4. Edit the widget's `ext.json` file and set the `extensionID` property to the value generated in the previous step.
5. Package the widget as a ZIP file. Use the structure described in Create the widget structure.
6. Display the **Installed** tab and click **Upload Extension**. Select the ZIP file.
7. Publish your changes.

Price the cart items

The custom widget alters the logic for pricing the shopping cart. This section describes how the pricing of the cart behaves when external pricing is enabled.

When a shopper views the product detail page for a specific product, the page displays the internal price that Oracle Commerce stores for the product. When the shopper modifies the shopping cart (adds or remove an item, or changes the quantity of an item), the custom widget makes a call to the external pricing service, and applies any external prices it receives

to the items in the cart. Note that this means that the price of a product in the cart may differ from the price displayed on the product detail page.

After obtaining prices from the external pricing service, the storefront sends the current contents of the cart to the Oracle Commerce server to calculate pricing (for example, to apply promotions). This call includes `externalPrice` and `externalPriceQuantity` values for any products in the cart that have external prices.

The following shows an example of the data sent to the server:

```
{
  "shoppingCart": {
    "items": [
      {
        "productId": "xprod1003",
        "quantity": 3,
        "catRefId": "xsku1013",
        "stockStatus": true,
        "discountInfo": [],
        "externalPrice": "21.00",
        "externalPriceQuantity": "1",
        "invalid": false
      },
      {
        "productId": "xprod1002",
        "quantity": 5,
        "catRefId": "xsku1007",
        "stockStatus": true,
        "externalPrice": "18.00",
        "externalPriceQuantity": "-1",
        "invalid": false,
        "currentPrice": 0
      }
    ],
    "coupons": []
  }
}
```

In the example above, the cart contains two products, xprod1003 and xprod1002. The quantity of xprod1003 is 3, but only one of those items will have the external price applied to it ($21.00), because the `externalPriceQuantity` value for xprod1003 is 1. The other two items will have the internal price applied.

The quantity of xprod1002 is 5, and all of these items will have the external price applied ($18.00), because the `externalPriceQuantity` value for xprod1002 is -1.

When the server receives the data above, it performs a pricing operation, using the external prices for any items that have them, and applying internal prices to the rest. The repriced cart data is returned to the storefront for display.

The following example shows part of the data returned to the storefront after a pricing operation:

```
{
  "shoppingCart": {
    "numberOfItems": 8,
```

```
"items": [
  {
    "onSale": false,
    "catRefId": "xsku1013",
    "shippingSurchargeValue": 0,
    "externalPrice": 21,
    "unitPrice": 36,
    "discountAmount": 0,
    "productId": "xprod1003",
    "externalPriceQuantity": 1,
    "rawTotalPrice": 93,
    "price": 93,
    "discountInfo": [],
    "listPrice": 36,
    "detailedItemPriceInfo": [
      {
        "amount": 21,
        "currencyCode": "USD",
        "tax": 0,
        "discounted": false,
        "orderDiscountShare": 0,
        "quantity": 1,
        "detailedUnitPrice": 21
      },
      {
        "amount": 72,
        "currencyCode": "USD",
        "tax": 0,
        "discounted": false,
        "orderDiscountShare": 0,
        "quantity": 2,
        "detailedUnitPrice": 36
      }
    ],
    "salePrice": 0,
    "quantity": 3
  },
  {
    "onSale": false,
    "catRefId": "xsku1007",
    "shippingSurchargeValue": 0,
    "externalPrice": 18,
    "unitPrice": 24,
    "discountAmount": 0,
    "productId": "xprod1002",
    "externalPriceQuantity": -1,
    "rawTotalPrice": 90,
    "price": 90,
    "discountInfo": [],
    "listPrice": 24,
    "detailedItemPriceInfo": [
      {
        "amount": 90,
        "currencyCode": "USD",
        "tax": 0,
```

```
            "discounted": false,
            "orderDiscountShare": 0,
            "quantity": 5,
            "detailedUnitPrice": 18
          }
        ],
        "salePrice": 0,
        "quantity": 5
      }
    ]
  },
  "discountInfo": {
    "orderCouponsMap": {},
    "orderDiscount": 0,
    "orderImplicitDiscountList": [],
    "unclaimedCouponsMap": {},
    "shippingDiscount": 0
  },
  "priceInfo": {
    "amount": 183,
    "total": 183,
    "shipping": 0,
    "totalWithoutTax": 183,
    "currencyCode": "USD",
    "shippingSurchargeValue": 0,
    "tax": 0,
    "subTotal": 183
  },
...
```


The response shows the effect of the external pricing. For example, the `detailedItemPriceInfo` object for xprod1003 shows that one item is priced at $21.00 (the external price) and the other two are priced at $36.00 each (the internal price). The total price of the three items is $93.00.

Configure the webhook

When the shopper clicks **Place Order**, the order is submitted. However, if any item in the order has an external price, the server first invokes the External Price Validation function webhook, which sends the external pricing data to the pricing system for verification.

Invoking the webhook ensures that the external prices have not changed since the last pricing operation and that the prices in the cart have not been modified.

For example, the following shows part of a sample request issued by this webhook:


```
...
"currencyCode" : "USD",
"operation" : "externalPricing",
"externalPrices": [
    {
      "externalPriceQuantity": 1,
      "externalPrice": 21,
```

```
      "catRefId": "xsku1013"
    },
    {
      "externalPriceQuantity": -1,
      "externalPrice": 18,
      "catRefId": "xsku1007"
    }
  ]
...
```

To send this data to the pricing system, you configure the webhook by specifying the URL, username, and password for accessing the pricing system. (See Configure webhooks.) If your environment uses account-based storefronts, you might also specify the account. You must also configure the pricing system to read the external pricing data, verify whether it is valid, and send a response that includes the appropriate response code. The response code should be one of the following:

- 5001 (VALID_EXTERNAL_PRICES)
- 5002 (INVALID_EXTERNAL_PRICES)

For example, if the external price data is valid, the response body might be:

```
{
  "ResponseCode": "5001"
}
```

If the response code is 5001, the order is submitted. If the response code is 5002, the order is not submitted, and an error is displayed on the checkout page. You can write custom logic to correct the error (for example, by removing items from the cart and then putting them back in so that the widget retrieves up-to-date price data from the external pricing system).

Use promotions from an external system

When working with promotions, you may want to use either Oracle Commerce and/or an external system for issuing your promotions.

When a pricing operation is initiated, the regular item and order pricing occurs. After order pricing, the system invokes the External Promotions webhook. This webhook allows an external system that you have configured, as outlined earlier in this chapter, to discount order and item prices determined by Commerce. Commerce receives a request to update pricing to items within the cart and the pricing changes are applied and displayed to the shopper.

The following properties are used for external promotion requests:

Property	Description
`externalCoupons`	Contains the external coupons list.
`order`	Contains the order object, which contains the order ID.
`profile`	Contains the profile repository item that is added to the request as a top-level JSON object.
`serviceName`	Contains the name of the webhook service.
`ExternalPromotionsIds`	Contains the list of external promotion IDs provided during the return process.

The following is an example of a request:


```
{
    "secondaryCurrencyCode": "string",
    "exchangeRate": 0,
    "profile": {},
    "currencyCode": "string",
    "operation": "string",
    "order": {
        "priceInfo": {},
        "discountInfo": {
            "orderCouponsMap": [
                {
                    "promotionLevel": "string",
                    "totalAdjustment": "string",
                    "promotionDesc": "string",
                    "promotionId": "string"
                }
            ],
            "orderDiscount": 0,
            "shippingDiscount": 0
        },
        "shoppingCart": {
            "numberOfItems": 2,
            "items": [
                {
                    "unitPrice": 24.99,
                    "amount": 36.98,
                    "quantity": 2,
                    "detailedItemPriceInfo": [
                        {
                            "discounted": false,
                            "secondaryCurrencyTaxAmount": 0,
                            "amount": 24.99,
                            "quantity": 1,
                            "tax": 0,
                            "orderDiscountShare": 0,
                            "detailedUnitPrice": 24.99,
                            "currencyCode": "USD",
                            "detailedItemPriceInfoID": "String"
                        },
                        {
                            "discounted": true,
                            "secondaryCurrencyTaxAmount": 0,
                            "amount": 11.99,
                            "quantity": 1,
                            "tax": 0,
                            "orderDiscountShare": 0,
                            "detailedUnitPrice": 11.99,
                            "currencyCode": "USD",
                            "detailedItemPriceInfoID": "String"
                        }
                    ],
                    "catRefId": "string",
                    "externalRecurringChargeDuration": "string",
```

```
                    "discountInfo": [
                        {
                            "promotionLongDesc": "<p>Save $13 on Wonderland
OR
                                                  XBOX 360</p>",
                            "promotionName": "Explicit Item Discount -
                                              SAVE13DOLLARS",
                            "promotionLevel": "item",
                            "coupon": "SAVE13DOLLARS",
                            "totalAdjustment": "-13.0",
                            "promotionDesc": "Explicit Item Discount -
                                              SAVE13DOLLARS",
                            "promotionId": "explicitItemAmountDiscount",
                            "giftWithPurchaseDiscountInfo": []
                        }
                    ]
                }
            ]
        },
        "coupons": [
            {
                "code": "",
                "description": "",
                "status": "",
                "level": "",
                "id": ""
            }
        ],
        "shippingAddress": {},
        "siteId": "string"
    },
  "externalCoupons": [
    {
        "code": "EXTERNALCOUPON",
        "description": "",
        "status": "",
        "level": "",
        "id": ""
    }
  ]
}
```


The `orderCouponsMap` includes the `promotionID`, which provides the external system with the ability to differentiate between internal promotions that have been applied to the order and those promotions that have been applied by the external system.

The External Promotions webhook responds with either an error code or a status message. Note that the external system must provide a unique ID for each promotion to be applied. This ID should be unique between both external promotions and internal promotions. External promotion IDs must not clash with internal IDs. If an external promotion uses an ID that matches the ID of an internal item or order promotion that has already been applied to the order, the external promotion is ignored.

The following properties are used for external promotion response:

Property	Type	Description
`adjustmentAmount`	Number	The amount to adjust the target item or order by.
`adjustmentOperation`	String	The external promotion adjustment operation to apply. Values for this property include: - `adjustItemPrice` – Adjust the price of an item - `adjustOrderPrice` – Adjust the price of an order
`adjustmentOrdering`	String	The order of items to target for adjustment based on the `highestFirst` or `lowestFirst` priced.
`coupon`	String	The coupon ID.
`description`	String	A description of the external promotion being applied by the adjustment.
`displayName`	String	The promotion display name.
`id`	String	The ID of the commerce item in the cart to be adjusted.
`promotionId`	String	The external promotion ID.
`quantity`	Number	The quantity of items to be adjusted.

The following is an example of the body of a response:

```
{
  "responseCode":"6101",
  "promotionAdjustments": [
    {
      "adjustmentOperation": "adjustItemPrice",
      "promotionId": "EP1001",
      "description": "$20 discount on XBOX 360",
      "id": "ci6000413",
      "quantity": "2",
      "adjustmentOrdering": "highestFirst",
      "adjustmentAmount": "-20",
      "displayName": "XBOX Forever",
      "coupon": "20DOLLARDISCOUNTXBOX"
    },
    {
      "adjustmentOperation": "adjustOrderPrice",
      "promotionId": "EP1003",
      "description": "$50 off order when total above $300",
      "adjustmentAmount": "-50",
      "displayName": "50DollarDiscount"
    }
  ]
}
```

Once the webhook responds, the operation described in the response, or any errors, are reported back.

Cancel external promotion orders

If an order is cancelled during the remorse period, the Cancellation webhook is triggered. This could allow details of the cancelled orders to be posted to the external pricing system upon cancellation, releasing any promotions that were applied during the order so that they may be reapplied if necessary. The Cancellation webhook contains all of the order-related information, as well as the reason for the cancellation.

If a remorse period is in effect, the Submit Order webhook is not fired until the end of the remorse period. Use the Remorse Period Started webhook to give details of the order to an external system prior to order being submit. For example, you could use the Remorse Period Started webhook to mark promotions to ensure that they are not unknowingly reapplied if a shopper creates another order during the remorse period.

For further information on remorse periods, refer to Understand the remorse period.

28

Customize Email Templates

Oracle Commerce provides FreeMarker templates you can use to customize the emails your store sends to shoppers.

FreeMarker is a Java template engine. You do not need to install FreeMarker in order to edit the template files. For information about FreeMarker templates, see the FreeMarker Manual at freemarker.org.

If your Commerce instance is running multiple sites, you can configure and customize templates on for each site. See Download and edit email templates to learn how to select a site when you download email templates and upload your changes. See Add a site to a template for information about site names and URLs in templates.

Download and edit email templates

Changes you make to your store through the Commerce administration interface do not affect the email templates. If you make changes to your store that you want to see reflected in emails you send to customers, you must manually update the templates.

Examples of changes you might want to reflect in your email templates are as follows:

- If you change a theme's style sheet on the Design page and you want your emails to have the same look and feel, you must manually update each email's style sheet in its template.
- If you change the Tax Processing settings to remove the tax summary line from your store's cart, checkout, and order summary pages, you must manually update the templates to remove the tax summary line from emails that contain order summaries. (See Customize tax display in templates for more information.)

Each email template package you download includes the following files:

File	Description
`locales/<langcode>/Strings.xlf`	Contains localized strings for the locale specified by `<langcode>`. For example, the Spanish strings are located in the file `locales/es/Strings.xlf`. You can edit text inside the `<source>` tags.
`html_body.ftl`	FreeMarker template file that configures the HTML body of the email.
`text_body.ftl`	FreeMarker template file that configures the plain text body of the email. This file is included only in the package for the Forgotten Password and New Account emails.
`Readme.txt`	A help file that describes the fields that you can reference in the templates.

File	Description
`subject.ftl`	FreeMarker template file that configures the subject line of the email.

To download an email template package:

1. Click **Settings**.
2. Select **Email Settings**.
3. If you run multiple sites from a single Commerce instance, select the site whose email templates you want to download.
4. Click the type of email whose template you want to download.
5. Under Content, click **Download Current Template**.
6. Specify whether to open the ZIP file or save it.

Once you have made changes to the template files, compress them into a ZIP file and upload it. Make sure the ZIP file has the same name as the one you downloaded and contains the following files:

- `locales/<langcode>/Strings.xlf`
- `html_body.ftl` for HTML-based email messages or `text_body.ftl` for plain text email messages.
- `subject.ftl`

To upload an email template:

1. Click **Settings**.
2. Select **Email Settings**.
3. If you run multiple sites from a single Commerce instance, select the site whose email templates you want to download.
4. Click the type of email whose template you want to upload.
5. Under **Content**, click **Upload New Templates**.
6. Locate the ZIP file to upload and then click **Open**.
   When the upload is complete, Commerce displays a success message. This update takes effect immediately and does not require publishing.

Customize tax display in templates

By default, your store's cart, checkout, and order summary pages display a separate tax summary line.

You can configure the tax processing settings to hide the tax summary line. For example, if your catalog's prices include tax, you wouldn't want to show the tax separately. (See Configure Tax Processing for more information about these settings.) If you configure your store's tax processing settings to hide the tax summary line, you must customize the templates to remove the tax summary line from emails that contain order summaries. (Order Placed, Items Shipped, Agent Cancel Order, Agent Edit Order, and Agent Return Order emails all contain order summaries.)

To remove the tax summary line from an email template:

1. Download the email template as described in Download and edit email templates.
2. Make the following changes to the `html_body.ftl` file:
   - Remove the tax header by deleting the following from the section marked with the comment <!-- Start of Shipment Contents: items, cost breakdown -->:

     ```
     ("ITEMS_SHIPPED_TAX_TITLE")}:<br>
     ${getString
     ```

   - Remove the tax field by deleting the following:

     ```
     <br>${shippingGroup.tax}
     ```

3. Upload the updated template as described in Download and edit email templates.

Customize line-item display in templates

This section describes how to customize email templates to display custom properties of line items.

As discussed in the Customize Order Line Items section, you can add custom properties to order line items, such as a property for specifying the initials to use to monogram an item. By default, email templates that show order information do not display custom properties. This section describes how to customize these templates to display any custom properties added to line items. In addition, it describes how to display product IDs and SKU IDs in line items. (Note that these customizations are independent of each other; you can, for example, add the IDs without adding custom properties.)

You can modify how line items are displayed in the following email templates:

- Abandon Order
- Items Shipped
- Order Approved
- Order Pending for Approval
- Order Placed
- Order Quoted
- Order Rejected
- Payment Failure
- Quote Failed
- Quote Requested
- Scheduled Order Placed Failed
- Store Cancel Order

To add product IDs and SKU IDs to an email template, insert the following in the order items list in the `html_body.ftl` file:

```
<br/>
Product ID : ${product.productId}
<br/>
```

```
SKU ID: ${product.catRefId}
<br/>
```

To add custom line-item properties to an email template, insert the following in the order items list in the `html_body.ftl` file:

```
<#if product.dynamicProperties??>
  <#list product.dynamicProperties as dynProperty>
   <#if dynProperty.propertyValue??>
    <br/>
    ${dynProperty.propertyLabel}: ${dynProperty.propertyValue}
   </#if>
  </#list>
</#if>
```

See Download and edit email templates for information about how to modify email templates.

The following example shows the order items list section of the Order Placed template, customized to display custom properties, product IDs, and SKU IDs. The customizations appear in bold:

```
<!--  Start of order items list-->
<#list data.orderItems as product>
<tr>
  <td
    style="font-family: Helvetica, arial, sans-serif; font-size: 14px;
    color: #687078; text-align: left; line-height: 24px; padding: 5px
10px
    5px 10px;"
    st-content="3col-content1" width="30%">
    <img src="${product.imageLocation}"
    alt="${product.title!}" width="50%">
  </td>
  <td
  style="font-family: Helvetica, arial, sans-serif; font-size: 14px;
  color: #687078; text-align: left; line-height: 24px; padding: 5px
10px
  5px 10px;"
  st-content="3col-content1" width="40%">
  <a href="${product.location}">${product.title!}</a>
  <br/>
  Product ID : ${product.productId}
  <br/>
  SKU ID: ${product.catRefId}
  <br/>
  <!-- Variants -->
  <#if product.variants??>
    <#list product.variants as variant>
      <#if variant.optionValue??>
        <br/>
        ${variant.optionName}: ${variant.optionValue}
      </#if>
    </#list>
```

```
  </#if>
   <#if product.dynamicProperties??>
   <#list product.dynamicProperties as dynProperty>
    <#if dynProperty.propertyValue??>
     <br/>
     ${dynProperty.propertyLabel}: ${dynProperty.propertyValue}
      </#if>
    </#list>
   </#if>
  </td>
  <td
    style="font-family: Helvetica, arial, sans-serif; font-size: 14px;
    color: #687078; text-align: center; line-height: 24px; padding: 5px
    10px 5px 10px;"
    st-content="3col-content1" width="10%">
    ${product.quantity}</td>
  <td
    style="font-family: Helvetica, arial, sans-serif; font-size: 14px;
    color: #687078; text-align: right; line-height: 24px; padding: 5px
    10px 5px 10px;"
    st-content="3col-content1" width="20%">
    ${product.price}</td>
</tr>
</#list>
<!--  End of order items list -->
```

Add company name and logo to account-based email templates

In account-based storefronts, you can add a company name and logo to the emails generated by Commerce by editing the `html_body.ftl` file associated with any given email template.

Note: The account-based commerce feature may not be enabled in your environment.

To add a company name and logo, insert the following variables in the `html_body.ftl` file for the appropriate template:

- `${data.organization.name}` represents the company name and corresponds to the value entered for the account name when creating the account.
- `${data.organization.logoURL}` represents the path to the company logo and corresponds to the path you provided for the Store Logo during account creation.

For example, the following is a modified version of the Forgotten Password template's `html_body.ftl`:

```
${getString("PASSWORD_RESET_SALUTATION", data.firstName)} <br><br>

${getStringNotEscaped("PASSWORD_RESET_LINE_1", data.password)} <br><br>

${data.organization.name} <br><br>
<img src="${data.organization.logoURL}"> <br><br>

${getString("PASSWORD_RESET_LINE_2")} <br><br>
```

```
${getString("PASSWORD_RESET_SENT_SIGNATURE_TEXT")} <br><br>
```

At run time, both variables are replaced in the generated email with the name and logo image associated with the account.

To modify an email template, you must download its constituent files, make the modifications to the `html_body.ftl` file, and then upload it again. For details on this process, see the Download and edit email templates section.

You can add your company name and logo to the following email templates:

- Abandon Order
- Account Assignment Changed
- Agent Cancel Order
- Agent Edit Order
- Agent Forgot Password
- Agent Return Order
- Agent Return Order Refund
- Agent Shopper Registration
- Forgotten Password
- New Account
- Order Placed
- Wish List New Comment
- Wish List New Member
- Wish List New Post

These email templates are not yet compatible with company name and logo additions:

- Items Shipped
- Store Cancel Order

Notify a contact of multiple account or role changes in a single email

If your store supports account-based commerce, you can configure Commerce to send emails notifying contacts of changes to their account and role assignments. When you make several account or role changes for a contact at the same time, Commerce can notify the contact of those changes in a single email.

To take advantage of this feature, you must use the most recent version of the Account Assignment Changed and Role Assignment Changed email templates as described in Download and edit email templates. When you are using the most recent version of these templates, Commerce automatically includes multiple account or role assignment changes in a single email. If you use older versions of the templates, Commerce will send a separate email each time you save an account or role assignment change.

The consolidated email notification displays account or role assignment changes made in a single endpoint call. In the administration interface, this usually means changes made before you clicked the **Save** button. Some actions (such as changing a parent organization, assigning a sub account as a parent, or removing both parent and sub account associations at the same time) result in multiple endpoint calls, however. In this case, the contact will still receive multiple emails, one for each endpoint call.

Customize recommendations in templates

The Abandon Cart and New Account email templates allow you to include product recommendations with the emails you send.

You can configure the product recommendations information sent to your shoppers as follows:

- For the Abandon Cart email, you can provide product recommendations, set the number of recommendations, and set the strategy and restrictions for the recommendations.
- For the New Account email, you can provide product recommendations and set the number of products recommended.

See Display product recommendations for more information about these settings.

To customize product recommendations in an email template:

1. Download the email template as described in Download and edit email templates.
2. Edit the code sample provided below and add the block to the `html_body.ftl` file. It is recommended you add the code after the Call to Action block.
3. Upload the updated template as described in Download and edit email templates.

```
<!-- Show recommendations if present -->
<#if data.recommendations??>
<!-- Start of separator -->
<table width="100%" bgcolor="#ffffff" cellpadding="0" cellspacing="0"
border="0"
       id="backgroundTable" st-sortable="separator">
  <tbody>
  <tr>
      <td>
          <table width="600" align="center" cellspacing="0" cellpadding="0"
                 border="0" class="devicewidth">
              <tbody>
              <tr>
                  <td align="center" height="30" style="font-size:1px;
                      line-height:1px;"> </td>
              </tr>
              <tr>
                  <td width="550" align="center" height="1" bgcolor="#d1d1d1"
                      style="font-size:1px; line-height:1px;"> </td>
              </tr>
              <tr>
                  <td align="center" height="30" style="font-size:1px;
                      line-height:1px;"> </td>
              </tr>
              </tbody>
```

```
          </table>
      </td>
  </tr>
  </tbody>
</table>
<!-- End of separator -->
<!-- Start of Product Recommendations -->
<table width="100%" bgcolor="#ffffff" cellpadding="0" cellspacing="0"
border="0"
       id="backgroundTable">
  <tbody>
  <tr>
    <td>
      <table width="600" cellpadding="0" cellspacing="0" border="0"
align="center"
             class="devicewidth">
        <tbody>
        <tr>
          <td width="100%">
            <table width="600" cellpadding="0" cellspacing="0"
border="0"
                   align="center" class="devicewidth">
              <tr>
              <tr>
                <td><b>Some other products you may like ...</b></td>
              </tr>
              <tr>
                <td> </td>
              </tr>
              <tr>
                <td>
                  <!-- col 1 -->
                  <table width="100%" align="left" border="0"
cellpadding="0"
                         cellspacing="0" class="devicewidth">
                    <tbody>

                    <tr>
                      <td>
                        <!-- start of text content table -->
                        <table width="100%" align="center" border="0"
                               cellpadding="0" cellspacing="0"
                               class="devicewidthinner">
                          <tbody>

                          <!-- title2 -->
                          <tr>
                            <td width="30%" style="font-family:
Helvetica, arial,
                                sans-serif; font-size: 18px; color:
#FFFFFF;
                                text-align:left; line-height: 24px;
                                background: #1c73a3; padding:5px 10px
5px 10px;"
                                st-title="3col-title1">Product
```

```
Name</td>
                          <td width="50%" style="font-family: Helvetica,
arial,
                              sans-serif; font-size: 18px; color: #ffffff;
                              text-align:center; line-height: 24px;
                              background: #1c73a3; padding:5px 10px 5px
10px;
                              " st-title="3col-title1"> </td>
                          <td width="20%" style="font-family: Helvetica,
arial,
                              sans-serif; font-size: 18px; color: #ffffff;
                              text-align:right; line-height: 24px;
                              background: #1c73a3; padding:5px 10px 5px
10px;
                              " st-title="3col-title1">Price</td>
                        </tr>
                        <!-- end of title2 -->
                        <!-- Spacing -->
                        <tr>
                          <td width="30%" height="15" style="font-size:1px;
                              line-height:1px; mso-line-height-rule:
exactly;
                              "> </td>
                          <td width="50%" style="font-size:1px; line-
height:1px;
                              mso-line-height-rule: exactly;"> </td>
                          <td width="20%" style="font-size:1px; line-
height:1px;
                              mso-line-height-rule: exactly;"> </td>
                        </tr>
                        <!-- Spacing -->
                        <!-- content2 -->

                        <#list data.recommendations as product>
                        <tr>
                          <td width="30%" style="font-family: Helvetica,
arial,
                              sans-serif; font-size: 14px; color: #687078;
                              text-align:left; line-height: 24px;
                              padding:5px 10px 5px 10px;
                              " st-content="3col-content1">
                            <div class="imgpop">
                                <img src="${product.imageLocation}"
width="83"
                                    height="111" alt="${product.title!}">
                            </div>
                          </td>
                          <td width="40%" style="font-family: Helvetica,
arial,
                              sans-serif; font-size: 14px; color: #687078;
                              text-align:left; line-height: 24px;
                              padding:5px 10px 5px 10px;
                              " st-content="3col-content1">
                              <a href="${product.location}">
                              ${product.title!}</a>
```

```
                              <td width="20%" style="font-family:
Helvetica, arial,
                                  sans-serif; font-size: 14px; color:
#687078;
                                  text-align:right; line-height: 24px;
                                  padding:5px 10px 5px 10px;
                                  " st-content="3col-content1">
                                <#if product.priceRange?? &&
                                     product.priceRange="true">
                                  <span>${product.priceMin!}
                                        - ${product.priceMax!}</span>
                                <#elseif product.onSale?? &&
product.onSale="true">
                                  <span style="text-decoration:
                                        line-through">$
{product.listPrice!}</span>
                                  <span style="color: red">
                                        ${product.salePrice!}</span>
                                <#else>
                                  ${product.listPrice!}
                                </#if>
                              </td>
                            </tr>
                            </#list>

                            <!-- end of content2 -->
                            </tbody>
                          </table>
                          <!-- end of text content table -->
                        </td>
                      </tr>

                      </tbody>
                    </table>
                  </td>
                </tr>
              </table>
            </td>
            <!-- spacing -->

            <!-- end of spacing -->
            </tr>
          </tbody>
        </table>
      </td>
    </tr>
    </tbody>
</table>
</#if>
<!-- end of Product Recommendations-->
```

Add a site to a template

If your Commerce instance supports more than one site, one site is designated as the default. When you create a new site, the values for email settings and templates are copied to the new site from the default site.

The values that are copied include any customizations you have made to email templates for the default site. When you download and upload email templates in an environment that supports multiple sites, make sure you select the correct site first. See Download and edit email templates for more information.

In emails generated by Commerce, the following variables in the `html_body.ftl` file associated with any given email template specify site name and URL information.

- `${data.sitename}` represents the store name and corresponds to the value entered for the site name when creating the site.
- `${data.storefrontUrl}` represents the site base URL provided when creating the site.

At run time, these variables are replaced in the generated email with the name and URL associated with the site.

29

Upload Third-Party Files

Oracle Commerce allows you to upload third-party files to your sites, such as files used for site and merchant domain verification.

These files are used by services such as Google Search and Apple Pay, and they must be accessible via the file path specified by the service. The topics in this section describes how to upload and manage any third-party files your storefront requires.

Create folders for third-party files

Commerce includes a folder named `/thirdparty` where you can upload files needed by third-party services such as Apple Pay and Google Search.

Files stored in the `/thirdparty` folder can be accessed on your site. For example, if your site is www.example.com and you upload a file named `myfile.html`, the file can be viewed at the following URL:

```
http://www.example.com/myfile.html
```

You can create subfolders of `/thirdparty` and upload files to these subfolders using the Commerce Admin API. For example, you could create a subfolder named `/thirdparty/myfolder`, and upload the file `myfile.html` to this subfolder. The file would then be accessible at this URL:

```
http://www.example.com/myfolder/myfile.html
```

To create a subfolder of `/thirdparty`, issue a `POST` request to the `/ccadmin/v1/files/createFolder` endpoint on the administration server. The request JSON for this endpoint includes a single property, `folder`, for specifying the pathname, which must begin with `/thirdparty`.

For example, the following request creates a `/myfolder` subfolder of the `/thirdparty` folder:

```
POST /ccadmin/v1/files/createFolder  HTTP 1.1
Authorization: Bearer <access_token>

{
  "folder": "/thirdparty/myfolder"
}
```

If the folder creation is successful, the JSON payload in the response is empty. To verify that the folder was created successfully, see View a list of files and folders.

Upload third-party files to folders

You can upload files to the `/thirdparty` folder or to subfolders that you create.

Uploading a file involves the following steps:

1. Generate a token for the file upload.
2. Upload the file to the desired location.
3. View a list of files and folders.
4. Publish the files and folders to the production environment.
5. Verify that the file is present on the production environment.

These steps are described in the sections that follow.

Generate a token for the file upload

Before you can upload a file, you must generate a token to associate with the upload. To do this, use the `startFileUpload` endpoint (`PUT /ccadmin/v1/files`). The request JSON for this endpoint includes three properties:

- `filename` – The pathname for the file, relative to the `/thirdparty` folder. So, for example, if you want to upload a file named `policy.html` to be located directly in the `/thirdparty` folder, the value of the filename property would be `policy.html`. (Do not include a leading slash; `/policy.html` is incorrect.) If you want to upload a file named `policy.html` to be located in the `/thirdparty/myfolder` subfolder, the value of the filename property would be `myfolder/policy.html`. (If the subfolder does not already exist, it is created automatically when the file is uploaded.)
- `segments` -- The file must be uploaded in segments of less than 1 GB each. If your file is larger than 1 GB, it must be up broken up into multiple segments, with each segment uploaded separately. The segments property specifies the number of segments to be uploaded.
- `uploadtype` – Set the value of this property to `thirdPartyFile`.

The following request body generates a token for uploading a third-party file named `/thirdparty/myfolder/sampleFile.txt` in two segments:

```
{
  "filename": "myfolder/sampleFile.txt",
  "segments": 2,
  "uploadtype": "thirdPartyFile"
}
```

The response body includes the token and an array of the indices of the segments to be uploaded. The indices are zero-based, so the first segment is 0:

```
{
    "links": [
        {
            "rel": "self",
            "href": "http://myserver.example.com:7002/ccadmin/v1/files"
```

```
        }
    ],
    "segmentsNeeded": [
        0,
        1
    ],
    "token": "18a7878ddf1ab_1751"
}
```

Upload the file to the desired location

The actual upload is performed using the `uploadSegment` endpoint (`POST/ccadmin/v1/files/<token>`, where `<token>` is the token generated in the `startFileUpload` call). You must call `uploadSegment` once for each file segment, and each time specify these three properties:

- `filename` -- The pathname for the file, relative to the `/thirdparty` folder. This must match the filename used to generate the token.
- `index` -- The index for the file segment being loaded.
- `file` -- The based64-encoded content of the segment.

The following sample request uses the token generated by the `startFileUpload` call in the previous section to upload the first segment of a file:

```
POST /ccadmin/v1/files/18d396ba091fa_1755 HTTP 1.1
Authorization: Bearer <access_token>

{
  "filename": "myfolder/sampleFile.txt",
  "index": 0,
  "file":
"TG9yZW0gaXBzdW0gZG9sb3Igc2l0IGFtZXQsIGNvbnNlY3RldHVyIGFkaXBpc2Npbmcg
WxpdC4gTWF1cmlzIGF1Y3RvciBmZXVnaWF0IGp1c3RvLCBpZCB1bHRyaWNlcyBsb3JlbSBzb2RhbG
VzI
HNpdCBhbWV0LiBJbiBhdWN0b3Igc2VkIHNlbSBldCBpYWN1bGlzLiBNYWVjZW5hcyBiaWJlbmR1bS
Bua
XNsIG5lcXVlLiBBZW5lYW4gYXQgb3JuYXJlIGRpYW0uIEZ1c2NlIGlkIGRpZ25pc3NpbSBsb3JlbS
wgc
nV0cnVtIGNvbmRpbWVudHVtIGxlY3R1cy4gUGVsbGVudGVzcXVlIHZhcml1cyBjb25zZWN0ZXR1ci
BtZ
XR1cyBub24gc2FnaXR0aXMuIEludGVnZXIgZXN0IHF1YW0sIGRpZ25pc3NpbSBub24gbGVjdHVzIG
5lY
ywgY29udmFsbGlzIGNvbnZhbGxpcyBuaXNpLiBFdGlhbSBpbiBlZmZpY2l0dXIgbnVsbGEuIFN1c3
Blb
mRpc3NlIGVnZXQgdGluY2lkdW50IHR1cnBpcywgbmVjIHRlbXB1cyB1cm5hLiA="
}
```

For a multi-segment upload, the response body for each segment but the last is similar to this:

```
{
    "success": true,
    "links": [
        {
```

```
            "rel": "self",
            "href":
"http://myserver.example.com:7002/ccadmin/v1/files/18d396ba091fa_1755"
        }
    ]
}
```

For a single-segment upload, or the last segment of a multi-segment upload, the response body is similar to this:

```
{
    "result": {
        "@class": "atg.cloud.file.ThirdPartyUploadResultSummary",
        "failedFiles": 0,
        "allFilesFailed": false,
        "newFiles": 1,
        "modifiedFiles": 0,
        "unzipped": false,
        "failedFilesReasons": {}
    },
    "success": true,
    "links": [
        {
            "rel": "self",
            "href":
"http://myserver.example.com:7002/ccadmin/v1/files/18d396ba091fa_1755"
        }
    ]
}
```

View a list of files and folders

To list third-party files and their containing folders, issue a `GET` request to the `/ccadmin/v1/files` endpoint on the administration server. This endpoint has a required query parameter, folder, that you use to specify the folder to return results for. For example:

```
GET /ccadmin/v1/files?folder=/thirdparty HTTP 1.1
Authorization: Bearer <access_token>
```

You can use the `assetType` query parameter to specify whether to return a list of the files only (`assetType=file`), the folders only (`assetType=folder`), or both (`assetType=all`). The default is file, so if you omit this query parameter, a list of the files is returned.

The following example shows a sample call to the `getFiles` endpoint that returns both files and folders:

```
GET /ccadmin/v1/files?folder=/thirdparty&assetType=all HTTP 1.1
Authorization: Bearer <access_token>
```

The following example shows sample output from this call:


```
{
    "total": 2,
    "totalResults": 2,
    "offset": 0,
    "limit": 250,
    "links": [
        {
            "rel": "self",
            "href":
"http://myserver.example.com:7002/ccadmin/v1/files?
folder=%2Fthirdparty&assetType=all"
        }
    ],
    "sort": [
        {
            "property": "name",
            "order": "asc"
        }
    ],
    "items": [
        {
            "path": "/thirdparty/myfolder",
            "repositoryId": "folder10004",
            "name": "myfolder",
            "url": "http://myserver.example.com:7002/file/thirdparty/
myfolder"
        },
        {
            "path": "/thirdparty/myfolder/sampleFile.txt",
            "extension": "txt",
            "metadata": {},
            "size": 916,
            "repositoryId": "f10001",
            "name": "sampleFile.txt",
            "checksum": 6238228597895851000,
            "lastModified": "2017-01-04T21:19:14.015Z",
            "type": "file",
            "url":
"http://myserver.example.com:7002/file/v1857419716804211141/thirdparty/
myfolder/sampleFile.txt"
        }
    ]
}
```


The `getFiles` endpoint can also take a filter query parameter to limit the set of folders and files returned. The value of this parameter is a simple sequence of characters (no wild-card or regular expression elements) that is used to do substring matching on file or folder names. For example, the following call returns a list of the files in the `/thirdparty` folder whose names include the string “red”:

```
GET /ccadmin/v1/files?folder=/thirdparty&filter=red  HTTP 1.1
Authorization: Bearer <access_token>
```

Publish the files and folders to the production environment

To make your changes available to the production environment, publish them, as described in Publish Changes.

Verify that the file is present on the production environment

View the file in your browser to verify that it is present on your production environment. For example, to view a file uploaded to `/thirdparty/myfolder/sampleFile.txt`:

```
http://myserver.example.com:7002/myfolder/sampleFile.txt
```

Upload a Google site ownership verification file

Google provides a number of methods for verifying ownership of your sites. Commerce supports the HTML file upload method.

Complete the following steps to upload a Google site ownership verification file:

1. Create a Google site ownership verification HTML file as described on this page:

   ```
   https://support.google.com/webmasters/answer/35179?
   hl=en#verification_details
   ```

   The name of the file differs from site to site. The naming convention is `google<code>.html`, where `<code>` is a sequence of letters and numbers. For example, `google2827eae44ccfac6b.html`.
2. Depending on how your site URL is specified when you create the file, you may need to use the `createFolder` endpoint to create a subfolder of `/thirdparty`. For example, if your site is specified as `www.example.com`, the file should be uploaded to `/thirdparty`, so you do not need a subfolder; if your site is specified as `www.example.com/es`, the file should be uploaded to `/thirdparty/es`, so you need to create a subfolder named `/es`.
3. Use the `startFileUpload` endpoint to generate a token for uploading the file to the appropriate folder.
4. Use the `doFileSegmentUpload` endpoint to upload the file.
5. Publish your changes to the production environment.
6. View the file in your browser to verify that it was uploaded correctly. For example, if your site is `www.example.com/es`, and the file you uploaded is named `google2827eae44ccfac6b.html`, the URL for the file is:

   ```
   http://www.example.com/es/google2827eae44ccfac6b
   ```

Upload an Apple Pay merchant identity certificate

To support Apple Pay on your site, you must upload a merchant identity certificate.

Complete the following steps:

1. Create the Apple Pay Merchant Identity Certificate, as described on this page:

   `https://developer.apple.com/reference/applepayjs/`

   The name of the file is as follows:

   `apple-developer-merchantid-domain-association`
2. Use the `createFolder` endpoint to create a subfolder of `/thirdparty` named `/.well-known`.
3. Use the `doFileSegmentUpload` endpoint to upload the file.
4. Publish your changes to the production environment.
5. View the file in your browser to verify that it was uploaded correctly. For example, if your site is www.example.com, the file has this URL:

   `https://www.example.com/.well-known/apple-developer-merchantid-domain-association`

Delete third-party files

To delete a single third-party file, issue a `POST` request to the `/ccadmin/v1/files/deleteFile` endpoint on the administration server.

The request JSON includes a required `filename` property for specifying the file to delete. For example:

```
POST /ccadmin/v1/files/deleteFile  HTTP 1.1
Authorization: Bearer <access_token>

{
 "filename": "/thirdparty/sampleFile.txt"
}
```

To delete multiple third-party files, issue a `POST` request to the `/ccadmin/v1/files/deleteFiles` endpoint on the administration server. The request body includes a `deletePaths` property that specifies an array of files to delete:

```
POST /ccadmin/v1/files/deleteFiles  HTTP 1.1
Authorization: Bearer <access_token>

{
  "deletePaths": [
    "/thirdparty/google2827eae44ccfac6b.html",
    "/thirdparty/.well-known/apple-developer-merchantid-domain-association"
  ]
}
```

If the deletion is successful, the JSON payload in the response is empty.

The request body for these endpoints can include an optional `recursive` property. If this property is set to true, the call recursively deletes child folders under the specified folders or files. If the property is omitted, it defaults to false.

To make your changes available to the production environment, publish them, as described in Publish Changes.

Manage files on multiple sites

If your Commerce instance is running multiple sites, the `/thirdparty` directory supports creation of site-specific subfolders and files.

This behavior makes it possible for you to have, for example, a different Apple Pay merchant identity certificate for each site, without running into naming conflicts.

For example, suppose you have two sites, `siteA` and `siteB`, and each site requires a unique version of a file named `example.txt`. To upload the file for `siteA`, first use the `startFileUpload` endpoint, and set the value of the `x-ccsite` header to `siteA`:

```
PUT /ccadmin/v1/files  HTTP 1.1
Authorization: Bearer <access_token>
x-ccsite: siteA

{
  "filename": "example.txt",
  "segments": 1,
  "uploadtype": "thirdPartyFile"
}
```

Next, using the token returned in the response, call the `uploadSegment` endpoint (with `x-ccsite` set to `siteA`) to upload the file, as described in Upload the file to the desired location.

Once you have uploaded the `siteA` version of the file, use `startFileUpload` and `uploadSegment` with `x-ccsite` set to `siteB` to create the version of the file for `siteB`. If you then view `example.txt` in your browser, the version of the file you see will depend on which site's URL you use.

You can also create a global file by omitting the `x-ccsite` header when you call `startFileUpload` and `uploadSegment`. When you use the `getFiles` endpoint to view a list of files, the files you see depend on the current site in the request. For example, if you call `getFiles` without the `x-ccsite` header, you see only the global versions of files. If you set `x-ccsite` to `siteA`, you see all `siteA`-specific files and folders, as well as any global files and folders that are not overridden by `siteA`-specific versions. (If there is both a global and a `siteA`-specific version of a file, you see the `siteA`-specific version.) Similar logic applies when using the `deleteFile` endpoint to delete a file.

Note that since the storefront always has a current site, if there is a site-specific version of a file for the current site as well as a global version of the file, the storefront always sees the site-specific version.

30

Manage Guest Checkout

Your site can require that anonymous shoppers log in before they check out.

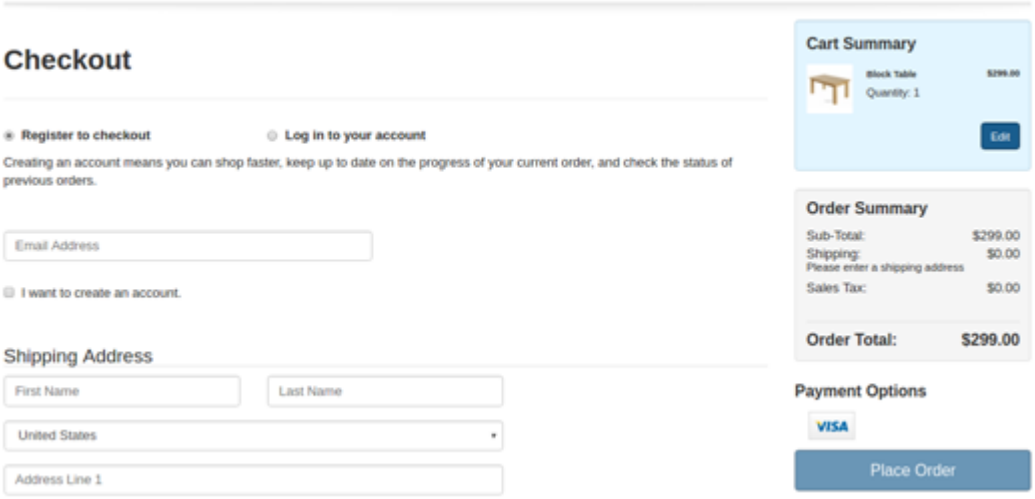
This feature can be used for anonymous shoppers who will log into either a personal account or a business account. The topics in this section describe the coding requirements to implement a restricted guest checkout UI.

Example for restricting guest checkout

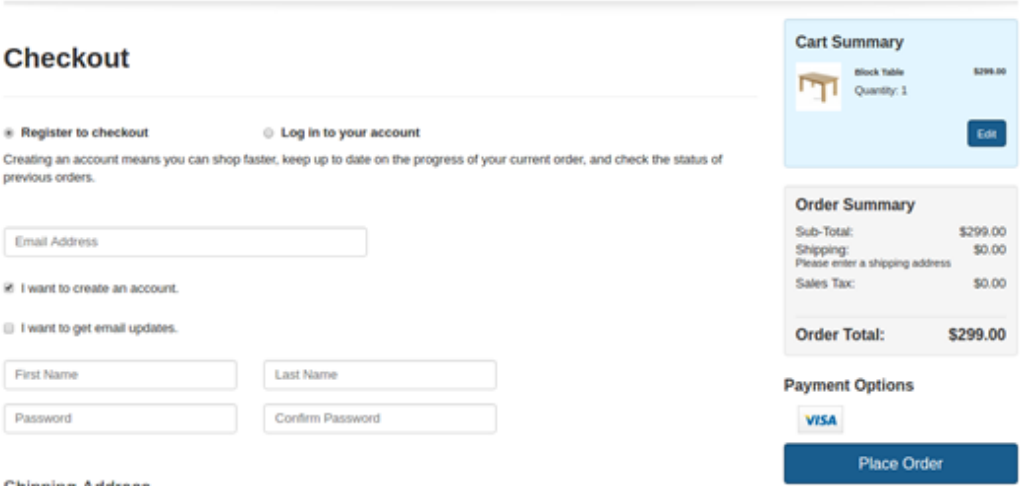
Preventing guest checkout is a two-step process involving disabling the guest checkout administration option and modifying the checkout UI on the storefront.

First, you must disable Guest Checkout on the Settings page in the Commerce administration interface. This sets the flag used for testing whether guest checkout is allowed. For details on how to set this flag, see Restrict guest checkout. Second, you must modify your storefront's checkout UI to prevent anonymous shoppers from accessing check out features until they have logged in. For example, you might disable the Place Order button on the Checkout page until the shopper logs in. The specifics of modifying your checkout UI to restrict guest checkout vary with each storefront's requirements. As such, a simple example has been provided to give you an understanding of how you can restrict access to the Place Order button until a shopper has logged in or agreed to create an account.

In the illustration below, the out-of-the-box Login-Checkout widget has been modified to replace the Checkout as Guest button with a Register to Checkout button. The Register to Checkout button has been selected but the I Want to Create an Account checkbox has not. This combination of settings indicates that the shopper is an anonymous shopper who does not yet have an account. For a storefront that restricts guest checkout, this combination of settings should result in a disabled Place Order button.



When the shopper enables the "I want to create an account" option, he is agreeing to become a registered shopper using the information he provides on the Checkout page. At this point, the Place Order button becomes enabled.



To create this UI, you first modify the Login – Checkout widget to introduce `ko if` and `ko ifnot` bindings that check for whether the `guestCheckoutEnabled` flag is true or false and then render the appropriate radio button text (either “Checkout as guest” or “Register to checkout”) depending on the state of the flag.

```
<!-- ko with: user -->
<div id="checkout-registration">
  <h2 data-bind="widgetLocaleText:'checkoutRegistrationText'"></h2>
  <hr>
  <fieldset id="checkoutOptions" data-bind="visible: !loggedIn()">
  <legend id="checkoutOptions-legend"
    data-bind="widgetLocaleText:'checkoutOption'"></legend>
  <div class="row">
    <div class="form-group">
      <div class="col-sm-6 col-lg-4 cc-checkoutRegistration-radio">
        <label class="radio"
           data-bind="attr:{ for: 'CC-checkoutRegistration-
userOption-'+
           $parent.order().checkoutGuest() }">
          <input type="radio" class="form control" name="account"
              data-bind="value: $parent.order().checkoutGuest,
              attr:{ id: 'CC-checkoutRegistration-userOption-
              '+$parent.order().checkoutGuest() },
              checked: $parent.order().checkoutOption"/>
          <!-- ko if: $data.contextData.global.guestCheckoutEnabled -->
            <span data-bind="widgetLocaleText:
'checkoutAsGuestText'"></span>
          <!-- /ko -->
          <!-- ko ifnot: $data.contextData.global.guestCheckoutEnabled
-->
            Register to checkout
          <!-- /ko -->
        </label>
      </div>
      <div class="col-sm-6 col-lg-8 cc-checkoutRegistration-radio">
        <label class="radio"
          data-bind="attr:{ for: 'CC-checkoutRegistration-userOption-
          '+$parent.order().checkoutLogin() }">
          <input type="radio" class="form control" name="account"
```

```
            data-bind="value: $parent.order().checkoutLogin,
            attr:{ id: 'CC-checkoutRegistration-userOption-
            '+$parent.order().checkoutLogin() },
            checked: $parent.order().checkoutOption"/>
           <span data-bind="widgetLocaleText: 'loginToAccountText'"></span>
        </label>
      </div>
    </div>
  </div>
  ...
  ...
```

Note: In the interest of simplicity, this example does not add a resource string for the "Register to checkout" label. To do that, you must download the widget, update the source files, and import the updated widget into Commerce.

Next, you modify the Order Summary – Checkout widget to update the `handleCreateOrder` enable data-binding with logic that controls whether the Place Order button is enabled or disabled, based on whether guest checkout is enabled and whether the shopper is logged in:

```
...
<div id="CC-checkoutOrderSummary-placeOrder" class="checkout row">
    <button class="cc-button-primary col-xs-12"
        data-bind="click: handleCreateOrder,
        enable: (!user().contextData.global.guestCheckoutEnabled &&
        user().loggedIn() && order().enableOrderButton) ||
        (!user().contextData.global.guestCheckoutEnabled && !
user().loggedIn()
        && order().enableOrderButton && order().createAccount) ||
        (user().contextData.global.guestCheckoutEnabled &&
        order().enableOrderButton)">
          <span data-bind="widgetLocaleText:'placeOrderText'"></span>
    </button>
</div>
...
```

Note about preventing self-registration in account-based storefronts

In account-based storefronts, accounts are created manually by merchant administrators in Commerce. Therefore, storefronts that support account-based contacts must be careful not to allow this type of contact to self-register.

To accommodate this use case, the example in the Example for restricting guest checkout section would need to be modified to restrict account-based contacts to logging in only. For storefronts that support both individual, registered shoppers and account-based contacts, your UI will have to make it clear that the self-registration process creates a personal account only, not a business account.

31
Manage Saved Carts

The saved-carts feature makes it possible for a registered shopper to create and save multiple shopping carts.

The topics in this section describe how to implement saved carts.

Understand saved carts

By default, Commerce supports creation and management of one shopping cart per shopper, per storefront.

The saved cart feature lets a shopper save multiple unfinished carts and retrieve them later. A shopper can create an order with only one cart at a time, though they can merge saved carts into the current cart.

This feature is available for both individual registered shoppers and account-based shoppers. If an account-based shopper is a member of more than one account, they cannot access their saved carts across accounts. That is, they can access saved carts only for the account for which they are currently logged into the store.

The order state for a saved card is INCOMPLETE, which specifies that the order is still in the purchasing stages. See Understand order states for more information.

To support saved carts, the `getAllOrdersForProfile` endpoint of the Store API includes a Boolean query parameter, `incompleteOnly`. The following sample request returns only incomplete orders for the logged-in shopper profile:

```
GET /ccstoreui/v1/orders/getAllOrdersForProfile?incompleteOnly=true
```

Once you enable the saved carts feature, a `cartName` order property is available to specify the cart associated with the order. This property will appear in the body of all webhooks that send order information, as well as in any REST API response that includes an order. The value of `cartName` is the string the shopper entered when they saved the cart.

A new view model, `multiCartViewModel`, is used to list all incomplete orders and provide support for pagination and search by `cartName` when a shopper wants to view their list of saved carts.

Note: When you enable saved carts, it is possible, though unlikely, that a shopper might see old incomplete carts the first time they view their saved cart list in their profile. This can occur if something unexpected happened during a past order flow; for example, if a shopper added items to their cart before logging in and the cart did not merge properly after login, an incomplete cart might have remained in the system. That cart would appear in the shopper's saved cart list. Shoppers can simply delete any older carts that appear the first time they view their saved carts list.

Enabling the saved carts feature does not affect scheduled orders, as Commerce maintains schedule information separately.

If you configured Commerce to use the External Price Group and Catalog webhook to assign a shopper a specific catalog and price group at login, and the shopper then opens a saved cart, Commerce reprices the cart based on the catalog and price group specified by the webhook response. See Assign Catalogs and Price Groups to Shoppers for more information about this webhook.

Create a widget to support saved carts

To enable and manage saved carts for a registered individual or account-based shopper, you write a custom widget to include on your storefront's Cart pages.

For detailed information about creating widgets, see Create a Widget.

This topic includes the following sections:

- Create the widget structure for the saved-carts sample widget
- Create the JavaScript file for the saved-carts sample widget
- Create template files for the saved-carts sample widget

Create the widget structure for the saved-carts sample widget

Widgets that include user interface elements must include display templates. The following shows an example of the files and directories in a saved-carts widget. Notice that it includes two display templates; these are described in detail in Create template files for the saved-carts sample widget.

```
MultiCartDemoWidget/
    ext.json
    widget/
        multiCart_v1/
            widget.json
            js/
                multi-cart.js
            less/
                widget.less
            locales/
            en/
                ns.multicart.json
            templates/
                display.template
                pagination.template
```

Because this widget includes user interface elements that allow the shopper to work with saved carts, you must not create it as a global widget. Set the global property in the `widget.json` file to false:

```
"global": false
```

The JavaScript code you write extends the `multiCartViewModel` class. For more information about the widget structure and the contents of the `ext.json` and `widget.json` files, see Create the widget structure .

Create the JavaScript file for the saved-carts sample widget

The widget's JavaScript file includes functions that let shoppers save, retrieve, merge, and delete carts:

- `listIncompleteOrders` gets all the incomplete orders (saved carts) associated with the logged-in shopper profile.
- `createOrderWithTemporaryItems` creates an incomplete order for a shopper who has not logged in.
- `createNewIncompleteCart` crates a new saved cart for the logged-in shopper
- `loadParticularIncompleteOrder` displays a saved cart.
- `mergeWithParticularIncompleteOrder` merges a saved cart with the current cart.
- `deleteParticularIncompleteOrders` deletes a saved cart.

The following example shows sample JavaScript that implements the saved-cart functionality:

```
define(

  //-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
  ['knockout', 'pubsub', 'notifier', 'CCi18n', 'ccConstants',
 'navigation', 'ccRestClient','viewModels/multiCartViewModel'],

  //-------------------------------------------------------------------
  // MODULE DEFINITION
  //-------------------------------------------------------------------
  function(ko, pubsub, notifier, CCi18n, CCConstants, navigation,
 ccRestClient, MultiCartViewModel) {

    "use strict";

    return {

      WIDGET_ID:        "multiCart",
      display: ko.observable(false),
      currentCartName: ko.observable(""),
      fetchSize: ko.observable(10),
      cartNameSearch: ko.observable(""),

      onLoad: function(widget) {
        var self = this;
        widget.listingViewModel = ko.observable();
        widget.listingViewModel(new MultiCartViewModel());
        widget.listingViewModel().itemsPerPage = widget.fetchSize();
        widget.listingViewModel().blockSize = widget.fetchSize();

        $.Topic(pubsub.topicNames.USER_AUTO_LOGIN_SUCCESSFUL)
.subscribe(function(){
            widget.listIncompleteOrders();
        });
        $.Topic(pubsub.topicNames.USER_LOGIN_SUCCESSFUL).subscribe(function()
```

```
{
        widget.listIncompleteOrders();
        if(widget.cart().items().length>0){
            widget.cart().isCurrentCallInProgress = true;
            widget.createOrderWithTemporaryItems();
        }
        });

        $.Topic(pubsub.topicNames.CART_PRICE_SUCCESS).subscribe(functio
n(){
        if(widget.user().loggedIn()){
            widget.listIncompleteOrders();
            widget.currentCartName("");
        }
        });
        $.Topic(pubsub.topicNames.CART_DELETE_SUCCESS).subscribe(functi
on(){
            if(widget.user().loggedIn()){
                widget.listIncompleteOrders();
            }
            });


        widget.listOfIncompleteOrders = ko.computed(function() {
            var numElements, start, end, width;
            var rows = [];
            var orders;
              var startPosition, endPosition;
              // Get the orders in the current page
              startPosition = (widget.listingViewModel().currentPage()
 - 1) * widget.listingViewModel().itemsPerPage;
              endPosition = startPosition +
 parseInt(widget.listingViewModel().itemsPerPage,10);
              orders =
widget.listingViewModel().data.slice(startPosition,
 endPosition);

            if (!orders) {
              return;
            }
            numElements = orders.length;
            width = parseInt(widget.listingViewModel().itemsPerRow(),
10);
            start = 0;
            end = start + width;
            while (end <= numElements) {
              rows.push(orders.slice(start, end));
              start = end;
              end += width;
            }
            if (end > numElements && start < numElements) {
              rows.push(orders.slice(start, numElements));
            }
            return rows;
          }, widget);
```

```
    },

    beforeAppear: function (page) {
        var widget = this;
        if (widget.user().loggedIn() == false) {
            widget.display(false);
        } else {
            widget.listIncompleteOrders();
            widget.display(true);
        }
      },

    /**
     * @function
     * @name multi-cart#listIncompleteOrders
     *
     * call to list incomplete orders for logged in profile.
     */
    listIncompleteOrders : function() {
       var self = this;
       var inputDate ={};
       //inputDate[CCConstants.SORTS] = "lastModifiedDate:desc";
       self.listingViewModel().sortProperty = "lastModifiedDate:desc";
       //set self.listingViewModel().cartNameSearch
       //string to search based on cartname
       if (self.user() && !self.user().loggedinAtCheckout()) {
       self.listingViewModel().refinedFetch();
       }
    },

    /**
     * @function
     * @name multi-cart#createOrderWithTemporaryItems
     *
     * method to create new incomplete cart with anonymous cart items
     */
    createOrderWithTemporaryItems : function() {
       var self = this;
       self.cart().createNewCart(true);
       self.cart().validateServerCart();
       self.cart().getProductData();
       self.cart().createCurrentProfileOrder();
    },

    /**
     * @function
     * @name multi-cart#createNewIncompleteCart
     */
    createNewIncompleteCart : function() {
       var self = this;
       self.cart().createNewCart(true);
ccRestClient.setStoredValue(CCConstants.LOCAL_STORAGE_CREATE_NEW_CART,true);
       self.cart().emptyCart();
       self.user().orderId('');
       self.user().persistedOrder(null);
```

```
        self.user().setLocalData('orderId');
        self.currentCartName("");
      },


      deleteParticularIncompleteOrders: function(pOrderId) {
        var self = this;
        self.cart().deleteParticularIncompleteOrders(pOrderId);
      },

      /**
       * @function
       * @name UserViewModel#loadParticularIncompleteOrder
       */
       loadParticularIncompleteOrder : function(pOrderId) {
       var self = this;
         self.cart().loadCartWithParticularIncompleteOrder(pOrderId);
       },
      /**
       * @function
       * @name UserViewModel#mergeWithParticularIncompleteOrder
       */
       mergeWithParticularIncompleteOrder : function(pOrderId) {
       var self = this;
         self.cart().mergeCartWithParticularIncompleteOrder(pOrderId);
       },

       saveIncompleteCart : function(pOrderId) {
       var self = this;
         self.cart().cartName(self.currentCartName());
         self.cart().priceItemsAndPersist();
       }

    };
  }
);
```

Create template files for the saved-carts sample widget

The widget's `display.template` file contains code that renders a page where shoppers can see a list of saved carts, display a saved cart, merge a saved cart with the current cart, or delete a saved cart.

The widget's `display.template` file contains the following code for rendering the page:

```
<!-- ko if: display-->
 <!-- ko with: cart -->
 <!-- ko if:($parent.user().loggedInUserName() &&
($parent.user().loggedIn()
 || $parent.user().isUserSessionExpired()))-->
     <div id="CC-multiCart">
          <div class="row col-md-12">
           <h3 class="modal-title text-center">Your Saved Carts</h3>
          </div>
```

```
            <div id="CC-multicartorder-table-md-lg-sm" class="row hidden-
xs">
            <section id="orders-info" class="col-md-12" >
             <table class="table" >
             <thead>
         <tr>
      <th class="col-md-2 " scope="col" data-bind="widgetLocaleText :
 'orderNumber'"></th>
           <th class="col-md-2 " scope="col" data-bind="widgetLocaleText:
 'cartName'"></th>
           <th class="col-md-2 " scope="col" data-bind="widgetLocaleText:
 'orderTotal'"></th>
           <th class="col-md-3" scope="col"><div class="sr-only"></div></th>
           <th class="col-md-3" scope="col"><div class="sr-only"></div></th>
           <th class="col-md-3 " scope="col" data-bind="widgetLocaleText:
 'delete'"></th>
               </tr>
              </thead>
              <!-- ko if: $parent.listOfIncompleteOrders().length > 0 -->
              <tbody data-bind="foreach:$parent.listOfIncompleteOrders">
              <tr>
           <td class="col-md-2" data-bind="text : $data[0].orderId"
 scope="row"></td>
           <td class="col-md-2" data-bind="text : $data[0].cartName"
 scope="row"></td>
           <td class="col-md-2" data-bind="currency: {price: $data[0].total,
 currencyObj: $data[0].priceListGroup.currency}"  scope="row"></td>
           <td class="col-md-3">
             <button class="cc-button-primary pull-right" href="#"
 data-dismiss="modal"
 data-bind="click:$parents[1].loadParticularIncompleteOrder.bind($parents[1],
$data[0].orderId)" >
              <span data-bind="widgetLocaleText: 'LoadThis'
 ,attr: {title: 'Clicking this will clear the cart and load this order'}">
</span>
             </button>
           </td>
           <td class="col-md-3">
             <button class="cc-button-primary pull-right" href="#"
 data-dismiss="modal" data-
bind="click:$parents[1].mergeWithParticularIncompleteOrder.bind($parents[1],
$data[0].orderId)" >
              <span data-bind="widgetLocaleText: 'MergeInto',attr:
 {title: 'Clicking this will merge the cart items into this order'}"></span>
            </button>
           </td>
           <td class="col-md-3">
           <button class="cc-button-primary pull-right" data-
bind="click:$parents[1].deleteParticularIncompleteOrders.bind($parents[1],
$data[0].orderId)" >
              <span data-bind="widgetLocaleText: 'delete' ,attr:
 {title: 'Clicking this will delete this cart'}"></span>
             </button>
           </td>
             </tr>
```

```
                </tbody>
                <!-- /ko -->
                <!-- ko if: $parent.listOfIncompleteOrders().length ==
0 -->
                <tbody>
                <tr>
                <td colspan="5">
                   <span data-bind="widgetLocaleText:'noOrders'">
                   </span></td>
                </tr>
                </tbody>
                <!-- /ko -->
                </table>
               </section>
               </div>
                 <!-- ko with: $parent.listingViewModel -->
  <div id="cc-paginated-controls-bottom"
 class="row col-md-12 visible-xs visible-sm visible-md visible-lg">
    <div data-bind="visible : (totalNumberOfPages() > 1)">
      <div>
        <div data-bind="template: { name:
$parents[1].templateAbsoluteUrl('/templates/
paginationControls.template')
 , templateUrl: ''}"
          class="row pull-right"></div>
      </div>
    </div>
  </div>
  <!-- /ko -->
      <div class="row col-md-12">
        <!-- ko if: $data.items().length == 0 -->
         <button type="button" class="btn btn-default"
 data-bind="click:$parent.createNewIncompleteCart.bind($parent)">
Create New</button>
         <!-- /ko -->
         <!-- ko if: $data.items().length > 0 -->
         <section id="cart-details-heading" >
         <h3 class="modal-title text-center"
 data-bind="widgetLocaleText:'currentCart'"></h3>
         </section>
         <section id="cart-info" class="col-md-12" >
         Cart Name: <span data-bind="text: cartName"></span>
         <table class="table" >
               <thead>
                 <tr>
            <th class="col-md-3 " scope="col" data-
bind="widgetLocaleText:
 'referenceId'"></th>
            <th class="col-md-3 " scope="col" data-
bind="widgetLocaleText:
 'quantity'"></th>
            <th class="col-md-3 " scope="col" data-
bind="widgetLocaleText:
 'total'"></th>
                </tr>
```

```
                </thead>
                <tbody data-bind="foreach:$data.items" >
                <tr>
                   <td class="col-md-3 text-left" data-bind="text :catRefId"
 scope="row"></td>
                   <td class="col-md-3 text-left" data-
bind="text :quantity()"
 scope="row"></td>
                   <td class="col-md-3 text-left" data-
bind="text :itemTotal()"
 scope="row"></td>
                </tr>
                </tbody>
                  </table>

                  <input type="text" class="col-md-4 form-control"
 name="currentCartName" id="currentCartName" data-bind="value:
 $parent.currentCartName, widgetLocaleText : {value:'cartNameText',
 attr:'placeholder'}">
                  <button type="button" class="btn btn-default"
 data-bind="click:$parent.saveIncompleteCart.bind($parent)">Save Cart</
button>
                  </section>
                  <section id="footer-buttons">
         <button type="button" class="btn btn-default" data-
bind="click:$parent.createNewIncompleteCart.bind($parent)">Create New</
button>
         </section>
         <!-- /ko -->
        </div>
   </div>
     <!-- /ko -->
   <!-- /ko -->
 <!-- /ko -->
```

The widget's `display.template` calls another template file, `paginationControls.template`. This template file contains .the following code for rendering multiple pages when the list of carts is long:

```
<div class="btn-group">

    <a href="#" class="btn btn-default" data-bind="click:
 getFirstPage, widgetLocaleText :
 {value:'goToFirstPageText', attr:'aria-label'},
 makeAccess: {readerText: 'Go to first page  ', cssContent: 'on'},
 css: { disabled: $data.currentPage() == 1 }, widgetLocaleText:
 'goToFirstPagePaginationSymbol'" >&lt;&lt;</a>
    <a href="#" class="btn btn-default" data-bind="click: decrementPage,
 widgetLocaleText : {value:'goToPreviousPageText', attr:'aria-label'},
 makeAccess: {readerText: 'Go to previous page  ', cssContent: 'on'},
 css: { disabled: $data.currentPage() == 1 }, widgetLocaleText:
 'goToPreviousPagePaginationSymbol'" rel="prev">&lt;</a>

    <!-- ko foreach: pages -->
```

```
      <a href="#" class="btn btn-default" data-bind="click:
 $parent.changePage.bind($parent, $data), css: {active:
 $data.pageNumber===$parent.clickedPage() }">
        <!-- ko if: $data.selected === true -->
          <span data-bind="widgetLocaleText : {value:'activePageText',
 attr:'aria-label'}, makeAccess: {readerText: 'Active page is  ',
 cssContent: 'on'}"></span>
        <!-- /ko -->
        <!-- ko if: $data.selected === false -->
          <span data-bind="widgetLocaleText : {value:'clickToViewText',
 attr:'aria-label'}, makeAccess: {readerText: 'Click to view page
 ',
 cssContent: 'on'}"></span>
        <!-- /ko -->
        <span data-bind="ccNumber: $data.pageNumber"></span>
      </a>
    <!-- /ko -->

    <a href="#" class="btn btn-default" data-bind="click:
incrementPage,
 widgetLocaleText : {value:'goToNextPageText', attr:'aria-label'},
makeAccess:
 {readerText: 'Go to next page   ', cssContent: 'on'}, css:
{ disabled:
 currentPage() == $data.totalNumberOfPages() }, widgetLocaleText:
 'goToNextPagePaginationSymbol'" rel="next">&gt;</a>
    <a href="#" class="btn btn-default" data-
bind="click: $data.getLastPage,
 widgetLocaleText : {value:'goToLastPageText', attr:'aria-label'},
makeAccess:
 {readerText: 'Go to last page  ', cssContent: 'on'}, css:
{ disabled:
 currentPage() == $data.totalNumberOfPages() }, widgetLocaleText:
 'goToLastPagePaginationSymbol'">&gt;&gt;</a>

</div>
```

Customize emails for saved carts

If your store supports saved carts, you can customize the Abandoned Order email template, which remind customers that they left unpurchased items in their shopping carts, to include the name of the saved cart.

Note: Though it is most useful in Abandoned Order emails, you can add the name of a saved cart to any of the following email templates that include order details: Items Shipped, Quote Failed, Order Approved, Order Rejected, Order Payment Initiated, Order Placed, Order Quoted, Quote Requested, Scheduled Order Placed Failed, Store Cancel Inflight Order, Store Cancel Order, and Store Return Order.

The `cartName` property is available to email templates to display the name of a saved cart. This property comes from the Orders resource in the Store REST API.

To display the name of a saved cart in an email template:

1. Download the email template as described in Customize email templates.

2. Update the `html_body.ftl` file.
   Add `cartName` to the main body of the email.
3. Upload the updated template as described in Customize email templates.

If a shopper has multiple saved carts that meet the Abandoned Cart settings criteria, Commerce sends a separate email for each cart.

See Configure Abandoned Cart settings for more information about Abandoned Order emails.

32

Manage the Use of Personal Data

The European Union General Data Protection Regulation (GDPR) enacts a set of legal requirements designed to control the collection and storage of personal data.

To address GDPR requirements, you may need to observe various practices regarding the handling of shopper information on your Oracle Commerce sites. This regulation is designed to protect the data privacy of all EU citizens and may require website customization.

Important: Consult legal counsel for professional guidance if you believe your websites and commerce operations may be subject to the GDPR. It is your responsibility to assess the legal and operational implications of the GDPR on your business and implement changes to any websites as necessary. For detailed information and guidelines on the European Union General Data Protection Regulation, refer to https://gdpr.eu.

This chapter discusses tools Commerce provides to help you address two key aspects of the GDPR, consent and right to erasure:

- Consent is the right of a shopper to allow, or disallow, the collection or processing of personal data. For information on setting up consent, refer to Configure consent requests.
- Right to erasure requires you to delete data about a shopper on your sites if the shopper requests it. For information on deleting shopper data, refer to Delete shopper information.

California Consumer Privacy Act

A privacy initiative similar to the GDPR, the California Consumer Privacy Act (CCPA), went into effect in the state of California on January 1, 2020. You can use the tools described in this chapter and in Control access to shopper data to help your sites meet the requirements of this act. Note that although the GDPR and the CCPA are similar in some ways, they are not identical. Consult legal counsel for professional guidance if you believe your websites and commerce operations may be subject to the CCPA. For information about the CCPA, see https://oag.ca.gov/privacy/ccpa. For additional information about Oracle Commerce and CCPA compliance, see Preparing Your Oracle Commerce Sites for the California Consumer Privacy Act.

Configure consent requests

You must determine which types of data processing activities need consent, how to get permissions, and how to add details to your site's terms and conditions.

You must also define the GDPR-based shopper profile properties and to determine the next steps in the shopper's experience when consent is given or revoked. Oracle Commerce does provide a set of tools that you can use and customize to assist with compliance of some of the GDPR requirements.

You can ask consent for various data processing types at different points in a shopper's visit. Examples of data processing types include, but are not limited to, processing orders, sending marketing material, enabling third-party data sharing, creating cookies, or personalizing the shopper's experience. You can capture consent when a shopper logs in, during the order checkout process, or when you work within the shopper's profile. Additionally, the Receive

Email Updates checkbox that is displayed on these pages allows you to request email consent.

When a shopper creates an order, your policy may be to consider that the shopper is inherently giving permission to use their personal data for the purpose of processing the order. It is up to you to provide any necessary disclaimer text or to customize order consent requests. To do this, you can customize widgets and profile properties.

If you prefer to request separate consent to capture data for order processing, and the shopper does not give consent, you must determine what actions occur. For example, you might prevent profile registration or guest checkout. Once you have determined the workflow, you should create the necessary customization.

Understand consent properties and cookies

The following table describes properties and cookies provided by Commerce for managing various types of consent:

Consent For	Property/Cookie	Description
Personalization	`requireGDPRP13nConsent`	This site-level property indicates if the shopper must provide consent when they register on your site. When consenting, the shopper is allowing you the right to perform personalization. For additional information, refer to the Manage personalization consent section.
Personalization	`GDPRP13nConsentGranted`	This profile property tracks a shopper's consent status. For additional information, refer to the Manage personalization consent section.
Personalization	`GDPRP13ConsentDate`	This profile property tracks the date that the shopper provided consent. For additional information, refer to the Manage personalization consent section.
Personalization and Site	`GDPRCookieP13nConsentNotRequired`	This cookie is placed on a shopper's browser if your site does not require consent, or if the shopper's locale is identified as a non-GDPR country. For additional information, refer to the Manage cookie-based consent section.

Consent For	Property/Cookie	Description
Personalization and Site	`GDPRCookieP13nConsentGranted`	This cookie is placed on a shopper's browser if the shopper has given their consent. For additional information, refer to the Manage cookie-based consent section.
Site	`requireGDPRCookieConsent`	This site-level property identifies if the shopper is required to accept the cookies used on your site. For additional information, refer to the Manage cookie-based consent section.

The following sections describe how to use these properties and cookies. Note that this documentation is not intended as legal advice for the GDPR. Please refer to your legal counsel for guidance.

Manage personalization consent

You may want to obtain a shopper's consent to perform profile-based personalization. For example, if you use audiences, or the product recommendations widget, you may need to collect the personal data stored in a shopper's profile.

You can collect profile data consent at shopper registration or order checkout by using the GDPR-based profile properties to indicate the need to display consent checkboxes. Configure personalization consent using the following widgets and elements:

- The `customerProfile` widget
- The `shopperDetail` widget
- The `checkoutRegistration` widget
- The `login-registration` element
- The `contact-login-for-managed-contacts` element

Note that these widgets will contain the necessary consent fields by default when you set your environment to require the GDPR consent.

To indicate that your site needs consent to use the shopper's personal data, use the Oracle Commerce Admin API `updateSite` endpoint to set `requireGDPRP13nConsent` to `true`. By default, this field is set to `false`. For example:

```
PUT /ccadmin/v1/sites/siteUS  HTTP/1.1

{
  "properties":
  {
    "requireGDPRCookieConsent":  true,
    "requireGDPRP13nConsent":  true
  }
}
```

Then, when a shopper accesses your site, they are presented with a checkbox asking if they would like to see relevant, or personalized, data. Additionally, they will be presented with the Receive Email Updates checkbox.

A shopper's consent status is stored in the `GDPRP13nConsentGranted` property of their profile. The date that the shopper provided consent is stored in the `GDPRP13nConsentDate` property.

You should work with your legal team to determine the actions required for various configurations. For example, if your site uses audiences and you have set the `requireGDPRP13NConsent` flag to true, shoppers must provide consent. If a shopper does not provide consent, the non-consenting shopper cannot be a member of any of your audiences that use shopper profile data. You may want to indicate to the shopper that this will occur if they do not consent.

Once a shopper has provided initial consent, you can determine what, if any, situations require the shopper to provide new or additional consent. By default, shoppers who have given consent will not be presented with additional consent requests unless you configure your storefront as such.

The following example shows how you could modify the Shopper Details and Customer Profile widget templates to require GDPR personalization consent:

```
<div class="row col-md-12" data-bind="visible:$parent.site().
    requireGDPRP13nConsent">
  <div class="form group">
    <div class="checkbox" id="CC-customerProfile-edit-
personalizationConsent-
            checkbox">
        <label for="CC-customerProfile-edit-personalizationConsent">
          <input type="checkbox" name="personalization-Consent"
                data-bind="checked: GDPRProfileP13nConsentGranted"
                id="CC-customerProfile-edit-personalizationConsent">
          <span data-bind="widgetLocaleText:
'personalizationConsentText'"
                id="CC-customerProfile-edit-personalizationConsent-
text"></span>
        </label>
    </div>
  </div>
</div>
```

Note that you can add text that is appropriate for your environment by editing the widget's resource file.

For information on audiences, refer to Define Audiences. For information on customizing the product recommendations widget, refer to the Product Recommendations.

Manage cookie-based consent

Cookie-based consent requests are made when you want to obtain consent from shoppers to use cookies that contain personal data during their site visits. Additionally, you can use cookie-based consent requests while creating personalization consent. For example, by setting the `requireGDPRCookieConsent` site property you can set the consent for receiving cookies. By setting the `requireGDPRP13nConsent` property, you

can set personalization consent.. The need to request consent is based upon the locale of the shopper, and whether the cookie consent property is set to `true`.

Use the Oracle Commerce Admin API `updateSite` to set `requireGDPRCookieConsent` to `true`. By default, this field is set to `false`. For example:

```
PUT /ccadmin/v1/sites/siteUS  HTTP/1.1
{
  "properties":
  {
    "requireGDPRCookieConsent":  true,
    "requireGDPRP13nConsent":  true,
   }
}
```

The following table displays the possibilities when you set the `requireGDPRCookieConsent` property to `true`. When you set the property to true, your consent dialog is displayed to the shopper when they visit the site:

Shopper Response	Effect
Gives consent	If the shopper gives consent, the GDPRCookieP13nConsentGranted cookie is placed on their browser. No cookies are deleted from the shopper's browser.
Does not give consent	If the shopper does **not** give consent, Oracle Commerce cookies that contain personal data will be deleted from the shopper's browser, with the exception of cookies that are identified within the `necessaryCookies` property list in the widget JSON. No further cookies are added to the browser.
GDPR not applicable	If the GDPR is not required, a GDPRCookieP13nConsentNotRequired cookie is placed on the shopper's browser.

Note: It is important to be aware of the cookies that your site uses, and, in particular, which cookies are deployed by third-party software. For a list of Oracle Commerce cookies, refer to Cookies used in Oracle Commerce.

If you have customized any of the following widgets, you may want to update to the latest default widget to get the new fields, or update your customized widgets to include the GDPR consent and profile personalization consent code elements. For information on upgrading customized widgets, refer to the Upgrade deployed widgets section in Design Your Store Layout :

- `customerProfile` widget
- `shopperDetail` widget
- `checkoutRegistration` widget

You can configure your personalization services, such as the audience feature, to look for the presence of the `GDPRProfileP13nConsentGranted` cookie on the shopper's browser and then perform the actions required for your site configuration. Refer to the Manage personalization consent section for information.

Cookie customization example

Oracle Commerce provides access to the file `CivicUKCookieControl_sample.zip` that contains sample code you can use to model the configuration of your site to comply with certain GDPR regulations. You can obtain this file from the Oracle Commerce Developer Community blog posts at:

https://community.oracle.com/groups/oracle-commerce-cloud-group/blog/2017/09/21/how-to-ensure-cookie-compliance-in-commerce-cloud-with-a-custom-widget

This code asks for consent based on the shopper's profile settings and the setting of the `requireGDPRCookieConsent` property. The example uses the Civic UK Cookie Control widget to see if `requireGDPRCookieConsent` is set to `true`.

https://www.civicuk.com/cookie-control/v8/documentation

The Civic UK Cookie Control widget is only an example of one way that you could customize your storefront. Oracle Commerce cannot provide or recommend the types of consent that you need to capture, or the mechanisms that you use to capture them. Work with your legal team to determine your requirements.

Note that you can use any cookie consent application, but when you create cookies, they must use the names specified by Oracle Commerce, the `GDPRCookieP13nConsentNotRequired` and the `GDPRCookieP13nConsentRequired` cookies. The following example makes use of the application Cookie Control. For information, refer to:

The following example provides a customized configuration for the widget JSON:

```
{
  "name": "Cookie Control (Civic UK)",
  "javascript": "cukCookieControl",
  "i18nresources":"cukCookieControl",
  "availableToAllPages": true,
  "global": true,
  "globalEnabled": true,
  "config": {
    "apiKey":"94b985b32474b87b3fc2533a19aadeb8c455d5",
    "product":"free",
    "position":"left",
    "theme":"light",
    "initialState":"open",
    "necessaryCookies":["JSESSIONID", "atgRecVisitorId",
        "oauth_token_secret-storefrontUI", "xdVisitorID"]
  }
}
```

In the above example, the following configurations are set:

- `apiKey` – A Cookie Control API key from the Civic UK Cookie Control API for a particular domain. This is a required property.
- `product` – The type of the Cookie Control license that is set from the widget. The value can be multisite, custom, or the default, free. This is a required property that corresponds to the API key.
- `position` – The position of the consent dialog. For button style widgets, the values can be left or right.

- `initialState` – Identifies that the dialog has to be open when the widget starts.
- `necessaryCookies` – Indicates the list of cookies that need to be protected from the deletion process.

The following is an example of a customized `cukCookieControl.js` file:

```
define(

  //-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
  ['knockout', 'pubsub', 'CCi18n', 'ccConstants',
     'https://cc.cdn.civiccomputing.com/8.0/cookieControl-8.0.min.js',
     'storageApi'],

  //-------------------------------------------------------------------
  // MODULE DEFINITION
  //-------------------------------------------------------------------
  function(ko, pubsub, CCi18n, CCConstants, CC, storageApi) {
    "use strict";

    return {
        widgetInitialised: ko.observable(false),
      onLoad: function(widget) {
        if(widget.apiKey() !== null ) {
          var cukConfig = {};
          cukConfig.apiKey = widget.apiKey();
          switch (widget.product()) {
            case 'free':
              cukConfig.product = "COMMUNITY";
              break;
            case 'multisite':
              cukConfig.product = "PRO_MULTISITE";
              break;
            case 'pro':
              cukConfig.product = "PRO";
              break;
            default:
              cukConfig.product = "COMMUNITY";
          }

          switch (widget.position()) {
            case 'left':
              cukConfig.position = "LEFT";
              break;
            case 'right':
              cukConfig.position = "RIGHT";
              break;
            default:
              cukConfig.position = "LEFT";
          }

          if(widget.position() !== 'left' && widget.position() !== 'right') {
              cukConfig.position = "LEFT";
            }
```

```
          if(widget.theme() === 'light'){
            cukConfig.theme = "LIGHT";
          } else {
            cukConfig.theme = "DARK";
          }

          if(widget.initialState() === 'open'){
            cukConfig.startOpen = "OPEN";
          } else {
            cukConfig.startOpen = "CLOSED";
          }

          cukConfig.optionalCookies = [
                                          {
                                              name : 'analytics',
                                              label : 'Analytical
cookies',
                                              description : 'Analytical
cookies help
                                                 us to improve our
website by
                                                 collecting and reporting
                                                 information on its
usage.',
                                              onAccept :
this.onConsentAccept,
                                              onRevoke :
this.onConsentRevoke
                                          }
                                      ];
          cukConfig.necessaryCookies = widget.necessaryCookies();
          cukConfig.text = {};
          cukConfig.text.title = '<p>' + CCi18n.t('ns.cukCookieControl
             :resources.cccTitle') + '</p>';
          cukConfig.text.intro = '<p>' + CCi18n.t('ns.cukCookieControl
             :resources.cccIntro') + '</p>';

// Main call to invoke Cookie Control, passing the configuration
object we have
// set up. cookieControl object is established by the
cookieControl-6.2.min.js
// file loaded in dependencies. Also subscribe to PAGE_CHANGED so we
can manage
// the use of cookies if user has disallowed their use.
          if(widget.site().requireGDPRCookieConsent){
            if(CookieControl) {
              CookieControl.load( cukConfig );
               $.Topic(pubsub.topicNames.PAGE_CHANGED).subscribe
                   (this.pageChanged.bind(this));
                        this.widgetInitialised(tre);
            }
          }else{
            storageApi.getInstance().saveToCookies
                ("GDPRCookieP13nConsentNotRequired",true,365);
```

```
            if(storageApi.getInstance().readFromCookies
                ("GDPRCookieP13nConsentGranted")!==null){

storageApi.getInstance().removeItem("GDPRCookieP13nConsentGranted");
            }
          }
        }
      },

      pageChanged: function(page){
        if(this.widgetInitialised() &&
storageApi.getInstance().readFromCookies
             ("GDPRCookieP13nConsentGranted")===null){
            CookieControl.deleteAll();
        }
      },
       onConsentAccept: function() {
        storageApi.getInstance().saveToCookies("GDPRCookieP13nConsentGranted"
            ,true,365);
      },

      onConsentRevoke: function() {
        CookieControl.deleteAll();
      }
    };
  }
);
```

This JavaScript makes a call to Civic UK CDN with the required parameters in the `cukConfig` object. In this example, only one category of `optionalCookies` is created with the name of Analytical Cookies. The `onAccept` function for this category sets the `GDPRCookieP13nConsentGranted` cookie and deletes all other cookies except those listed in the `necessaryCookies` list in the `widget.json` file. If more categories are required, you can write separate `onAccept` and `onRevoke` functions for each category.

When a widget loads, if the `requireGDPRCookieConsent` property for cookies is set to false, a `GDPRCookieP13nConsentNotRequired` cookie is created.

The `ns.cukCookieControl.json` file contains example text that you could use for the messages presented to the shopper:

```
{
  "resources": {
    "cccTitle" : "This site uses cookies to store information on your
computer.",
    "cccIntro" : "Some of these cookies are essential to make our site work
and
        others help us to improve by giving us some insight into how the
site is
        being used.",
    "cccFull" : "Click for full Privacy Policy..."
  }
}
```

You can add configurations to the `cukConfig` object. For more configuration information, refer to the Civic UK documentation.

Cookies used in Oracle Commerce

This section describes the cookies that are issued with Oracle Commerce. This list provides information that may assist you when you are configuring your cookie control for shopper consent. It also indicates cookies that should be protected from deletion by adding them to the `necessaryCookies` list, as described in the Cookie customization example.

`FILE_OAUTH_TOKEN` cookie

The `FILE_OAUTH_TOKEN` cookie, which has a life of 24 hours, stores a token that is needed to access files using the `/files` servlet on the administration server. Note that this cookie is for the administration interface only and does not contain any personal data. This cookie can be deleted on the client-side, if necessary. It does not need to be included in the `necessaryCookies` list.

`JSESSIONID` cookie

The `JSESSIONID` cookie, which has a life that lasts only until the user's browsing session ends, helps the server to manage user sessions. It is a standard Java servlet container cookie. While not accessible to scripts, this cookie can be deleted from the client-side. However, the cookie will be re-sent during the next request from the user.

This cookie tracks each request from the same browser, ensuring that the same session data is available on the server side. It does not contain any personal data. You must include this cookie in the `necessaryCookies` list, or else WebLogic will create new sessions for every request that comes in.

`atgRecVisitorId` cookie

This cookie does not collect personal data. It has a life expiration of 20 years. It is accessible to scripts, and can be deleted from the client-side. You must add this cookie to the `necessaryCookies` list.

`oauth_token_secret-storefrontUI` cookie

The `oauth_token_secret-storefrontUI` cookie is necessary for storefront user interface operations, as it is used to store the OAuth token of the user that is logged in and keeps the shopper's login token active during page reloads and multiple tab access. This cookie does collect personal data in the form of the `profileId`. While the cookie is accessible from scripts, it cannot be deleted from the client-side. If you delete this cookie, shoppers may have to log in again after opening new tabs or refreshing pages. Deleting this cookie would also cause some checkout payment flows to fail when a shopper gets redirected to an external payment site like PayPal. When the browser gets returned to the storefront, the shopper's authentication state is lost and the checkout process cannot proceed. You should add this cookie to the `necessaryCookies` list.

`EETrViID` cookie

The `EETrViID` cookie is an Oracle WebLogic cookie that stores the Visitor ID. It does not contain any personal data. This cookie cannot be deleted, and therefore cannot be modified by JavaScript in the browser. This cookie does not need to be added to the `necessaryCookies` list.

`BIGIP` cookie

The `BIGIP` cookie is an HTTP Only cookie that maintains a connection to a single application instance. This cookie is not accessible from scripts and contains no personal data. This cookie expires at the end of the session.

`xd[tenantID]_[siteID]` cookie

These cookies are generated by Visitor ID services and track the visits, which are site-specific. This cookie should be added to the protected list as it does not collect personal data. Note that the `_[siteID]` is only added to the cookie name if your environment supports multiple sites. You should know your own tenant ID and site ID.

For example: `xdtp6a0c0_siteUS`, where `xdtp6a0c0` is the tenant ID and `_siteUS` is the site ID.

`xs[tenantID]_[cartSharingGroupId]` cookie

These cookies are used to find the current incomplete order for an anonymous shopper when the current site is in a cart sharing group. They do not collect personal data.

Soft Login cookie

The Soft Login cookie, which has a life of 13 months, contains a cryptographically secure version of the expiration `timestamp` and the user's profile ID. If the shopper does not provide consent, the soft login cookie is not added to their browser, and soft login will not occur. This cookie does collect personal data, and therefore should not be included in the `necessaryCookies` list. If you delete this cookie, the soft login capability will not function. For information on soft login, refer to Configure the logged-in shopper session. For information on disabling the soft login feature, see Disable soft login.

Configure consent for account-based commerce

If your environment is configured with business accounts, as described in the Configure Business Accounts, you may want to configure profile properties that enable consent requests for account-based contacts. It is up to you to determine what types of consent to gather. For example, you may want to allow some accounts to grant consent on behalf of their contacts.

For example, you could configure your site to recognize when an account-based contact logs in for the first time and present them with various consent requests. Once you have consent, contacts who have visited before are not presented with additional consent checkboxes unless you configure it otherwise.

By default, when an account-based contact logs in, the Contact Login element of the Header widget checks to see if this is the contact's first login. If it is, the contact is presented with a checkbox for consent to receive marketing emails, and a checkbox for personalization consent if the site has been configured to require consent.

Note that should it be necessary, you can provide agents and delegated administrators with the ability to use the `getMember` and `updateMember` endpoints in the Admin REST API to update a shopper's consent properties. Administrators may also use the `updateProfile` endpoint to update an account-based shopper's consent properties. For information on configuring custom properties, see Manage Shopper Profiles.

Delete shopper information

In addition to consent requirements, the GDPR also ensures a shopper the right to erasure.

The right to erasure requires you to delete all data about a shopper on your sites if the shopper requests it. To support this right, Commerce provides endpoints in the Agent and Admin API that enable removal of personal data for consumer-based commerce shoppers and account-based commerce contacts. Using these endpoints, you can do the following:

- Redact orders. Personal information about the shopper in a specified order is removed from the order and replaced with new data that does not identify the shopper. The order itself is retained in the system for reporting purposes.
- Delete orders. The order and its constituent objects (such as line items, shipping groups, and payment groups) are removed from the server entirely and cannot be recovered. Note that only internal users who have the Administrator role can use the order deletion endpoints.
- Delete or redact other objects that may contain personal data, such as purchase lists.
- Delete profiles of registered shoppers or contacts. The shopper's profile is deleted and personal data is discarded. For a registered shopper or contact, it is recommended that you delete or redact orders and other objects that contain personal data before deleting the shopper's profile.
- Delete notification requests for shopper-initiated email notifications for back in stock products, using either notification request ID, or using a profile ID/email ID.

Note that redaction and deletion are supported for orders associated with guest shoppers as well as registered shoppers.

This section describes how to use the REST APIs to delete shopper and contact information from your sites. It includes the following topics:

- Redact orders
- Delete orders
- Delete shopper profiles
- Delete contact data for account-based commerce
- List of non-redactable order properties

Important: Consult legal counsel for professional guidance about maintaining compliance with the GDPR right to erasure.

Redact orders

Commerce provides endpoints in the Agent API for redacting orders.
These endpoints overwrite shopper information in orders and replace it with automatically generated data.

To redact an order, you need its order ID. To find orders for a specific registered shopper, use the `getOrders` endpoint in the Admin API or the `searchOrders` endpoint in the Agent API. (The Admin API `getOrders` endpoint searches all orders, including orders needing approval, order quotes, and scheduled orders.) For example:

```
GET /ccadmin/v1/orders?queryFormat=SCIM&q=profileId eq "110658"
HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

The response returns the orders associated with the specified profile ID. You can use these results to find the IDs of the orders you want to redact.

For some shoppers, you may not have a profile ID. This would be the case if you already deleted the shopper's profile, or if the shopper never registered and instead checked out as a guest. If so, you can find the shopper's orders by using the `q` query parameter to search for orders that contain a specific property value. For example:

```
GET /ccadmin/v1/orders?queryFormat=SCIM&q=shippingGroups.lastName eq
"Brady"  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

To redact an individual order, you use the `redactOrder` endpoint in the Agent API, and include the order ID as a path parameter in the URL of the request. You specify the properties to redact in the `properties` array in the body of the request. For example:

```
POST /ccagent/v1/orders/o10042/redact  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
"properties": [
   "shippingGroups.lastName",
   "shippingGroups.city"
   ]
}
```

You can redact multiple orders in one request using the `redactOrders` endpoint in the Agent API. You specify the orders to redact in the `orderIds` array in the body of the request, and the properties to redact in the `properties` array. For example:

```
POST /ccagent/v1/orders/redact  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
  "orderIds": [
    "o10007",
    "o10042",
    "o10312",
    "o10842"
  ],
"properties": [
   "shippingGroups.lastName",
   "shippingGroups.city"
   ]
}
```

Redactable and non-redactable properties

When you redact an order, you should remove all personal data. (Consult legal counsel to determine which data should be considered personal.) Other data, however, should remain

intact, so the order can be used in reporting. For example, the items purchased and the cost of the order are useful for tracking sales, and can be retained once they are no longer associated with a specific shopper.

To protect shopper privacy without discarding key data, Commerce designates which properties can be redacted and which cannot. For a complete list of the non-redactable properties, see the List of non-redactable order properties. Any order property that does not appear in this list can be redacted, including any custom properties.

Note that if you attempt to redact a property that is not redactable, the call will fail, and none of the specified properties will be redacted. For example:

```
POST /ccagent/v1/orders/o10057/redact  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
"properties": [
   "shippingGroups.lastName",
   "shippingGroups.city",
   "shippingGroups.priceInfo.currencyCode"
   ]
}
```

The response indicates which of the specified properties are not redactable:

```
{
    "errorCode": "28403",
    "message": "The following properties are blocked from redaction,
please
       remove from the set of properties to be redacted -
       [shippingGroups.priceInfo.currencyCode]",
    "status": "400"
}
```

Redacted values

When you redact an order property, the value used to overwrite the existing value depends on the data type. The following table lists the various data types and the values they are set to when redacted:

Type	Redacted Value
Integer	0
Long	0
Double	0.0
Float	0.0
Boolean	null
Date	Jan 1 12:00:00 GMT 1970
Timestamp	Thursday, January 1, 1970 12:00:00 AM
String	Randomly generated string
Enumeration	Cannot be redacted

There are some properties that are exceptions to the values in the table:

- `profileId` – The `profileId` property for the first order you redact is set to `redact100001`, and for subsequent orders it is set to `redact100002`, `redact100003`, and so on. If you use the `redactOrders` endpoint to redact multiple orders in a single call, then all orders specified in the call that have the same `profileId` are given the same redacted value, as described in Understand pseudonymization and anonymization
- `creditCardNumber` – The redacted value of the `creditCardNumber` property is always `xxxxxxxxxxxx1111`.
- `approverIds` (account-based orders only) – The redacted value of the `approverIds` property is the empty string.

Also, note that if you redact a Boolean custom property that has a default value, the property is set to the default value rather than to null.

Understand pseudonymization and anonymization

During redaction, string property values are replaced with automatically generated random strings. There are some differences you should be aware of between how redaction is done within a single call and across multiple calls.

If you redact orders individually (either by using the `redactOrder` endpoint or by using `redactOrders` and specifying a single order ID in each call), then there is no carry over of redacted string values from one call to the next. For example, even if all of the orders to be redacted have the same value for the `shippingGroups.email` property, the value this property is set to during redaction will be different in each call (and therefore for each order). The orders are *anonymized* so that they have no apparent connection to each other. The drawback of anonymization is that information that may be useful for reporting (such as the number of orders associated with a specific email address) is lost.

If you use the `redactOrders` endpoint to redact multiple orders in a single call, however, the same string is used in all of these orders to redact the same value. So if the value of `shippingGroups.email` is the same in each order before redaction, the same random string will be used in each order to redact this property. Although the original value is replaced, the fact that all of the orders had the same value for the property will be preserved. In this case, the order is *pseudonymized*, indicating that a single random string is used as a pseudonym for the original value. The drawback of pseudonymization is that someone examining the orders may be able to draw inferences about the original values. Hence pseudonymization is somewhat less secure than anonymization. Keep in mind, too, that pseudonymization applies only to orders redacted in the same REST call. If you make a subsequent call to redact additional orders, the redacted values used in the first call are not carried over to the second.

Redact scheduled orders

A scheduled order consists of two parts: an order template that specifies the items to include in the orders that are created, and a scheduling object that determines when those orders are submitted. You can use the Scheduled Orders endpoints in the Agent API to delete the scheduling object for a scheduled order, to ensure that no further orders are created from the template. First, use the `listScheduledOrdersByProfile` endpoint to find the shopper or contact's scheduled orders. You can then use the `deleteScheduledOrders` endpoint to delete the schedules for those orders.

To remove personal data in the template, you can redact the order template, just as you would any other order. However, if you want to redact orders that have been created from the template, you must locate them and specify them separately.

For more information about scheduled orders, see Create Scheduled Orders. For more information about the Scheduled Orders endpoints in the Agent API, see the REST API documentation in the Oracle Help Center.

Order Redact webhook

Oracle Commerce includes an Order Redact event webhook. You can configure this webhook to notify external systems when an order is redacted. The webhook payload contains the order ID of the redacted order. For example:

```
{
  "orderId": "o30411"
}
```

See the Use Webhooks chapter for information about configuring and using webhooks.

Delete orders

Internal users who have the Administrator access role can use REST endpoints to delete orders. When an order is deleted, the order and its constituent objects (such as line items, shipping groups, and payment groups) are removed from the server entirely, so any reports you generate will not take into account data from the order. If you want to remove personal information from orders while still retaining other data for reports, you can redact the orders instead.

To delete an order, you need its order ID. To find orders for a specific registered shopper, use the `getOrders` endpoint in the Admin API or the `searchOrders` endpoint in the Agent API, as described in Redact orders.

To delete an individual order, you use the `deleteOrder` endpoint in the Admin API, and include the order ID as a path parameter in the URL of the request. For example:

```
DELETE /ccadmin/v1/orders/o10042  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

You can delete multiple orders in one request using the `deleteOrders` endpoint in the Admin API. You specify the orders to delete in the `orderIds` array in the body of the request. For example:

```
POST /ccadmin/v1/orders/delete  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
  "orderIds": [
    "o10007",
    "o10042",
    "o10312",
    "o10842"
  ]
}
```

Delete scheduled orders

A scheduled order consists of two parts: an order template that specifies the items to include in the orders that are created, and a scheduling object that determines when those orders are submitted. When a new order is created from the template, the order's `createdByOrderId` property contains a reference to the template. To delete a scheduled order, use the `deleteOrder` or `deleteOrders` endpoints to delete the order template. When you delete an order template, the associated scheduling object is deleted as well. However, if you want to delete orders that have been created from the template, you must locate them and specify them separately.

Delete returns and exchanges

When a shopper returns an item, a return request object is created and associated with the order. If the shopper makes multiple returns against one order, each return creates a new return request object that is added to the original order.

When a shopper requests an exchange, a return request object and a new order are created. The return request contains a reference to the follow-up order. If the shopper subsequently requests a return or exchange for the follow-up order, a new return request object is created for that order. Return requests and exchange orders can be chained together without limit, and each return request can have its own chain of exchanges and further returns.

When you delete an order, any return request objects belonging to the order are also deleted. But if a return request object has a follow-up (exchange) order associated with it, that order is not deleted when the original order is deleted. You must locate any exchange orders and delete them separately.

Delete quotes

When a shopper requests a quote, a `quoteInfo` object is created and associated with the order, and the order state changes to `PENDING_QUOTE`. When the quote is received, the original order and its `quoteInfo` object are copied to create a new order. (The copy's order state is thus `PENDING_QUOTE`, and its `quoteInfo` reflects the original `quoteInfo`.) The state of the original order is then changed to `QUOTED`, and the original order and `quoteInfo` object are updated to reflect the quoted values.

Each time the shopper requests a `requote`, a new `QUOTED` order and `quoteInfo` are created. The result is that there is one order for every quote, plus a single order whose state is `PENDING_QUOTE`. Each of these orders has at most one `quoteInfo`.

When a quoted order is deleted, its associated `quoteInfo` is also deleted. Other orders created through the quoting process are not deleted automatically. You must locate these orders and delete them separately.

Order Delete webhook

Oracle Commerce includes an Order Delete event webhook. You can configure this webhook to notify external systems when an order is deleted. The webhook payload contains the order ID of the deleted order. For example:

```
{
  "orderId": "o30419"
}
```

See the Use Webhooks chapter for information about configuring and using webhooks.

Delete shopper profiles

Commerce provides endpoints in the Agent API for deleting shopper profiles. These endpoints delete profiles securely to ensure they can no longer be accessed. Note that once a profile is deleted, it cannot be accessed on any site running on your Commerce instance.

Important: Before you delete a shopper's profile, you should first redact or delete all of that shopper's orders. Deleting a profile does not automatically remove personal data from orders associated with it.

To delete a profile, you need its profile ID. You can find the profile by using the `searchProfiles` endpoint in the Agent API. Use the `q` query parameter to search for profiles that contain a specific property value. For example:

```
GET /ccagent/v1/profiles?queryFormat=SCIM&q=email eq
"floeb@example.com"  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

The response includes the profile ID:

```
{
  . . .
    "items": [
        {
            "firstName": "Fred",
            "lastName": "Loeb",
            "profileType": null,
            "repositoryId": "110332",
            "shippingAddress": {
                "phoneNumber": "617-555-1212",
                "postalCode": "01012",
                "repositoryId": "130417"
            },
            "id": "110332",
            "email": "floeb@example.com",
        }
    ]
}
```

To delete a single profile, you use the `deleteProfile` endpoint in the Agent API. You specify the profile to delete by including the profile ID as a path parameter in the URL of the request. For example, to delete the profile shown above:

```
DELETE /ccagent/v1/profiles/110332  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

To delete multiple profiles with a single request, you use the `deleteProfiles` endpoint in the Agent API. You specify the profiles to delete in the `profileIds` array in the body of the request. For example:

```
DELETE /ccagent/v1/profiles  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
  "profileIds": [
    "110332",
    "150027",
    "160035"
  ]
}
```

Delete or redact purchase lists

You can use the Purchase List endpoints in the Agent API to delete or redact purchase lists associated with a specific shopper or contact. To find the purchase lists, use the `getPurchaseList` endpoint. You can then use the `deletePurchaseList` endpoint to delete the shopper's purchase lists individually, by providing the purchase list ID as a path parameter. As an alternative, you can use the `updatePurchaseList` endpoint to redact any personal data in the purchase list by overwriting it.

For more information about purchase lists, see the Enable Purchase Lists chapter. For more information about the Purchase List endpoints in the Agent API, see the REST API documentation in the Oracle Help Center.

Delete back in stock notification requests

When you delete a shopper profile, you may also want to delete any email notification requests associated with the shopper for back in stock products. Commerce provides two endpoints in the Admin API for this purpose.

The `deleteProductNotification` endpoint deletes a single notification request whose ID is specified as a path parameter. For example:

```
DELETE /ccadmin/v1/productnotify/330007  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

The `deleteProductNotificationByProfileIdOrEmail` endpoint deletes all of the notification requests that match a specific email address or profile ID. The email address can be specified using the `email` query parameter, or the profile ID can be specified using the `profileId` query parameter. For example, to delete all of the notification requests associated with a specific profile ID:

```
DELETE /ccadmin/v1/productnotify?profileId=160035  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

For more information about these endpoints, see the REST API documentation in the Oracle Help Center.

Shopper Profile Delete webhook

Oracle Commerce includes a Shopper Profile Delete event webhook. You can configure this webhook to notify external systems when a profile is deleted. The webhook payload contains the profile ID of the deleted profile. For example:

```
{
  "profileId": "110658"
}
```

See the Use Webhooks chapter for information about configuring and using webhooks.

Delete contact data for account-based commerce

If any of your sites support account-based commerce, you have shoppers called contacts who are associated with specific accounts. You can delete the profiles for contacts just as you would for consumer-based commerce shoppers. You can also redact or delete orders associated with contacts.

However, there are some additional considerations you must be aware of when you delete contacts and redact or delete data associated with their orders. This section discusses other steps you should perform and additional data you should delete or redact. It includes the following topics:

- Delete a contact
- Delete an approver
- Redact account and contact registration requests

Note that if your sites store personal data in other account-based objects such as organizations, you can also use endpoints in the REST APIs to overwrite this data.

For more information about account-based commerce, see Configure Business Accounts.

Delete a contact

Before deleting a contact's profile, there are a few related actions you should perform:

- Redact or delete any orders associated with the contact. Note that account-based orders have a few additional properties (`approverMessages`, `approvalSystemMessages`, `approverIds`, and `organizationId`) not found in consumer-based orders. These additional properties are all redactable.
- Use the administration interface or Admin API to redact any registration requests associated with the contact. See Redact registration requests.
- Redact or delete the contact's scheduled orders.
- Manually reject any orders submitted by the contact that are still pending approval.
- Redact or delete the contact's purchase lists.

Delete an approver

A contact's profile cannot be deleted if the contact has the Approver role for any account. You must first remove the Approver role from the contact for all accounts before deleting the profile. You can do this in the administration interface or by using the Admin API. Note that if there is only one approver for an account, you cannot

remove the Approver role from that contact until you add another approver to the account.

Before you delete an approver's profile, you should remove the approver's ID from orders. To find the orders that have been approved by this approver, use the `getOrders` endpoint in the Admin API. For example:

```
GET /ccadmin/v1/orders?queryFormat=SCIM&q=approverIds co "bb-110031"
HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

Once you have the list of orders, use the `redactOrder` or `redactOrders` endpoint to redact the `approverIds` property. For example, to redact the `approverIds` property on a single order:

```
POST /ccagent/v1/orders/o10051/redact  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
"properties": [
   "approverIds"
   ]
}
```

When you redact an order's `approverIds` property, its value is set to the empty string.

Redact registration requests

If a contact has submitted any registration requests, you should redact these requests before deleting the contact's profile. Note that you should reject or approve any pending registration requests before redacting them or deleting the contact. If you delete a contact that has any pending requests, you will only be able to reject those requests afterward.

You can use the Organization Requests endpoints in the Admin API to redact a contact's registration requests. Use the `listOrganizationRequests` endpoint to find the relevant requests, and then use the `updateOrganizationRequests` endpoint to overwrite the fields you want to redact.

List of non-redactable order properties

The following is a list of all of the order properties whose values cannot be redacted:

```
id
state
creationDate
createdByOrderId
submittedDate
lastModifiedDate
completedDate
priceInfo
taxPriceInfo
siteId
locale
priceGroupId
taxExempt
```

```
taxCalculated
externalOrderPriceDetails
recurringChargePriceInfo
secondaryCurrencyCode
exchangeRate
catalogId
commerceItems.commerceItemId
commerceItems.catalogId
commerceItems.catalogRefId
commerceItems.catalogKey
commerceItems.productId
commerceItems.siteId
commerceItems.quantity
commerceItems.state
commerceItems.stateDetail
commerceItems.productTaxCode
commerceItems.externalPriceDetails
commerceItems.quantityWithFraction
commerceItems.recurringChargePriceInfo
commerceItems.externalRecurringChargeDetails
priceInfo.rawSubTotal
priceInfo.tax
priceInfo.shipping
priceInfo.manualAdjustmentTotal
priceInfo.type
priceInfo.currencyCode
priceInfo.amount
priceInfo.discounted
priceInfo.amountIsFinal
priceInfo.finalReasonCode
priceInfo.adjustments.adjustmentDescription
priceInfo.adjustments.pricingModel
priceInfo.adjustments.manualPricingAdjustment
priceInfo.adjustments.coupon
priceInfo.adjustments.totalAdjustment
priceInfo.adjustments.quantityAdjusted
priceInfo.adjustments.quantityWithFractionAdjusted
commerceItems.priceInfo.listPrice
commerceItems.priceInfo.rawTotalPrice
commerceItems.priceInfo.salePrice
commerceItems.priceInfo.onSale
commerceItems.priceInfo.orderDiscountShare
commerceItems.priceInfo.quantityDiscounted
commerceItems.priceInfo.quantityAsQualifier
commerceItems.priceInfo.priceList
commerceItems.priceInfo.discountable
commerceItems.priceInfo.shippingSurcharge
commerceItems.priceInfo.quantityWithFractionAsQualifier
commerceItems.priceInfo.quantityWithFractionDiscounted
commerceItems.priceInfo.currentPriceDetails
commerceItems.priceInfo.type
commerceItems.priceInfo.currencyCode
commerceItems.priceInfo.amount
commerceItems.priceInfo.discounted
commerceItems.priceInfo.amountIsFinal
```

```
commerceItems.priceInfo.finalReasonCode
commerceItems.priceInfo.adjustments.adjustmentDescription
commerceItems.priceInfo.adjustments.pricingModel
commerceItems.priceInfo.adjustments.manualPricingAdjustment
commerceItems.priceInfo.adjustments.coupon
commerceItems.priceInfo.adjustments.adjustments.totalAdjustment
commerceItems.priceInfo.adjustments.quantityAdjusted
commerceItems.priceInfo.adjustments.quantityWithFractionAdjusted
taxPriceInfo.cityTax
taxPriceInfo.countyTax
taxPriceInfo.stateTax
taxPriceInfo.countryTax
taxPriceInfo.valueAddedTax
taxPriceInfo.miscTax
taxPriceInfo.isTaxIncluded
taxPriceInfo.secondaryCurrencyTaxAmount
taxPriceInfo.type
taxPriceInfo.currencyCode
taxPriceInfo.amount
taxPriceInfo.discounted
taxPriceInfo.amountIsFinal
taxPriceInfo.finalReasonCode
taxPriceInfo.adjustments.adjustmentDescription
taxPriceInfo.adjustments.pricingModel
taxPriceInfo.adjustments.manualPricingAdjustment
taxPriceInfo.adjustments.coupon
taxPriceInfo.adjustments.totalAdjustment
taxPriceInfo.adjustments.quantityAdjusted
taxPriceInfo.adjustments.quantityWithFractionAdjusted
shippingGroups.priceInfo.rawShipping
shippingGroups.priceInfo.shippingTax
shippingGroups.priceInfo.secondaryCurrencyTaxAmount
shippingGroups.priceInfo.type
shippingGroups.priceInfo.currencyCode
shippingGroups.priceInfo.amount
shippingGroups.priceInfo.discounted
shippingGroups.priceInfo.amountIsFinal
shippingGroups.priceInfo.finalReasonCode
shippingGroups.priceInfo.adjustments.adjustmentDescription
shippingGroups.priceInfo.adjustments.pricingModel
shippingGroups.priceInfo.adjustments.manualPricingAdjustment
shippingGroups.priceInfo.adjustments.coupon
shippingGroups.priceInfo.adjustments.totalAdjustment
shippingGroups.priceInfo.adjustments.quantityAdjusted
shippingGroups.priceInfo.adjustments.quantityWithFractionAdjusted
commerceItems.currentPriceDetails.tax
commerceItems.currentPriceDetails.orderDiscountShare
commerceItems.currentPriceDetails.orderManualAdjustmentShare
commerceItems.currentPriceDetails.quantityAsQualifier
commerceItems.currentPriceDetails.quantityWithFractionAsQualifier
commerceItems.currentPriceDetails.quantityWithFraction
commerceItems.currentPriceDetails.secondaryCurrencyTaxAmount
commerceItems.currentPriceDetails.type
commerceItems.currentPriceDetails.currencyCode
commerceItems.currentPriceDetails.amount
```

```
commerceItems.currentPriceDetails.discounted
commerceItems.currentPriceDetails.amountIsFinal
commerceItems.currentPriceDetails.finalReasonCode
commerceItems.currentPriceDetails.adjustments.adjustmentDescription
commerceItems.currentPriceDetails.adjustments.pricingModel
commerceItems.currentPriceDetails.adjustments.manualPricingAdjustment
commerceItems.currentPriceDetails.adjustments.coupon
commerceItems.currentPriceDetails.adjustments.totalAdjustment
commerceItems.currentPriceDetails.adjustments.quantityAdjusted
commerceItems.currentPriceDetails.adjustments.quantityWithFractionAdjus
ted
paymentGroups.paymentGroupClassType
paymentGroups.paymentMethod
paymentGroups.amount
paymentGroups.amountAuthorized
paymentGroups.amountDebited
paymentGroups.amountCredited
paymentGroups.currencyCode
paymentGroups.state
paymentGroups.submittedDate
paymentGroups.cancelledDate
shippingGroups.shippingGroupClassType
shippingGroups.shippingMethod
shippingGroups.state
shippingGroups.submittedState
relationships.shippingGroup
relationships.commerceItem
relationships.quantity
relationships.returnedQuantity
relationships.amount
relationships.state
relationships.quantityWithFraction
relationships.returnedQuantityWithFraction
externalPriceDetails.externalPrice
externalPriceDetails.externalPriceQuantity
```

33

Implement Access Control for Internal Users

Oracle Commerce includes a role-based access control system that is applied to all internal users such as merchandisers and administrators.

Each internal user can be assigned one or more roles. Depending on the roles assigned, the user may have access to a number of areas of Commerce. These include:

- Specific pages of the administration interface and the Agent Console, and the functionality available on those pages.
- Catalogs and price groups.
- Properties of shopper profiles.

Commerce includes several predefined roles, and you can create additional custom roles. Roles primarily function as containers for various entities that are used to control access. These entities are:

- privileges – Predefined rights that grant access to specific functionality. Each predefined role has a single privilege assigned to it. Merchants can assign privileges to custom roles, but cannot create new privileges, or edit or delete existing privileges.
- security criteria – Merchant-defined restrictions on the data access granted by privileges. For example, if a role has the Catalog privilege, a merchant might create a security criterion that restricts access to specific catalogs and assign the security criterion to that role, or create a separate role and assign the security criterion to it.
- generic access rights – Merchant-defined rights that can be used to control read and write access to the properties of shopper profiles. Generic access rights are particularly useful for ensuring compliance with consumer privacy laws such as the European Union General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA).

Note that although privileges and access rights are used for different purposes, privileges are treated internally as a special type of access right. So, for example, the same endpoint property is used to add privileges and access rights to a custom role. To reduce the possibility of confusion, access rights for controlling property access are referred to as generic access rights.

This chapter describes how to create and modify these elements of access control and assign them to roles. For information about assigning roles to internal users and implementing an overall access control strategy, see Understand Role-based Access Control.

Use and modify roles for internal users

You can use predefined roles and create custom roles. You can also modify roles by adding security criteria and generic access rights.

This section describes the predefined roles included with Commerce and how to create and modify custom roles. For information about assigning roles to internal users, see Understand Role-based Access Control.

Use predefined roles

Commerce includes a number of predefined roles. Each predefined role contains a privilege of the same name that provides access to a certain area of the administration interface. For example, the Design role contains the Design privilege, which gives access to the Design page in the administration interface to users assigned the role. You cannot remove the privilege from a predefined role or add a different one to it. If you want a role to contain multiple privileges, you must create a custom role, as described below.

Note that you can add security criteria and generic access rights to a predefined role, but a more flexible approach is to leave the predefined roles as is and add security criteria and generic access rights to custom roles only.

The following table describes the predefined roles and their corresponding privileges. Note that each role and privilege has an ID which is used to identify it in API calls.

Role and Privilege Name	Role ID	Privilege ID	Description
Administrator	adminRole	ora.adminPrivilege	Full access to the Commerce administration interface.
CS Agent	csAgentRole	ora.csAgentPrivilege	Full access to the Agent Console except Manual Adjustments. (Available only if your subscription includes the Agent Console.)
CS Agent Supervisor	csAgentSupervisorRole	ora.csAgentSupervisorPrivilege	Full access to the Agent Console. (Available only if your subscription includes the Agent Console.)
Account Manager	accountManagerRole	ora.accountManagerPrivilege	Full access to the Accounts page. (This feature may not be enabled in your environment.)
Catalog	catalogRole	ora.catalogPrivilege	Full access to the Catalog page. You must also assign the Media privilege if the user will upload images for products, SKUs, and collections.

Role and Privilege Name	Role ID	Privilege ID	Description
Dashboard	dashboardRole	ora.dashboardPrivilege	Read access to the dashboard, which is the landing page for the administration interface that users see when they log into Commerce. All privileges that control access to the administration interface allow the user to see the dashboard, but only the Dashboard or Administrator privilege allows a user to see the summary reports there. A user who can see the summary reports on the dashboard needs the Reporting or Administrator privilege to click through to view the full reports.
Design	designRole	ora.designPrivilege	Full access to the Design page.
Marketing	marketingRole	ora.marketingPrivilege	Full access to the Marketing page.
Media	mediaRole	ora.mediaPrivilege	Full access to the Media page.
Operations	operationsRole	ora.operationsPrivilege	Full access to the Service Operations page.
Preview	previewRole	ora.previewPrivilege	Access to the Preview button.
Publishing	publishingRole	ora.publishingPrivilege	Full access to the Publishing page.
Reporting	reportingRole	ora.reportingPrivilege	Full access to the Reports page.
Search	searchRole	ora.searchPrivilege	Full access to the Search page
Settings	settingsRole	ora.settingsPrivilege	Full access to all Settings pages except Access Control, Extensions, Extension Settings, Email Settings, and Web APIs. (Access to those settings is granted by the Administrator privilege.)

Create custom roles

In addition to providing the predefined roles described above, Commerce provides support for creating custom roles. Unlike predefined roles, you can delete custom roles, and you can add privileges to and remove privileges from them.

To create a custom role, use the `createAdminRole` endpoint in the Admin API, as shown in the following example. As this example shows, when you create a role, you can assign privileges, security criteria, and access rights in the same call. Alternatively, you can create the role and later assign these entities, as described in Modify roles.

```
POST /ccadmin/v1/adminRoles  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "repositoryId": "HomeGoodsMerchandiser",
  "securityCriteria": [
    {
      "id": "securityCriterionHomeGoodsCatalog"
    },
    {
      "id": "securityCriterionHomeGoodsPriceGroup"
    }
  ],
  "accessRights": [
    {
      "repositoryId": "ora.catalogPrivilege"
    },
    {
      "repositoryId": "ar1"
    }
  ],
  "name": "Home Goods Merchandiser"
}
```

The response is similar to this:

```
{
    "securityCriteria": [
        {
            "name": "Security Criterion for HomeGoodsPriceGroup",
            "description": "Grants access to HomeGoodsPriceGroup",
            "id": "securityCriterionHomeGoodsPriceGroup"
        },
        {
            "name": "Security Criterion for HomeGoodsCatalog",
            "description": "Grants access to catalog assets of the
HomeGoodsCatalog",
            "id": "securityCriterionHomeGoodsCatalog"
        }
    ],
    "name": "Home Goods Merchandiser",
    "repositoryId": "HomeGoodsMerchandiser",
```

```
    "description": null,
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/adminRoles"
        }
    ],
    "accessRights": [
        {
            "displayName": "Access Right 1",
            "name": "ar1",
            "repositoryId": "ar1",
            "description": "First of several access rights.",
            "id": "ar1",
            "type": "generic"
        },
        {
            "displayName": "Catalog",
            "name": "Catalog",
            "repositoryId": "ora.catalogPrivilege",
            "description": "Catalog Privilege",
            "id": "ora.catalogPrivilege",
            "type": "privilege"
        }
    ],
    "securityCriteriaLastModified": "2021-03-08T21:12:30.047Z",
    "id": "HomeGoodsMerchandiser",
    "category": "Custom"
}
```


Modify roles

You can use the `updateAdminRole` endpoint in the Admin API to modify predefined roles or custom roles by adding or removing security criteria and generic access rights. For custom roles, you can also use this endpoint to add and remove privileges.

Note that when you call this endpoint, the body of the request can include the `securityCriteria` property, the `accessRights` property, or both. If you include one of these properties in the call, the entities specified in the property replace the ones currently assigned to the role. So, for example, if you include the `securityCriteria` property to add a security criterion to a role, the value of the property should include all of the security criteria that are currently assigned to the role, in addition to the security criterion you want to add; if you specify only the new criterion, the existing criteria will be removed from the role. Similarly, if you include the `accessRights` property to add a privilege or a generic access right, the value of the property should include the privileges and generic access rights that are currently assigned to the role. If you add a generic access right to a predefined role, be sure to include the role's privilege in the `accessRights` property, or the request will be rejected.

Create security criteria

Security criteria can be assigned to roles to restrict access to specific catalogs or price groups.

A security criterion restricts the specific data a user can access. For example, a user might be assigned the Catalog role and also a custom role containing a security criterion that limits the user to modifying only one specific catalog. Another role might have a security criterion that denies update access to two specific price groups but permits update access to all others.

Currently, security criteria affect users who have roles with the Catalog privilege or Administrator privilege, and restrict update access only, not read access. Any user with one of these privileges can view any catalog or price group. Also, if a user has one of these privileges but no security criteria, the user can update any catalog or price group.

This chapter describes how to create and view security criteria. The use of security criteria to control update access to catalogs and price groups is described in detail in Understand Role-based Access Control.

Create security criteria for catalogs

An individual security criterion applies either to catalogs or to price groups, and includes one of these constraint types:

- `grant` – Grants update access to specific catalogs or price groups.
- `deny` – Denies update access to specific catalogs or price groups.
- `grantNone` – Denies create and update access to all catalogs or price groups.

The following example illustrates using the `createAdminSecurityCriterion` endpoint in the Admin API to create a security criterion for catalogs. This criterion grants update access to a single catalog. Note the following properties in the payload:

- `id` – The ID to give the security criterion you are creating.
- `securityCriteriaResource.id` – The ID of the type of resource the security criterion applies to. Specify `ora.catalogAssetResource` for catalogs or `ora.pLGAssetResource` for price groups.
- `actions` – An array specifying the actions the criterion affects. Currently this must be `["create", "update", "delete"]`, as shown in the example below. You can specify this explicitly, or you can omit the property and it will default to this value.
- `constraintType` – The type of constraint the security criterion includes. See the list above.
- `constraints.constraintConfig.id` – The ID of the asset type the security criterion applies to. Specify `ora.catalogConstraintConfiguration` for catalogs or `ora.pLGConstraintConfiguration` for price groups.
- `constraints.values` – An array of the IDs of the specific catalogs or price groups the security criterion applies to.

```
POST /ccadmin/v1/adminSecurityCriteria  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
  "id": "securityCriterionHomeGoodsCatalog",
  "name": "Security Criterion for HomeGoodsCatalog",
  "description": "Grants access to catalog assets of the
HomeGoodsCatalog",
```

```
  "securityCriteriaResource": {
    "id": "ora.catalogAssetResource"
  },
  "actions": [
    "create",
    "update",
    "delete"
  ],
  "constraintType": "grant",
  "constraints": [
    {
      "constraintConfig": {
        "id": "ora.catalogConstraintConfiguration"
      },
      "values": [
        "HomeGoodsCatalog"
      ]
    }
  ]
}
```

The response is similar to the following:

```
{
    "constraintType": "grant",
    "roles": [],
    "name": "Security Criterion for HomeGoodsCatalog",
    "description": "Grants access to catalog assets of the HomeGoodsCatalog",
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/
adminSecurityCriteria"
        }
    ],
    "securityCriteriaResource": {
        "name": "Catalog Assets",
        "id": "ora.catalogAssetResource"
    },
    "lastModified": "2021-03-08T19:52:43.752Z",
    "id": "securityCriterionHomeGoodsCatalog",
    "constraints": [
        {
            "values": [
                "HomeGoodsCatalog"
            ],
            "constraintConfig": {
                "id": "ora.catalogConstraintConfiguration"
            },
            "id": "scc-100001"
        }
    ],
    "actions": [
        "update",
```

```
        "delete",
        "create"
    ]
}
```

Create security criteria for price groups

The following example illustrates using the `createAdminSecurityCriterion` endpoint to create a security criterion for a price group. This criterion grants update access to a single price group.

```
POST /ccadmin/v1/adminSecurityCriteria  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

 {
  "id": "securityCriterionHomeGoodsPriceGroup",
  "name": "Security Criterion for HomeGoodsPriceGroup",
  "description": "Grants access to HomeGoodsPriceGroup",
  "securityCriteriaResource": {
    "id": "ora.pLGAssetResource"
  },
  "actions": [
    "create",
    "update",
    "delete"
  ],
  "constraintType": "grant",
  "constraints": [
    {
      "constraintConfig": {
        "id": "ora.pLGConstraintConfiguration"
      },
      "values": [
        "HomeGoodsPriceGroup"
      ]
    }
  ]
}
```

The response is similar to the following:

```
{
    "constraintType": "grant",
    "roles": [],
    "name": "Security Criterion for HomeGoodsPriceGroup",
    "description": "Grants access to HomeGoodsPriceGroup",
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/
adminSecurityCriteria"
        }
    ],
```

```
    "securityCriteriaResource": {
        "name": "Price Groups",
        "id": "ora.pLGAssetResource"
    },
    "lastModified": "2021-03-08T20:12:19.302Z",
    "id": "securityCriterionHomeGoodsPriceGroup",
    "constraints": [
        {
            "values": [
                "HomeGoodsPriceGroup"
            ],
            "constraintConfig": {
                "id": "ora.pLGConstraintConfiguration"
            },
            "id": "scc-100002"
        }
    ],
    "actions": [
        "create",
        "update",
        "delete"
    ]
}
```

List available security criteria

You can use the `listAdminSecurityCriteria` endpoint to see a list of your available security criteria. The following example returns the two security criteria created in the examples above:

```
GET /ccadmin/v1/adminSecurityCriteriaResources  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

The response is similar to the following:

```
{
    "total": 2,
    "totalResults": 2,
    "offset": 0,
    "limit": 250,
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/
adminSecurityCriteriaResources"
        }
    ],
    "sort": [
        {
            "property": "id",
            "order": "asc"
        }
    ],
```

```
    "items": [
        {
            "constraintConfigurations": [
                {
                    "constrainingAssetType": "catalog",
                    "displayName": "Catalogs",
                    "constrainingAssetDisplayNameProperty":
"displayName",
                    "id": "ora.catalogConstraintConfiguration",
                    "constrainingAssetGroup": "/atg/commerce/catalog/
ProductCatalog/"
                }
            ],
            "name": "Catalog Assets",
            "id": "ora.catalogAssetResource"
        },
        {
            "constraintConfigurations": [
                {
                    "constrainingAssetType": "priceListGroup",
                    "displayName": "Price Groups",
                    "constrainingAssetDisplayNameProperty":
"displayName",
                    "id": "ora.pLGConstraintConfiguration",
                    "constrainingAssetGroup": "/atg/commerce/pricing/
priceLists/PriceLists/"
                }
            ],
            "name": "Price Groups",
            "id": "ora.pLGAssetResource"
        }
    ]
}
```


Create generic access rights

You can create generic access rights and use them to limit which internal users can access specific shopper data.

A generic access right is essentially a label that is applied to metadata attributes of shopper profile properties. Generic access rights are used by the Commerce access control system to determine which users can access the values of those properties. For example, if a property's `readAccessRight` attribute is set to the ID of a specific generic access right, then internal users who are assigned roles that include that access right will be able to see the value of that property. (Because they are simply labels, you may also be able to use generic access rights in other contexts such as integrations, though you will need to implement the logic for applying them in your own code.)

This section describes how to create and view generic access rights. The use of generic access rights to control access to properties is described in detail in Control access to shopper data.

Create a new generic access right

Commerce does not include any predefined generic access rights for internal users. If you want to use generic access rights, you need to create them using the `createAdminAccessRight` endpoint in the Admin API. For example:

```
POST /ccadmin/v1/adminAccessRights  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
    "displayName": "Access Right 1",
    "name": "ar1",
    "repositoryId": "ar1",
    "description": "First of several access rights."
}
```

The response is similar to this:

```
{
    "displayName": "Access Right 1",
    "name": "ar1",
    "repositoryId": "ar1",
    "description": "First of several access rights.",
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/
adminAccessRights"
        }
    ],
    "id": "ar1",
    "type": "generic",
    "category": {
        "displayName": "Custom",
        "id": "customAccessRightCategory"
    }
}
```

The Use and modify roles for internal users section of this chapter describes how to assign generic access rights to roles. For information about how to apply access rights to profile properties, see Control access to shopper data.

List generic access rights

To list all of the available generic access rights, use the `listAdminAccessRights` endpoint. For example:

```
GET /ccadmin/v1/adminAccessRights  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

You can return a list of the predefined privileges in addition to the generic access rights by including the `includePrivileges` query parameter in the call. For example:

```
GET /ccadmin/v1/adminAccessRights?includePrivileges=true  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json
```

The following shows part of the list of privileges in the response:

```
...

    "items": [
        {
            "displayName": "Account Manager",
            "name": "Account Manager",
            "repositoryId": "ora.accountManagerPrivilege",
            "description": "Account Manager Privilege",
            "id": "ora.accountManagerPrivilege",
            "type": "privilege"
        },
        {
            "displayName": "Administrator",
            "name": "Administrator",
            "repositoryId": "ora.adminPrivilege",
            "description": "Administrator Privilege",
            "id": "ora.adminPrivilege",
            "type": "privilege"
        },

...
```

Control access to shopper data

The Oracle Commerce access control system can be used to manage internal users' access to shopper data, such as profile properties.

Access to these properties is controlled using metadata attributes of the properties. You assign roles and generic access rights to these attributes to specify, for each individual property, which groups of internal users can access the property, and the type of access granted (either read, write, or both).

You can use property access control to support compliance with privacy laws such as the European Union General Data Protection Regulation (GDPR), by enforcing restrictions on who can access a shopper's personal data. For example, you might want to allow an administrator to see all of the properties in a shopper's profile, but allow customer service agents to see only a subset of the profile properties.

The following are the primary attributes used to control which users can read or write the value of a property:

- `readRole` – You can set this attribute to the ID of a role. Users with the specified role can see the property value.
- `writeRole` – You can set this attribute to the ID of a role. Users with the specified role can set or change the property value.

- `readAccessRight` – You can set this attribute to the ID of an access right. Users with the specified access right can see the property value.
- `writeAccessRight` – You can set this attribute to the ID of an access right. Users with the specified access right can set or change the property value.

By default, these attributes are null for any given property, which means that any user who is logged in can see and modify the property's value. When you set one or more of these attributes, you are actually revoking access from users who lack the specified role or access right. Note that each attribute can only be set to a single ID.

A user's ability to access properties is determined both by the roles the user has and by the access rights those roles have:

- A user has read access to a property if the user has the role that is specified by the property's `readRole` attribute, or if the user has a role that has the access right that is specified by the property's `readAccessRight` attribute.
- A user has write access to a property if the user has the role that is specified by a property's `writeRole` attribute, or if the user has a role that has the access right that is specified by the property's `writeAccessRight` attribute.

You should make sure that your settings for these attributes do not result in any users having write access to a property but not read access. Such a user would be able to modify the property's values but not be able to see the changes he or she makes.

To avoid this situation, be careful when you set these attributes to ensure that the users with write access to a given property are a subset of those with read access. Here are some examples:

- If you want all users to be able to see a property's values, but want only administrators to be able to modify the values, then you could set the `writeRole` attribute to `adminRole`, and leave `readRole` null.
- If you do not want any users to be able to edit the values of a property, you could set the `writeAccessRight` attribute to an access right that is not assigned to any roles (and therefore is not associated with any users). You can then use `readRole` or `readAccessRight` to assign read access as desired.
- If you want a certain group of users to have read and write access to a property, but for no other users to have either form of access, you could set `writeRole` and `readRole` to the same role.

Note: Subsystems of Commerce that exchange data with external systems (for example, webhooks and bulk export and import) are not affected by property access control settings. If you want to restrict property access in these subsystems, you may need to implement access control on the external systems.

In addition, the Freemarker email templates included with Commerce are not affected by property access control settings. If you want to restrict property access in emails your store sends, you must manually remove properties from the templates. (See Customize Email Templates for more information.)

Configure the data to return

Properties have two attributes, `readSecurityLevel` and `writeSecurityLevel`, that control how Commerce responds when a user lacking the necessary role or access right attempts to access the property.

The `readSecurityLevel` attribute can be set to one of the following values:

- `ignore` – Return a masking value rather than the actual value of the property. If `readSecurityLevel` is not set, it defaults to `ignore`.
- `deny` – Omit the property from the response entirely. This option is available only for custom properties.

The `writeSecurityLevel` attribute can be set to one of the following values:

- `ignore` – Attempts to modify the property value will fail silently. If `writeSecurityLevel` is not set, it defaults to `ignore`.
- `deny` – Attempts to modify the property value will result in errors. Note, however, if the user attempts to set the value to the same value that was originally returned (that is, either the current value of the property or a masking value, depending on the user's read access), no error will result. This is to handle cases where a form populated with current values attempts to write all of those values back when it is submitted. This option is available only for custom properties.

Note that the values of these attributes have an effect only if read or write access attributes are set on the property as well. For example, if the `readSecurityLevel` attribute is set to `deny` for a custom property, the property is omitted from a response only if read access is restricted by the `readRole` or `readAccessRight` attribute, and the user does not have the specified role or access right.

Return masking values

If the value of a property's `readSecurityLevel` attribute is `ignore`, then attempts to access the property by users lacking the necessary access right or role return a placeholder called a masking value. The logic for determining the value is as follows:

- If the property is not required, null is returned.
- If the property is required, and its default value is set, the default value is returned. (Note that for a custom property that is required, the default value must be set.)
- If the property is required, and its default value is not set, a generic value is returned. The value depends on the data type of the property:
  - String – the empty string
  - Numeric value -- 0
  - Date or timestamp -- Jan 1 1970
  - Enumeration -- the first enumerated value

You can override this logic by explicitly specifying a placeholder value for the property using the `securityMaskingValue` attribute. For example, you might want to set the placeholder value for strings to "`XXXXX`," to make it clear that the actual value is being suppressed rather than empty. Note that the `securityMaskingValue` you specify must match the data type of the property.

Set the values of access control attributes

To set the access control attributes on specific properties, you use the endpoints in the Admin API for modifying the item type for those properties. For example, to configure access control on profile properties, you use the `updateShopperType` endpoint to modify the `user` shopper type.

The following example illustrates setting the role and access right attributes of a property, as well as its `securityMaskingValue`:

```
PUT /ccadmin/v1/shopperTypes/user  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en
Content-Type: application/json

{
 "properties": {
     "lastName": {
         "readRole": "audit",
         "writeRole": "audit",
         "readAccessRight": "ar10",
         "writeAccessRight": "ar10",
         "shopperReadable": true,
         "shopperWriteable": true,
         "securityMaskingValue": "XXXXX"
    }
  }
}
```

The response shows the attribute values you set:

```
...
"lastName": {
    "shopperWriteable": true,
    "readRole": "audit",
    "readSecurityLevel": null,
    "readAccessRight": "ar10",
    "securityMaskingValue": "XXXXX",
    "length": 254,
    "shopperReadable": true,
    "label": "Last Name",
    "type": "shortText",
    "writeSecurityLevel": null,
    "writeAccessRight": "ar10",
    "required": false,
    "searchable": false,
    "writable": true,
    "internalOnly": false,
    "uiEditorType": "shortText",
    "default": null,
    "audienceVisibility": null,
    "localizable": false,
    "textSearchable": false,
    "writeRole": "audit",
    "dimension": false,
    "editableAttributes": [
        "shopperWriteable",
        "readRole",
        "readSecurityLevel",
        "readAccessRight",
        "securityMaskingValue",
```

```
        "shopperReadable",
        "label",
        "writeSecurityLevel",
        "writeAccessRight",
        "required",
        "searchable",
        "internalOnly",
        "default",
        "audienceVisibility",
        "textSearchable",
        "writeRole",
        "dimension",
        "multiSelect"
    ],
    "multiSelect": null
},
...
```

These settings restrict both read and write access to only users that either have the `audit` role or the `ar10` access right. If a user has neither of these and attempts to view a shopper's profile, the `lastName` property value is masked in the response. For example:

```
...
"lastName": "XXXXX",
"GDPRProfileP13nConsentDate": null,
"GDPRProfileP13nConsentGranted": false,
"gender": "female",
...
```

Note that you should not set access-control attributes on a property that points to another object or collection of objects. Instead, set the attributes on the individual properties of the objects.

For a property that holds an array of strings or numeric values, you can set access-control attributes to specify access rights and roles, but you cannot set the `securityMaskingValue` attribute.

Understand access control for applications

The access control system for internal users can also be used to manage access for registered applications such as integrations and server-side extensions.

When an application is registered in the administration interface, it is automatically assigned a role that provides it access to a wide range of Commerce endpoints on the administration server. If you want to restrict the application's access, you can remove this role from the application and assign a custom role to it that provides access to only specific endpoints.

Default role and privileges

A predefined role, Advanced Application plus Extensions (role ID = ora.advancedApplicationPlusExtensionsRole), is assigned to each registered

application by default. This role contains two privileges that provide access to a broad range of functionality in Commerce:

- The Advanced Application privilege (privilege ID = ora.advancedApplicationPrivilege), confers access to many (but not all) of the endpoints that an internal user with the Administrator privilege can access.
- The Extensions privilege (privilege ID = ora.extensionsPrivilege), confers access to certain endpoints for managing extensions.

These privileges are designed to be used only by registered applications and should not be assigned to internal users. For information about predefined roles and privileges intended for internal users, see Use and modify roles for internal users.

Manage extensions from applications

The access provided by the Extensions privilege and the Advanced Application privilege include the ability to call certain Admin API endpoints for managing extensions:

- The Extensions privilege provides access to the `/ccadmin/v1/applicationIds` endpoints, when `type` = `extension`. Note that these endpoints, when `type` = `application`, cannot be accessed by applications.
- The Advanced Application provides access to the `/ccadmin/v1/extensions` endpoints.

Endpoint privileges

Commerce includes a special type of privilege for use by applications, called an *endpoint privilege*, that you can include in custom roles to grant access to specific endpoints. You cannot create new endpoint privileges, but you can use one or more of the predefined endpoint privileges, which are described in the table below. Note that these endpoints omit any PCI (credit card) data when invoked from an application.

If an endpoint has an associated endpoint privilege, an application with that endpoint privilege has access to the endpoint, regardless of what other privileges the application has. If an application does not have the endpoint privilege, the application might still have access to the endpoint, depending on the functional privileges it has. For example, the Advanced Application privilege confers access to many endpoints.

Endpoint Privilege Name	Endpoint Privilege ID	Associated Endpoint
Admin Bulk Export Endpoint Privilege	endpoint.ccadmin.executeExport	`POST /ccadmin/v1/exportProcess`
Admin Get Export Status Endpoint Privilege	endpoint.ccadmin.getExportProcess	`GET /ccadmin/v1/exportProcess/{token}`
Admin Abort Export Endpoint Privilege	endpoint.ccadmin.abortExportProcess	`POST /ccadmin/v1/exportProcess/{token}/abort`
Admin Get Shopper Type Properties Endpoint Privilege	endpoint.ccadmin.getShopperType	`GET /ccadmin/v1/shopperTypes/{id}`
Admin Get Item Type Properties Endpoint Privilege	endpoint.ccadmin.getItemType	`GET /ccadmin/v1/itemTypes/{id}`
Admin List Promotions Endpoint Privilege	endpoint.ccadmin.listPromotions	`GET /ccadmin/v1/promotions`
Admin List Sites Endpoint Privilege	endpoint.ccadmin.getSites	`GET /ccadmin/v1/sites`

Endpoint Privilege Name	Endpoint Privilege ID	Associated Endpoint
Admin Get Site Endpoint Privilege	endpoint.ccadmin.getSite	`GET /ccadmin/v1/sites/{id}`

Create custom roles for applications

To confer more limited access to an application than the predefined Advanced Application plus Extensions role allows, you can use the `createAdminRole` endpoint in the Admin API to create custom roles. You then assign one or more of these roles to the application, and remove the Advanced Application plus Extensions role from the application.

A custom role for applications can contain functional privileges (such as the Advanced Application privilege), endpoint privileges, or both. (Roles can also have security criteria and generic access rights, but these apply only to internal users and have no effect on applications.) The following example creates a custom role called CustomAppRole1 and gives it the Extensions privilege and the Admin Bulk Export Endpoint Privilege:

```
POST /ccadmin/v1/adminRoles  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "repositoryId": "CustomAppRole1",

  "accessRights": [
    {
      "repositoryId": "ora.extensionsPrivilege"
    },
    {
      "repositoryId": "endpoint.ccadmin.executeExport"
    }
  ],
  "name": "CustomAppRole1"
}
```

The response looks similar to this:

```
{
    "securityCriteria": [],
    "name": "CustomAppRole1",
    "repositoryId": "CustomAppRole1",
    "description": null,
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/adminRoles"
        }
    ],
    "accessRights": [
        {
            "displayName": "Admin Bulk Export Endpoint Privilege",
```

```
            "name": "Admin Bulk Export Endpoint Privilege",
            "repositoryId": "endpoint.ccadmin.executeExport",
            "description": "POST /ccadmin/v1/exportProcess Endpoint
Privilege ",
            "id": "endpoint.ccadmin.executeExport",
            "type": "endpoint"
        },
        {
            "displayName": "Extensions",
            "name": "Extensions",
            "repositoryId": "ora.extensionsPrivilege",
            "description": "Confers the ability to update and delete
extensions.",
            "id": "ora.extensionsPrivilege",
            "type": "privilege"
        }
    ],
    "securityCriteriaLastModified": null,
    "id": "CustomAppRole1",
    "category": "Custom"
}
```

Assign roles to an application

To register a new application and assign roles to it, you use the `createApplicationID` endpoint in the Admin API. For example:

```
POST /ccadmin/v1/applicationIds  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "name": "MyApp",
  "type": "application",
  "roles": [
    "CustomAppRole1"
  ]
}
```

The response looks similar to this:

```
{
    "creationTime": "2021-11-08T16:30:53.000Z",
    "roles": [
        {
            "repositoryId": "CustomAppRole1"
        }
    ],
    "source": "custom",
    "type": "application",
    "token": "<application_key>",
    "external": false,
    "createdBy": {
        "firstName": "Amber",
```

```
        "lastName": "Admin",
        "repositoryId": "admin",
        "id": "admin"
    },
    "repositoryId": "71ca5c5d-3e08-46cd-9781-b531ac67ea13",
    "name": "MyApp",
    "inUse": false,
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/
applicationIds"
        }
    ],
    "id": "71ca5c5d-3e08-46cd-9781-b531ac67ea13",
    "createdById": null
}
```

Notice that the response includes two key identifiers for the application: the `token` property, which contains the application key, and the `id` property, which contains the internal ID for the application.

You can also register an application in the administration interface, as described in Register applications. If you register applications this way, or if you have existing applications you want to assign roles to, you can use the `updateApplicationID` endpoint in the Admin API to assign roles. The request and response payloads for this endpoint are similar to the payload for `createApplicationID`. For example, to assign a custom role named CustomAppRole2 to the MyApp application shown above:

```
PUT /ccadmin/v1/applicationIds/71ca5c5d-3e08-46cd-9781-b531ac67ea13
HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "name": "MyApp",
  "type": "application",
  "roles": [
    "CustomAppRole2"
  ]
}
```

When you use the `updateApplicationID` endpoint to add roles to an application, the roles you specify replace any roles currently assigned. If you want the application to retain the roles previously assigned to it, be sure to include them in the request payload. In the example above, if you want to add CustomAppRole2 without removing CustomAppRole1, you need to specify both roles in the request.

Note: When you register an application, if you do not explicitly specify roles, the application is automatically assigned several predefined internal user roles, plus the Advanced Application plus Extensions role. This is true for any applications you register in the administration interface, and for applications you register using the `createApplicationID` endpoint without specifying roles. It is also true for existing applications you registered previously without specifying roles.

The internal roles are included for backward compatibility reasons and are not functional for registered applications. If you subsequently assign roles to an application using the `updateApplicationID` endpoint, these roles are removed.

Limitations on custom access control

The access control system does not provide a way for code within your application to check what roles or access rights are assigned to the application. Therefore, your application cannot use roles or the access rights they contain to control access to its custom functionality. Access to functionality that Commerce provides, however, will be enforced based on the privileges that the application's roles contain.

34

Manage an Account-based Storefront

This chapter provides information on additional requirements you must satisfy when creating an account-based storefront. It also provides information on adding the delegated administration feature to your storefront.

Note: The account-based commerce feature may not be enabled in your environment.

Manage account-based shopper profiles

Account-based storefronts cannot use the Customer Profile widget that is included in Commerce out of the box.

This widget allows shoppers to edit their billing and shipping addresses. In account-based storefronts, addresses are managed at the account level and shoppers should not be able to modify them, which makes the Customer Profile widget inappropriate for account-based storefronts. Instead, account-based storefronts must create a new widget that displays profile information but does not enable address editing.

This section describes how to create this new widget.

Download the existing Customer Profile widget

You can create your new widget using the code from the existing Customer Profile widget as a starting point. For instructions on how to download the code for the existing Customer Profile widget, see Download widget source code.

Modify the Customer Profile widget

The `display.template` for the existing widget includes the following lines of code:

```
<div id="CC-customerProfile-profileDetails-section"
    class="row cc-customerProfile-profile-details"
    data-bind="template: { name: templateAbsoluteUrl(
              '/templates/customerProfileDetails.template') ,
              templateUrl: ''}">
```

This code references another template file, `customerProfileDetails.template`, that contains the code that is responsible for rendering the profile's addresses:

```
<div class="col-sm-6" id="CC-customerProfile-shipping">
  <div class="row cc-customerProfile-shipping-address"
       id="CC-customerProfile-shipping-details"
       data-bind="template: { name: templateAbsoluteUrl(
       '/templates/customerProfileShippingAddress.template') ,
       templateUrl: ''}"></div>
</div>
```

At a minimum, these lines of code in the `customerProfileDetails.template` must be removed or commented out to prohibit shoppers from editing or adding addresses. However, this will also prevent the addresses from being displayed on the Profile Layout (note that the shopper can still see her addresses on the Checkout Layout and Checkout Layout with GiftCard pages). As an alternative, you can modify the `customerProfileShippingAddress.template` referenced in this code to retrieve addresses from the account instead of the individual shopper.

If you choose to edit the `customerProfileShippingAddress.template` file, you must remove any options in that file that edit or add new addresses. To display the addresses, you must use the `organizationAddressBook` array from the `UserViewModel` in place of the `shippingAddressBook` array. For example, in the following code:

```
<div class="col-xs-12">
  <!-- ko with: user -->
  <fieldset id="CC-customerProfile-edit-fields">
    <legend class="cc-profile-legend-title"
            id="CC-customerProfile-shippingAddress-label">
      <span data-bind="widgetLocaleText:'shippingAddressText'"></span>
   …
   …
      <!-- View Begins -->
      <div class="col-sm-10" id="CC-customerProfile-shippingAddress-
view-region">
        <!-- ko foreach: shippingAddressBook -->
        <!-- ko if: postalCode -->
        <address class="CC-customerProfile-shipping-address
                        cc-customer-profile-shipping-address-view">
          <div class="pull-right">
            <!-- ko if: $parent.shippingAddressBook().length > 1 -->
              <button class="btn btn-default btn-sm"
                      data-bind="click:
                                 $parents[1].handleSelectDefaultShippin
gAddress,
                                 disable: isDefaultAddress(),
…
```

This line:

```
<!-- ko foreach: shippingAddressBook -->
```

Should change to:

```
<!-- ko foreach: organizationAddressBook -->
```

And this line:

```
<!-- ko if: $parent.shippingAddressBook().length > 1 -->
```

Should change to:

```
<!-- ko if: $parent.organizationAddressBook.length > 1 -->
```

After making the necessary code changes, you must create a new extension package to contain the widget and upload it to Commerce. To make the new extension package, you can copy the extension package from the original Customer Profile Widget, however, you must modify the widget directory name and the meta-data in the `ext.json` and `widget.json` files for your new widget. For more details, see Understand extensions.

Implement access control in business accounts

For stores with business accounts, there are two types of users who can have their access to properties controlled with roles and access rights:

- Internal users who can work with the administration interface's Accounts page because they were assigned the Administrator or Account Manager role.
- Storefront users who have been assigned the Administrator role (delegated administrators) or the Approver role. These roles allow them to manage certain aspects of their business accounts. Delegated administrators do not have access to the Commerce administration UI. Delegated administrators perform all account-management tasks on their My Account pages, which are available once they log into your store.

With the support of role-based access control, you can also use these roles and access rights to enforce restrictions on which internal users can access a shopper's personal data. More specifically, you can control which properties an internal user can see or edit. For example, if you set the `writeRole` attribute of a property to Administrator, only someone with the Administrator role can set the value of a specific property.

Delegated administrators can also have their access to properties controlled with roles and access rights. For more information on delegated administrators, refer to Understand delegated administration. For specific details on how a delegated administrator can have their access to properties controlled with roles and access rights, refer to the Create access rights for use with shopper roles section of this document.

Understand how access rights and roles affect the Accounts page

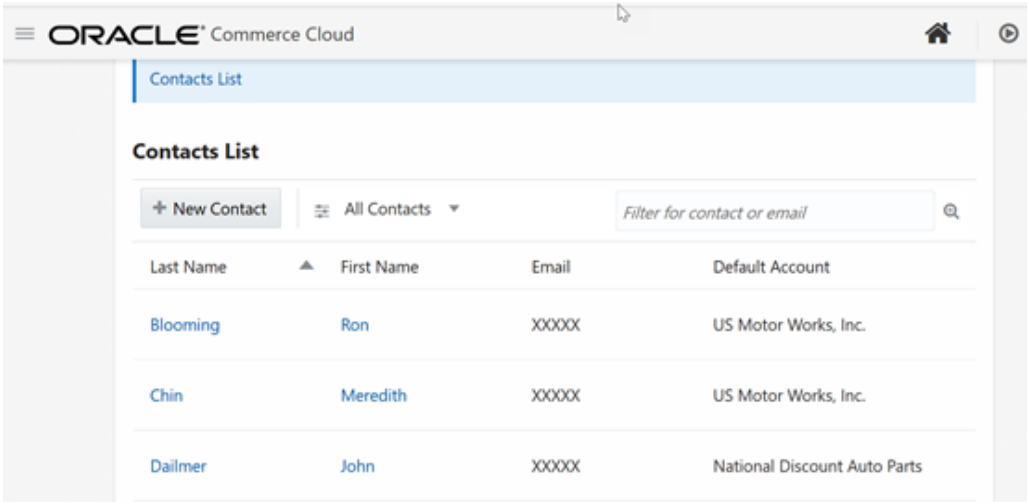
Internal users with a role of Administrator or Account Manager have access to properties found on the Accounts page in the Oracle Commerce Cloud administration UI. With support of role-based access control, you can limit what a user is able to see or modify on the Accounts page based on role and access rights. This can be particularly useful when trying to limit access to information in the following areas:

- Contacts list
- Contact details
- Account details - This includes the following tabs: General, Addresses, Contacts, Contracts, Approvals, Shipping/Payments, and Registration Requests (if the feature has been enabled). The use of role-based access control could be particularly useful on tabs that display personal information such as the Addresses and Contacts tabs.

By focusing on tabs that include sensitive and personal information, you can take advantage of this feature to effectively control what a user is able to see or modify on the Accounts page.

The following image shows an example of what an internal user might see on the Accounts page of the administration interface if role-based access control is implemented for an

account-based store. Specifically, an access right has been assigned to both the `readAccessRight` and `writeAccessRight` attributes of the `user` shopper type's `email` property. In this case the user is not assigned a role that includes the same access right – as a result the email address of each contact appears as the placeholder value `XXXXX`. See the Configure the data to return section of this document for more information.



Other things you should pay attention to when you are considering using role-based access control to control access to properties on the Accounts page include the following:

- If you make account name, contact name, or contact email a restricted property, you should specify a non-null masking value. Otherwise, business users who view the list of contacts or accounts, but don't have access to these fields, may see null values, and therefore may not be able to open the details of a contact or account.
- If a user tries to view a list, and the default sort is on one or more fields to which they do not have read access, the list returns unsorted results. To the requesting user, all instances of the field (in all rows) will have the same masked value. You can sort by clicking the headings in the tables that appear on the following Accounts pages: Contacts List, Accounts List, and Registration Requests List as well as on the All Sites and Contacts tabs for an account's details page, which you see when you click on that specific account in the Accounts List.
- If a user tries to search on a field to which they do not have read access, no results will be returned. This refers to any search boxes (the ones that have Filter as the label in them) on the Accounts pages (for example, accounts, contacts, registration requests, and sites).
- If a user tries to do an Advanced search and any of the fields they search on is one to which they do not have access, no results will be returned. Advanced search is available on the Contacts List, Accounts List, and the Registration Requests List pages.

Plan access control for an account-based store

This section describes things to keep in mind as you implement property access features for an account-based store.

- Each of a property's access-control attributes (`readRole`, `writeRole`, `readAccessRight`, and `writeAccessRight`) can be set only to a single ID. If you plan to use access rights to control property access for both internal users and

delegated administrators, make sure that corresponding access rights and custom roles you create for internal users and shoppers have the same `repositoryId` value.
For example, suppose you use the `createAdminAccessRight` endpoint to create an access right whose ID is `ar10`. You then assign the access right to a property's `readAccessRight` attribute. Any internal user whose role includes the access right `ar10` will have read access to that property. If you want delegated administrators to also have read access to that property, you must use the `createAccessRight` endpoint to create an access right with the exact same ID, `ar10`. Then, any delegated administrator whose role includes the access right `ar10` will also have read access to the property.

- If you set access-control attributes on a property, think about whether you want to set the property's `shopperReadable` and `shopperWriteable` attributes to `true`. If these attributes are `false`, shoppers (including delegated administrators) who do not have the appropriate access right assigned to them via a role, will not be able to access the properties for their own accounts and orders. For more information about these attributes, see Understand roles and access rights.
- Depending on how you implement property restrictions, you may want to customize the storefront widgets you have configured for layouts that your delegated administrators access. (For example, the Order Approval Settings, Account Contacts, and Account Addresses widgets.) You could add logic that provides better user experience for delegated administrators. For example, if a delegated administrator does not have write access to a property, the widget could include logic that prevents them from saving the property, rather than waiting to get an error from the server.
- Commerce provides one set of endpoints for creating access rights and roles for shoppers and another for creating them for internal users. However, there is only one set of endpoints for assigning access rights to properties. If you plan to control property access for both internal users and delegated administrators, make sure that corresponding access rights you create with `createAdminAccessRight` (for internal users) and `createAccessRight` (for delegated administrators) have the same `repositoryId` value. This helps ensure consistency when you implement property access control for both types of users.
  For more information about assigning access rights to properties, see Set access control on properties.
- If your account-based store is configured to allow shoppers to submit account registration requests, do not restrict a shopper's write access to any properties of the `organizationRequest` item type, as this will cause any registration request the shopper submits to fail.

Create access rights for use with shopper roles

You create access rights to control delegated administrators' access to properties using the `createAccessRight` endpoint in the Admin API.

The following example creates an access right that can be used with shopper roles:

```
POST /ccadmin/v1/accessRights  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
    "displayName": "Shopper Email Access Right 1",
    "name": "shopperEmailAr1",
    "repositoryId": "shopperEmailAr1",
```

```
    "description": "First of several storefront access rights."
}
```

The response is similar to this:

```
{
  "displayName": "Shopper Email Access Right 1",
  "name": "shopperEmailAr1",
  "repositoryId": "shopperEmailAr1",
  "description": "First of several storefront access rights.",
  "links": [
    {
      "rel": "self",
      "href": "http://www.example.com:7002/ccadmin/v1/accessRights"
    }
  ]
}
```

Create custom shopper roles

Commerce provides the ability to create custom shopper roles that will contain shopper access rights. Custom roles are global roles, which can be assigned to delegated administrators in any account. You should not use the built-in Storefront Roles (Buyer, Account Address Manager, Administrator, Approver, and Profile Address Manager), for property access control. While the REST API does not prevent you from assigning access rights to the built-in Storefront Roles, these roles are account-specific and using them to control access to individual properties may not produce the results you expected when you planned your access-control strategy.

Note: Custom shopper roles do not appear in the administration interface, so you must assign them to shopper profiles with the `createProfile`, `updateProfile`, or `updateUserRoles` endpoints in the Admin API.

To create a custom shopper role, use the `createRole` endpoint in the Admin API. The following example creates a custom role and assigns an existing access right to it in the same request. You can also create a custom role with no access rights and assign them to the role later with the `updateRole` endpoint, or just use the role (without access rights) to control property access.

```
POST /ccadmin/v1/roles  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "name": "No Email Access",
  "repositoryId": "noEmail",
  "description": "Delegated admin who cannot access shopper email
addresses.",
  "accessRights": [
    {
      "repositoryId": "shopperEmailAr1"
    }
  ]
}
```

The response is similar to this:

```
{
    "name": "No Email Access",
    "repositoryId": "noEmail",
    "description": "Delegated admin who cannot access shopper email
addresses.",
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/roles"
        }
    ],
    "accessRights": [
        {
            "repositoryId": "shopperEmailAr1",
        }
    ],
    "category": "Custom"
}
```

The following example assigns two existing access rights to an existing custom role:

```
PUT /ccadmin/v1/roles/noPhone  HTTP/1.1
Authorization: Bearer <access_token>
Content-Type: application/json

{
  "accessRights": [
    {
      "repositoryId": "bbar1",
      "repositoryId": "bbar2"
    }
  ]
}
```

The response is similar to this:

```
{
    "name": "No Phone Number Access",
    "repositoryId": "noPhone",
    "description": "Delegated admin who cannot access shopper phone
numbers.",
    "links": [
        {
            "rel": "self",
            "href": "http://www.example.com:7002/ccadmin/v1/roles"
        }
    ],
    "accessRights": [
        {
            "repositoryId": "bbar1",
            "repositoryId": "bbar2",
        }
```

```
    ],
    "category": "Custom"
}
```

Create custom properties for accounts

This section describes how to use the Commerce REST web services APIs to add custom properties to accounts.

See Use the REST APIs for information you need to know before using the services.

Note that you can also create custom properties for the contacts associated with accounts. Contacts are stored as shopper profiles, so any custom properties you add to shopper profiles are available for contacts as well. See Manage Shopper Profiles for information.

View an account

To view an existing account, first log into the Admin API on the administration server using a profile that has the Administrator role. For example:

```
POST /ccadmin/v1/mfalogin HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=admin1@example.com&password=A3ddj3w2&totp_
code=365214
```

Then issue a request to the `getOrganization` endpoint, providing the ID of the account you want to view, and including the access token that was returned by `/ccadmin/v1/mfalogin`. For example:

```
GET /ccadmin/v1/organizations/100002 HTTP/1.1
Authorization: Bearer <access_token>
```

The response shows the predefined account properties that are exposed by Commerce, and the values of the properties in the specified account. Note that the `members` property is an array in which the elements are the IDs of the contacts for the account:

```
{
    "customerType": "Supplier",
    "contract": {
        "creationDate": "2016-10-17T20:25:44.000Z",
        "startDate": null,
        "externalContractReference": "",
        "description": null,
        "catalog": {
            "repositoryId": "cloudCatalog"
        },
        "terms": null,
        "priceListGroup": {
            "repositoryId": "defaultPriceGroup"
        },
```

```
            "endDate": null,
            "displayName": "Sherman",
            "repositoryId": "100002"
        },
        "vatReferenceNumber": null,
        "links": [
            {
                "rel": "self",
                "href": "http://myserver.example.com:7002/ccadmin/v1/
organizations/100002"
            }
        ],
        "organizationLogoURL": null,
        "type": "company",
        "repositoryId": "100002",
        "dunsNumber": 000000000,
        "uniqueId": null,
        "id": "100002",
        "description": "Supplier of sprockets to a wide range of industries.",
        "name": "Sherman Sprockets",
        "active": true,
        "secondaryAddresses": [
            {
                "address": {
                    "postalCode": "02116",
                    "phoneNumber": "1-555-555-1212",
                    "state": "MA",
                    "address1": "1 Wixom Street",
                    "address2": null,
                    "companyName": "Sherman Sprockets",
                    "repositoryId": "120002",
                    "country": "US",
                    "city": "Boston"
                },
                "addressType": "Sherman"
            }
        ],
        "billingAddress": null,
        "taxReferenceNumber": null,
        "shippingAddress": {
            "postalCode": "02116",
            "phoneNumber": "1-555-555-1212",
            "state": "MA",
            "address1": "1 Wixom Street",
            "address2": null,
            "companyName": "Sherman Sprockets",
            "repositoryId": "120002",
            "country": "US",
            "city": "Boston"
        },
        "members": [
            {
                "repositoryId": "110001"
            }
            {
```

```
            "repositoryId": "110002"
        }
    ],
    "organizationLogo": null
}
```

You can modify the values of the properties of an account using the `PUT /ccadmin/v1/organizations/{id}` endpoint on the administration server.

View the `organization` item type

Accounts include a predefined set of properties for storing information, such as the DUNS number and the account name. The set of properties available for an account is determined by the `organization` item type, which serves as a template for accounts. You can view this item type with the following call:

```
GET /ccadmin/v1/itemTypes/organization HTTP/1.1
Authorization: Bearer <access_token>
```

The following example shows a portion of the response corresponding to one of the predefined account properties. Each property has a group of attributes whose values control the behavior associated with the property:

```
{
    "id": "organization",
    "links": [
        {
            "rel": "self",
            "href": "http://myserver.example.com:7002/ccadmin/v1/
itemTypes/organization"
        }
    ],
    "displayName": "Organization",
    "specifications": [
        {
         ...
            "writable": true,
            "localizable": false,
            "label": "Description",
            "type": "shortText",
            "id": "description",
            "uiEditorType": "shortText",
            "textSearchable": false,
            "multiSelect": null,
            "dimension": false,
            "internalOnly": false,
            "default": null,
            "editableAttributes": [
                "textSearchable",
                "multiSelect",
                "dimension",
                "internalOnly",
                "default",
                "label",
```

```
                "required",
                "searchable"
            ],
            "length": 254,
            "required": false,
            "searchable": false
        },
        ...
}
```

To modify the `organization` item type, you can create custom properties or modify existing properties by setting the values of these attributes. See Settable attributes of shopper type properties for descriptions of these attributes.

Add custom properties to the `organization` item type

You can use the `updateItemType` endpoint in the Commerce Admin API to add custom properties to the `organization` item type. When you add a custom property to the `organization` item type, the property is added to all accounts, including any new accounts created afterward and any accounts that already exist.

The ID of a custom property must include the underscore character (_). This ensures that the ID will not conflict with any properties that Commerce adds to accounts in the future. The endpoint produces an error if you attempt to create a custom property without an underscore in its ID.

The following example illustrates using the `updateItemType` endpoint to add a custom property. Note that the request header must specify the `x-ccasset-language` value:

```
PUT /ccadmin/v1/itemTypes/organization HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "id": "organization",
  "specifications": [
    {
          "id": "customer_tier",
          "label": "Customer tier",
          "type": "shortText",
          "uiEditorType": "shortText",
          "internalOnly": false,
          "required": false,
          "audienceVisibility": "b2b",
    }
  ]
}
```

The response includes the custom property you added:

```
...
{
     "writable": true,
     "localizable": false,
```

```
        "label": "Customer tier",
        "type": "shortText",
        "id": "customer_tier",
        "uiEditorType": "shortText",
        "textSearchable": false,
        "multiSelect": null,
        "dimension": false,
        "internalOnly": false,
        "default": null,
        "editableAttributes": [
            "textSearchable",
            "multiSelect",
            "dimension",
            "internalOnly",
            "default",
            "label",
            "required",
            "searchable"
            "audienceVisibility"
        ],
        "length": 254,
        "required": false,
        "searchable": false
    },
...
```

The `audienceVisibility` is the string that determines whether the property appears as a choice in the Attributes field of the audience interface. For account properties, this value should be set to `b2b`. See Define Audiences.

You can create a new account and set the values of custom properties (as well as the predefined properties) using the `createOrganization` endpoint. To set a custom property on an existing account, use the `updateOrganization` endpoint. For example:

```
PUT /ccadmin/v1/organizations/100001 HTTP/1.1
Authorization: Bearer <access_token>

{
    "customer_tier": "silver"
}
```

Add a custom property that lets administrators log internal notes

This section describes a code sample that adds a custom property to the organization type. This property displays a rich-text field on each account's General tab in the Commerce administration interface. Account administrators can use this editor to log internal notes about the account. In this example, the rich-text field is used to add and track internal notes related to registration requests.

Note: Keep in mind that you cannot search on this custom property in the administration interface. You can search only on custom short text properties. See Work with accounts for information about searching on custom properties.

The following sample request creates the rich-text editor, which is added to all accounts, including any new accounts created afterward and any accounts that already exist.

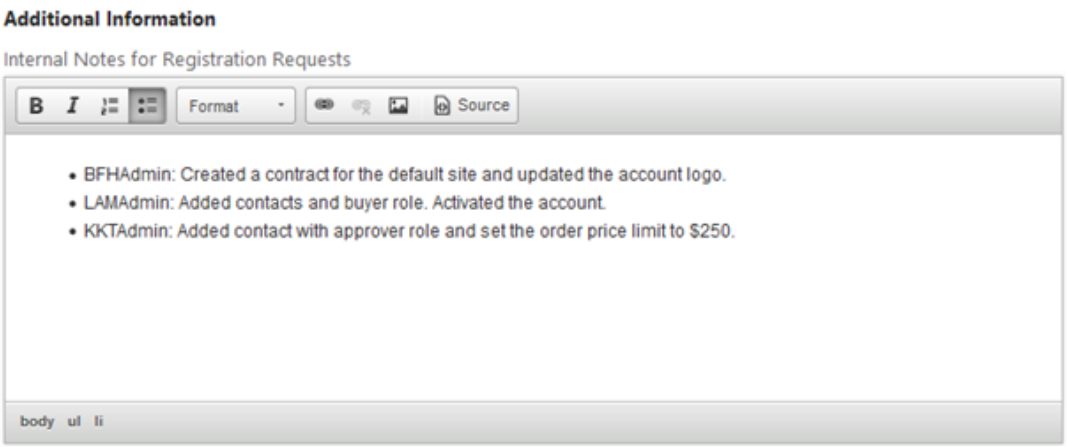
```
PUT /ccadmin/v1/itemTypes/organization HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
  "specifications": [
    {
     "label": "Internal Notes for Registration Requests",
     "id": "internal_notes",
     "default": null,
     "required": false,
     "localizable": false,
     "internalOnly": false,
     "textSearchable": false,
     "searchable": false,
     "multiSelect": false,
     "type": "richText",
     "uiEditorType": "richText"
    }
  ]
}
```

The editor appears at the bottom of the General tab for each account, in the Additional Information section. As with all property editors, the editor appears in each account, but the values are unique to the account where they are entered.

Keep the following tips in mind when you use the editor to create internal notes:

- Format each individual note, or task, as a separate bullet point.
- Each internal user should add their name or username to each note they create.
- Append new notes to the existing list. It is best not to modify or delete any existing notes because there is no way to restore a note that has been changed or deleted.

The following illustration shows an account's rich-text editor, with notes from three different account administrators.



Render custom properties of accounts on the storefront

To render custom properties of accounts on your storefront, customize the JavaScript in the Account Details widget to access the `dynamicProperties` observable array in the `Organization` view model. For example:

```
define(
    //-------------------------------------------------------------------
--
    // DEPENDENCIES
    //-------------------------------------------------------------------
--
    ['jquery', 'knockout', 'ccLogger', 'ccRestClient', 'ccConstants',
    'viewModels/dynamicPropertyMetaContainer'],
    //-------------------------------------------------------------------
--
    // Module definition
    //-------------------------------------------------------------------
--
    function($, ko, CCLogger, ccRestClient, CCConstants,
      DynamicPropertyMetaContainer) {
      'use strict';
      return {
        dynamicProperties: ko.observableArray(),
        onLoad : function(widget) {
        },

        beforeAppear: function(page) {
         var widget = this;
         if (widget.user() && widget.user().organizations &&
            widget.user().organizations().length > 0) {
              this.getDynamicPropertyMetadata(widget);
            }
         },

        getDynamicPropertyMetadata: function(widget) {
           var dynamicPropertyMetaInfo =
              DynamicPropertyMetaContainer.getInstance();
           if (dynamicPropertyMetaInfo &&
              dynamicPropertyMetaInfo.dynamicPropertyMetaCache &&
              dynamicPropertyMetaInfo.dynamicPropertyMetaCache.
              hasOwnProperty("organization")) {
                this.dynamicProperties(dynamicPropertyMetaInfo.
                dynamicPropertyMetaCache["organization"]);
            }
          },

         updateDynamicProperties: function() {
           var data = ko.toJS(this.user().organizations()[0]);
              ccRestClient.request("updateOrganization",data,
              this.updateDynamicPropertySuccess.bind(this),
              this.updateDynamicPropertyFailure.bind(this),
              this.user().organizations()[0].id());
           },
```

```
            updateDynamicPropertySuccess: function() {
            },
            updateDynamicPropertyFailure: function() {
            }
        }
    }
);
```

Modify the widget's template file to include Knockout bindings similar to the following:

```
<div id="CC-org-dynamic-property">
  <!-- ko foreach: dynamicProperties -->
    <label data-bind="attr: {id: 'CC-label-org-dynamicProperty-'+id()},
     text: label">
    </label>
    <input data-bind="attr: {id: 'CC-edit-org-dynamicProperty-'+id(),
     type: uiEditorType, required: required},
     value: $parent.user().organizations()[0][id()]">
    <br>
  <!--/ko-->
</div>
```

Add delegated administration to your storefront

A delegated administrator is a contact whose storefront role has been set to administrator.

Using delegated administration features in the storefront itself, a delegated administrator can perform tasks such as adding new contacts to an account, specifying roles (such as buyer or administrator) for contacts in an account, and specifying account-level billing and shipping addresses.

This section provides information about modifying your storefront to provide delegated administration features to contacts with administrator privileges.

Note: For more information on assigning delegated administration privileges to contacts, see Understand delegated administration.

Add delegated administration widgets to the Profile Layout

To add the delegated administration feature to your storefront pages, you must modify the Profile Layout. Specifically, the Profile Layout has access to two widgets that provide delegated administration features:

- The Account Contacts widget provides the delegated administrator with an interface for viewing, adding, removing, and modifying account contacts. Using this widget, the delegated administrator can also activate a contact as well as give administrator privileges to a contact. Note that a delegated administrator cannot remove the administrator role from her own profile; however, she can remove the administrator role from another contact. Similarly, a delegated administrator cannot deactivate herself but she can deactivate other delegated administrators.
- The Account Addresses widget allows a delegated administrator to specify account-level addresses. This widget also allows a delegated administrator to specify a default billing address and a default shipping address for the account.

To add these widgets to the Profile Layout, you must create a version of the Profile Layout for account-based shoppers only. To do this, go to the Design page, clone the Profile Layout, give it a descriptive name, enable the 'Display layout to account shoppers only' option, and save the clone.

To add the delegated administration widgets to the clone you created, you can create a vertical tab stack and place the widgets on individual tabs within the stack. To restrict the display of the delegated administration tabs to contacts that have administrator privileges, you can add something similar to the following code snippet in the vertical tab stack's template (this snippet assumes you used "My Profile", "Account Addresses", and "Account Contacts" as the display names for the tabs that hold the Customer Profile, Account Contacts, and Account Addresses widgets, respectively):

```
($data.displayName() == 'My Profile') ||
((($data.displayName() == 'Account Contacts') ||
($data.displayName() == 'Account Addresses')) &&
($masterViewModel.data.global.user.roles[0].function === 'admin'))
```

This code snippet shows the My Profile tab to all contacts but restricts the display of the Account Contacts and Account Addresses tabs to contacts with administrator privileges.

For more information on vertical tab stacks, see Customize your store layouts and Use Stacks for Increased Widget Layout Control.

Render custom properties of contacts on the storefront

Note: The information in this section applies only to rendering custom properties of contacts when accessed on the storefront by delegated administrators. For more general information about rendering custom properties of shopper profiles, see Access custom properties using the UserViewModel.

To render custom properties of contacts on your storefront, customize the JavaScript in the Account Contacts widget to access the `dynamicProperties` observable array in the `delegatedAdminContacts` view model. For example:

```
define(
    //-------------------------------------------------------------------
--
    // DEPENDENCIES
    //-------------------------------------------------------------------
--
    ['jquery', 'knockout', 'ccLogger', 'ccRestClient', 'ccConstants',
    'viewModels/dynamicPropertyMetaContainer'],
    //-------------------------------------------------------------------
--
    // Module definition
    //-------------------------------------------------------------------
--
    function($, ko, CCLogger, ccRestClient, CCConstants,
      DynamicPropertyMetaContainer) {
      'use strict';
      return {
        dynamicProperties: ko.observableArray(),
        onLoad : function(widget) {
```

```
          var self = this;
          widget.listingViewModel= ko.observable();
          widget.listingViewModel(new DelegatedAdminContacts());
        },
        getDynamicPropertyMetadata: function(widget) {
         var dynamicPropertyMetaInfo =
DynamicPropertyMetaContainer.getInstance();
          if (dynamicPropertyMetaInfo &&
          dynamicPropertyMetaInfo.dynamicPropertyMetaCache &&
          dynamicPropertyMetaInfo.dynamicPropertyMetaCache.
          hasOwnProperty("user")) {
            this.dynamicProperties(dynamicPropertyMetaInfo.
          dynamicPropertyMetaCache["user"]);
          }
        },
        beforeAppear: function(page) {
         var widget = this;
         if (widget.listingViewModel&&
          widget.listingViewModel().dynamicPropertyMetaInfo &&
          widget.listingViewModel().dynamicPropertyMetaInfo.
          dynamicPropertyMetaCache && widget.listingViewModel().
          dynamicPropertyMetaInfo.dynamicPropertyMetaCache.
          hasOwnProperty(CCConstants.ENDPOINT_SHOPPER_TYPE_PARAM)) {
            this.getDynamicPropertyMetadata();
         }
        },

       koToJS: function() {
         var widget = this;
         var data = {};

         for(var i =0; i < widget.dynamicProperties().length; i++) {
          data[widget.dynamicProperties()[i].id()] =
          widget.listingViewModel()[this.dynamicProperties()[i].id()]();
         }
         widget.isDelegatedAdminFormEdited = true;
         return data;
       },
    }
  }
);
```

Modify the widget's template file to include Knockout bindings similar to the following:

```
<div id="CC-contact-dynamic-property">
  <!-- ko foreach: dynamicProperties -->
    <label data-bind="attr: {id: 'CC-label-contact-dynamicProperty-'+id()},
      text: label">
    </label>
    <input data-bind="attr: {id: 'CC-edit-contact-dynamicProperty-'+id(),
      type: uiEditorType, required: required},
      value: $parent.listingViewModel()[id()]">
    <br>
```

```
  <!--/ko-->
</div>
```

Notify a contact of delegated administration changes

When a delegated administrator adds or removes a contact from an account, an Account Assignment Changed notification email is sent to the contact. When a delegated administrator adds or removes a role from a contact, a Role Assignment Changed notification email is sent to the contact. For more information on these email templates and how you can modify them, see Configure Email Settings and Customize Email Templates.

Ensure PayPal shoppers provide first and last name

Account-based storefronts have an additional requirement for shoppers who use PayPal as their payment method.

Specifically, an account-based storefront must ensure that the shopper provides a first and last name before clicking the PayPal button on the Checkout page.

To understand this requirement, it is helpful to have some context. When the PayPal button is clicked, a request is made to create an order. To pass validation, this request can either:

- Include a complete and valid address for the shopper, or
- Contain no address information for the shopper (the expectation in this case is that the shopper will provide complete address information after going through the PayPal process and returning to the Commerce storefront to complete the order).

Either case is acceptable and will pass validation. The issue for an account-based storefront is that every contact has a partial address, in the form of the address provided for the contact's parent account, that does not include a first and last name. Using this partial address when attempting to create the order causes a validation error.

The best method for ensuring that a shopper provides a first and last name before clicking the PayPal button depends on your storefront's requirements. Note that the first and last name requirement exists only for the Checkout page. The Cart page, which also has a PayPal button, does not have the same restriction because it never passes an address when it makes the create order request (because the shopper has no way of entering address information on the Cart page).

35

Integrate With a Procurement System

The Commerce punchout features let you integrate an account-based store with a procurement system.

This integration allows a shopper who is logged into a procurement system to access your storefront, view items from the assigned catalog, add items to the cart, and return to the procurement system, which approves the purchase.

Understand punchout

The Commerce punchout features let you integrate an account-based store with a procurement system.

This integration allows a shopper who is logged into a procurement system to access a Commerce storefront, view items from the assigned catalog, add items to the cart, and return to the procurement system.

Once the procurement system approves the purchase, it sends Commerce a purchase order, which is used to create the Commerce order.

Commerce punchout functionality is provided through server-side extensions (SSEs). Commerce identifies incoming punchout orders that come through the SSE (as cXML-based orders) by setting the `originOfOrder` property to `punchout` for incomplete orders and `purchaseOrder` once the order has been approved by the procurement system.

The following are some key terms that you need to be familiar with before you implement a punchout integration:

- The procurement system is an external system where the buyer starts the shopping process, and where the buyer's proposed purchases are approved.
- The supplier/seller is the Commerce merchant.
- A purchase order is a message sent from the procurement system to Commerce. The purchase order includes details about the cart the punchout shopper created. Commerce requires the purchase order so it can create an order.
- After a shopper has submitted cart data to the procurement system, they may need to update the order. This is called a re-punchout. The only difference between the punchout and re-punchout process is that after the process has returned the security token, re-punchout creates or updates incomplete orders and returns the order information in JSON format. The procurement system passes the cart information in cXML format for re-punchout; no cart information is passed during punchout.

During re-punchout, if the cart contains an invalid product or SKU, Commerce automatically removes it. If the cart contains a product or SKU that is out of stock, it remains in the cart and storefront displays a message that notifies the shopper of its inventory status.

Punchout functionality can be implemented only for a Commerce store that is configured for account-based commerce. (See Configure Business Accounts for more information.)

Punchout is implemented using cXML (commerce eXtensible Markup Language) 1.2, which is an open-standard XML-based data-exchange format. Before configuring punchout, you should be familiar with cXML. For information about cXML 1.2, see `http://cxml.org/`.

Understand Commerce punchout limitations

Commerce does not support level 2 punchout, which implements CIF (Content Interchange Format) catalogs to allow shoppers to view your Commerce catalog on the procurement system's site before punching out to your Commerce site to see real-time data, like prices, and add items to the cart.

Commerce does not support punchout marketplaces or networks, where buyers can connect to a number of different suppliers. Commerce supports only direct punchout.

Punchout can be used with most Commerce features that can be configured for account-based stores. However, punchout cannot be used with scheduled orders or quoting.

Configure CORS support

To access Commerce endpoints from the procurement system, you must configure CORS (cross-origin resource sharing) support in Commerce by explicitly specifying the procurement system's domain as one that is permitted to make requests to the punchout site. See CORS support for more information about configuring CORS support. See Work with the punchout server-side extension for details about the information that must be included in the `punchoutSetupResponse` Commerce sends to the procurement system (in response to the `punchoutSetupRequest` request).

Enable punchout for an account

This section describes how to set up and enable a new punchout account for Commerce.

It contains the following sections:

- Understand the punchout account setup and enablement tasks
- Create the punchout account
- Enable punchout on the punchout account
- Generate an authorization code
- Set the punchout cart time setting
- Set the frequency of incomplete punchout order clean up
- Learn about the generic punchout shopper profile
- Learn more about using punchout

Understand the punchout account setup and enablement tasks

To best understand the account setup and punchout enablement processes for the punchout feature, it's good to know at a high level the tasks that need to be completed. These tasks are:

1. Create the punchout account (if this has not been done already).
2. Enable punchout on the punchout account.

3. Generate an authorization code. This code will be shared with the procurement system in completing the account connection details to Oracle Commerce.
4. Download and customize the punchout server-side extensions, then upload them to your Node.js server. See Work with the punchout server-side extension for more information.
5. Set the punchout cart time setting.

The next sections provide details on how to complete these tasks.

Create the punchout account

Whenever you want to set up a new business buyer for a punchout account, you first need to go through the usual steps required in creating the business account that will use punchout (if you have not done so already).

This process includes designating contract info, providing addresses, choosing catalogs, selecting price groups, etc. Refer to Configure Business Accounts for more complete information on this task.

Note: For this release, no contacts can be set up for punchout accounts as Oracle Commerce does not persist the profile of punchout shoppers. To learn more about how punchout handles the profile of punchout shoppers, refer to Learn about the generic punchout shopper profile.

Additionally, merchants should define a relatively small set of business contacts in Oracle Commerce under whose profiles the procurement system will submit orders after approving them. (For example, these could be the profiles of the approvers at the procurement system.)

You have the following choices to help you define the small set of contacts:

- Import the profiles of users whom they expect to submit orders
- Customize a server side extension to create profiles on the fly when Oracle Commerce receives a purchase order

Profiles created in either of these ways will be regular contacts, just like the ones created by administrators. They can access the account/site directly just like any other contact on the account.

Enable punchout on the punchout account

After you have set up your punchout account, the next step is to enable punchout on that account. Since not every account will use punchout, this is the way to distinguish punchout accounts from non-punchout accounts.

Use the following steps to enable punchout for an account by using the administration interface:

1. In the administration interface, click the **Accounts** icon to go to the **Accounts** page.
2. Select the account that you want to use the punchout feature.
3. From the **Account** page of that account, select the **Punchout** tab.
4. Select **Allow Punchout Shopping** to enable the punchout feature for that account. This property:
    - Allows the designated account to accept punchout (cXML only) shopping requests if the account has an integration to a procurement system.
    - Is inheritable. All sub-accounts inherit this field by default.
    - Is not site-specific.

- Is unchecked by default.

5. Select **Enable Authorization Code** to enable the account to have the ability to generate a punchout authorization code. A **Generate** button appears. This property:

   - Allows you to generate the code that an external system provides in each cXML punchout or purchase order request for authentication purposes. This is the authorization code that will be shared with the procurement system that is using punchout.

   The account's account ID and URL are also shared with the procurement system.

   - Is inheritable. All sub-accounts inherit this field by default.
   - Is not site-specific.
   - Is unchecked by default.

Generate an authorization code

Punchout transactions are signed so that the system receiving the event can verify their authenticity. Once you select **Enable Authorization Code** on the **Punchout** tab, the Generate button appears below this option on the administration user interface. This button gives you the ability to generate a punchout authorization code for that account. This is the code that an external system provides in each cXML punchout or purchase order request for authentication purposes. This code can be regenerated at any time if need be.

The authorization code is shared with the procurement system implementing punchout. You must manually communicate this code (for example, via phone or email) to someone on the procurement side during implementation of the procurement system integration. After this is done, the procurement system sends it to Oracle Commerce with each request.

To generate the punchout authorization code, do the following

1. In the administration interface, click the **Accounts** icon to go to the **Accounts** page.
2. Select the account that has the punchout feature enabled.
3. Select the **Punchout** tab of the particular punchout account that requires an authorization code.
4. Click the **Generate** button to generate a unique authorization code. A warning appears that tells you that if you continue, the new authorization code will be saved immediately. If you have already generated a code before, this code will replace the existing code. If this is the case, make sure you update any external sources that are using the existing code to access this account.
5. Click **Save** to save the code that you have generated. The authorization code is shared with the procurement system implementing punchout. You must manually communicate this code (for example, via phone or email) to someone on the procurement side during implementation of the procurement system integration. After this is done, the procurement system sends it to Oracle Commerce with each request.

Set the punchout cart time setting

The administration user interface also lets you set a time period after which the system removes a punchout cart with its items. Specifically, this setting is used to clear off old

carts which have not been punched out and are no longer needed. This is done as follows:

1. In the administration interface, click the **Settings** icon.
2. Select **Order Settings**.
3. In the Punchout Carts section of the page, enter the number of days you wish as the time period after which the system removes a punchout cart. 7 days is the default value. You may enter values up to 99 days.
4. Click **Save** to save your setting.
5. Configure the service that runs periodically to remove incomplete punchout orders that have exceeded the time limit you specified. See Set the frequency of incomplete punchout order clean up.

This setting is global across all sites and appears if there is at least one business account.

Set the frequency of incomplete punchout order clean up

A service runs periodically to review the order repository and remove any incomplete punchout orders that have exceeded the time limit specified in the **Days Until a Punchout Cart is Removed** setting. (See Set the punchout cart time setting.) To set the initial frequency of the order cancellation service, you issue a `POST` request to the `scheduledJobs` endpoint, with a payload that specifies the `PunchoutOrderScheduledJob` component and the schedule, an example of which is provided below. To update the schedule, you issue a `PUT` request to the same endpoint.

```
POST /ccadmin/v1/merchant/scheduledJobs

{
"componentPath": "PunchoutOrderScheduledJob",
"scheduleType": "periodic",
"schedule":
  {
   "period" : 1000000
  }
}
```

The `scheduleType` and `schedule` properties determine the frequency used when running the service. Setting these properties is described in detail in the Configure the scheduled order service section.

Learn about the generic punchout shopper profile

For each account that uses punchout, a generic punchout shopper profile is created. Even though Oracle Commerce receives an email address and name in the shopping request from the procurement system, these are only for the purpose of displaying to the shopper (should the merchant decide to customize a widget to do so). From the perspective of Oracle Commerce, the generic punchout shopper profile is associated with all punchout shopping requests.

The generic punchout shopper profile contains the following information:

- Last Name = “User”
- First Name = “Punchout”
- email address = `punchoutuser@organizationId.com`

This generic punchout shopper is created when the first valid request is received from the procurement system. The generic punchout shopper is also returned by the contact APIs.

In the top-level Contacts list in the administration user interface, the Account Contacts list in the administration user interface, and the Account Contacts widget in Storefront, you can see the last name, first name, and email address of the generic punchout shopper in the list view. This is the same as for a regular shopper. No link is provided to open the contact details, however.

For the top-level Contacts list in the administration user interface, if there is a generic punchout shopper in any account, you will see a message stating that the list may include system-generated punchout contacts that are read-only. You will also see this message in the Account Contacts list in the administration user interface and the Account Contacts widget if the account has a generic punchout shopper.

This following is some additional information about the generic punchout user profile:

- A user who knows the URL of the contact details of the generic shopper profile can view it.
- The API can operate on the generic punchout shopper just as on any other contact.
- A generic punchout shopper profile cannot receive emails.
- A generic punchout shopper cannot log into the Storefront except programmatically as part of the punchout session.
- If the punchout shopper's session expires, the shopper will have to return to the procurement system and then punch out again.
- If an account stops using punchout and then later re-starts using punchout, when the system receives the first new punchout request, it will ensure that the account's generic punchout shopper is in place correctly. For example:
  - If someone had used an API to remove the generic shopper profile from the account, the system puts it back under the account.
  - If someone had used an API to deactivate the generic shopper profile, the system re-activates it.
- When you choose to add an existing contact to an account, a dropdown of existing contacts is displayed to choose from, remove other accounts' generic punchout shoppers from the dropdown list.

Learn more about using punchout

The following are recommendations and notes to keep in mind when enabling the punchout feature:

- Commerce does not collect shipping information and does not calculate taxes for a punchout cart.
- Commerce does not currently store the punchout shopper's first name, last name, or email address on carts. Persisted punchout carts are tracked using the order ID.
- The punchout shopper's first name, last name, and email address are currently not mandatory. A merchant can display them, however, if needed. See Display information about a punchout shopper in a widget for more information.

- By default, an order originating from a punchout account can be paid using the invoice payment method only. However, merchants can customize the server side extensions to support other payment methods.

Work with the punchout server-side extension

Commerce punchout functionality is provided through SSEs that can run on the `Node.js` server associated with your Commerce environment.

To use the punchout features, you must download the extensions from the Commerce administration server. You then customize the extension and upload it to your `Node.js` server. The punchout server-side extensions implement custom REST endpoints, which have the prefix `/ccstorex/custom`.

Commerce includes punchout server-side extensions that you can download and customize for your environment. This section describes the punchout extensions that are included with Commerce. See in Use developer tools to customize your store for details about how to download and customize server-side extensions.

The following table describes the punchout server-side extensions.

Server-side extension	Description
`punchout-app.zip`	Includes functionality that enables the punchout flow between the procurement system and Commerce. Use this sample application to customize the default functionality provided in the `punchout-lib` library.
`punchout-lib.zip`	Supports setup, edit and complete flows for punchout in cXML standard with Commerce. Do not make any changes to this library.
`purchase-order-app.zip`	Includes functionality that enables the purchase order flow between the procurement system and Commerce. Use this sample application to customize the default functionality provided in the `purchase-order-lib` library.
`purchase-order-lib.zip`	Supports submitted orders in Commerce from procurement system's purchase order in cXML format. Do not make any changes to this library.

Each ZIP file includes a `readme.md` file that describes classes and endpoints and includes information about how to install and extend the extension.

The server-side extensions provide the core punchout functionality with the following JavaScript classes. You can extend these classes, for example, to customize mappings or make additional Commerce REST API requests.

Class Name	Description
`PunchOutSetup`	Provides methods to authenticate, create the shopper token, and create the shopping cart in Commerce if re-punchout is required.
`PunchOutComplete`	Provides methods that convert the order request body JSON to the cXML `PunchOutOrderMessage`, which is sent to the procurement system.

Class Name	Description
`PurchaseOrder`	Provides methods that call the store `priceOrder` endpoint, check if prices are within tolerance limit, and create the order.
`PunchOutUtils`	Provides utilities for functionality such as fetching SKU prices, calling Commerce APIs, and parsing XML.

Work with the punchout endpoints

This section describes the endpoints included in the punchout server-side extension. All the endpoints are public URLs and all requests must be sent via HTTPS.

`punchoutSetUp` and `punchoutRedirection` endpoints

Issue a `POST` request to the `/ccstorex/custom/v1/punchOut/punchOutSetUp` endpoint to establish a punchout (or re-punchout) session from the procurement system.

The request includes organization details, the shared secret key, and other relevant information. It generates the OAuth token for logging in the punchout shopper, caches the token, and generates a unique ID against it.

The `GET /ccstorex/custom/v1/punchOut/punchOutRedirection/{sessionId}` endpoint handles the redirection to the Commerce storefront page from the procurement system. Using the `uniqueID` from the URL, this endpoint fetches the actual OAuth token from the cache. A form, which has the OAuth token as the input, is automatically submitted to the storefront home page. On submission, the storefront extracts the OAuth token from the request body and logs the punchout shopper into the system. The cache entry is then deleted.

Note: For storefronts built with Open Storefront Framework, this configuration happens at the framework level at runtime.

The following example shows a sample cXML request body. In the header, `Identity` is the Organization ID and `SharedSecret` is the organization's authorization code.

```
<cXML>
    <Header>
        <Sender>
            <Credential domain='organizationId'>
                <Identity>or-10001</Identity>
                <SharedSecret>authorization_code</SharedSecret>
            </Credential>
        </Sender>
    </Header>
    <Request>
        <PunchOutSetupRequest operation='create'>
            <BuyerCookie>1CX3L4843PPZO</BuyerCookie>
            <BrowserFormPost>
            <URL>http://localhost:1616/punchoutexit</URL>
          </BrowserFormPost>
            <Contact>
                <Name>buyer_name</Name>
                <Email>buyer_email</Email>
                <Extrinsic name='lastName'>punchout</Extrinsic>
```

```
            </Contact>
          </PunchOutSetupRequest>
    </Request>
</cXML>
```

The following example shows a sample cXML response, which returns the Commerce storefront URL returned by the `punchoutRedirection` endpoint, suffixed with the unique ID: as `form-url-encoded` values:

```
<cXML>
<PunchOutSetupResponse>
    <StartPage>
      <URL>http://admin-server-hostname/ccstorex/custom/v1/punchout/
punchoutRedirection/d5a3990d-1d47-4b8d-9e27-5a4d269e3efe</URL>
    </StartPage>
</PunchOutSetupResponse>/
</cXML>
```

`punchoutComplete` endpoint

When the punchout shopper has finished adding items to their cart and wants to return to their procurement system, issue a `POST` request to the `/ccstorex/custom/v1/punchOut/punchoutComplete` endpoint to convert the order JSON to `PunchOutOrderMessage` cXML. See Add a punchout checkout button to the Order Summary widget for a sample widget that implements this endpoint.

The request body is a JSON representation of the punchout shopper's order and the response body is a cXML representation of the order. See Order submit webhook for a sample JSON representation of an order.

`purchaseOrder` endpoint

Issue a `POST` request to the `/ccstorex/custom/v1/punchOut/purchaseOrder` endpoint to convert the procurement system's purchase order cXML to JSON and create an order in Commerce.

By default, this server side extension supports invoice payment only, but. you can customize the extension to support other payment methods.

The following example shows a sample cXML request body.

```
<cXML>
  <Header>
      <Sender>
          <Credential domain='organizationId'>
              <Identity>or-100001</Identity>
              <SharedSecret>key</SharedSecret>
          </Credential>
      </Sender>
  </Header>
  <Request>
    <OrderRequest>
    <OrderRequestHeader orderID="DO102880"
orderDate="2012-08-03T08:49:09+07:00" type="new">
```

```
        <Contact>
            <Name>First Name</Name>
            <Email>Email@example.com</Email>
            <Extrinsic name='lastName'>John_Smith</Extrinsic>
            <Extrinsic name='parentOrganization'>or-100001</Extrinsic>
        </Contact>
        <Total>
            <Money currency="USD">86.50</Money>
        </Total>
        <ShipTo>
            <Address isoCountryCode="US" addressID="1000467">
                <Name xml:lang="en">Acme, Inc.</Name>
                <PostalAddress name="default">
                    <DeliverTo>John Q. Smith</DeliverTo>
                    <DeliverTo>Buyers Headquarters</DeliverTo>
                     <Street>123 Main Street</Street>
                     <City>Mountain View</City>
                    <State>CA</State>
                    <PostalCode>94089</PostalCode>
                    <Country isoCountryCode='US'>United States</Country>
                </PostalAddress>
                 <Email name="default">john_smith@example.com</Email>
                <Phone name="work">
                    <TelephoneNumber>
                        <CountryCode isoCountryCode="United States">1</
CountryCode>
                                <AreaOrCityCode>800</AreaOrCityCode>
                                <Number>5555555</Number>
                            </TelephoneNumber>
                      </Phone>
            </Address>
        </ShipTo>
        <BillTo>
            <Address isoCountryCode="US" addressID="12">
                <Name xml:lang="en">Acme Accounts Payable</Name>
                <PostalAddress name="default">
                    <Street>124 Union Street</Street>
                    <City>San Francisco</City>
                    <State>CA</State>
                    <PostalCode>94128</PostalCode>
                    <Country isoCountryCode="US">United States</Country>
                </PostalAddress>
                <Phone name="work">
                    <TelephoneNumber>
                        <CountryCode isoCountryCode="US">1</CountryCode>
                        <AreaOrCityCode>415</AreaOrCityCode>
                        <Number>6666666</Number>
                    </TelephoneNumber>
                </Phone>
            </Address>
        </BillTo>
        <Shipping>
            <Money currency="USD">10.00</Money>
            <Description xml:lang="en-US">FedEx 2-day</Description>
        </Shipping>
```

```
        <Tax>
            <Money currency="USD">1.5</Money>
            <Description xml:lang="en">CA State Tax</Description>
        </Tax>
    </OrderRequestHeader>
    <ItemOut quantity="2" lineNumber="1">
        <ItemID>
            <SupplierPartID>Camera_1002</SupplierPartID>
            <SupplierPartAuxiliaryID>SKU_3005A</SupplierPartAuxiliaryID>
        </ItemID>
        <ItemDetail>
            <UnitPrice>
                <Money currency="USD">10</Money>
            </UnitPrice>
            <Description xml:lang="en">Laptop Notebook, 300 MHz</Description>
            <UnitOfMeasure>EA</UnitOfMeasure>
            <Classification domain="UNSPSC">43171801</Classification>
            <URL>http://www.example.com/Punchout.asp</URL>
            <Extrinsic name="ExtDescription">Enhanced keyboard</Extrinsic>
        </ItemDetail>
        <Shipping>
            <Money currency="USD">10.00</Money>
            <Description xml:lang="en-US">standardShippingMethod</Description>
        </Shipping>
        <ShipTo>
            <Address isoCountryCode="US" addressID="1000467">
                <Name xml:lang="en">Acme, Inc.</Name>
                <Email name="default">john_smith@exmaple.com</Email>
                <Phone name="work">
                <TelephoneNumber>
                <CountryCode isoCountryCode="United States">1</CountryCode>
                <AreaOrCityCode>800</AreaOrCityCode>
                <Number>5555555</Number>
                </TelephoneNumber>
                </Phone>
                <PostalAddress name="default">
                    <DeliverTo>John Q. Smith</DeliverTo>
                    <DeliverTo>Buyers Headquarters</DeliverTo>
                    <Street>123 Main Street</Street>
                    <City>Mountain View</City>
                    <State>CA</State>
                    <PostalCode>94089</PostalCode>
                    <Country isoCountryCode="US">United States</Country>
                </PostalAddress>
            </Address>
        </ShipTo>
        <Distribution>
            <Accounting name="DistributionCharge">
                <AccountingSegment id="7720">
                    <Name xml:lang="en-US">Account</Name>
                    <Description xml:lang="en-US">Office Supplies</Description>
                </AccountingSegment>
                <AccountingSegment id="610">
                    <Name xml:lang="en-US">Cost Center</Name>
                    <Description xml:lang="en-US">Engineering Mgt</Description>
```

```
            </AccountingSegment>
        </Accounting>
        <Charge>
            <Money currency="USD">20.00</Money>
            <!--<Percentage percent="20"/>
            <Money currency="USD">0.00</Money>-->
        </Charge>
    </Distribution>
    <Tolerances>
        <PriceTolerance>
            <Money currency="USD">100.00</Money>
        </PriceTolerance>
    </Tolerances>
  </ItemOut>
 </OrderRequest>
</Request>
</cXML>
```

The following example shows a sample cXML response body for the previous request.

```
<cXML>
  <Response>
     <Status code="200" text="OK. Order ID: o30630"/>
  </Response>
</cXML>
```

Configure your storefront for punchout shoppers

You must make changes to a number of the storefront layouts as part of the punchout integration.

To learn how to make changes to existing layouts, widgets, and elements, see Design Your Store Layout. To learn about creating custom widgets and elements, see Create a Widget.

This section includes the following topics:

- Design the storefront for punchout shopping
- Indicate that a logged-in shopper is a punchout shopper
- Add a punchout checkout button to the Order Summary widget
- Display information about a punchout shopper in a widget

Design the storefront for punchout shopping

Keep the following in mind when you design your store layout to support punchout shopping.

- Oracle recommends that you create a separate site just for punchout shoppers. The punchout site has its own URL, which you provide to the procurement system, plus its own catalog, price group, and contract terms. See Configure Sites to learn how to create additional sites.
- Punchout shoppers require only a subset of the features you might otherwise provide on your store and they might be confused if they see design elements that

do not apply to them. For example, punchout shoppers do not select shipping methods or have wish lists.

A punchout site should include Home, Product, Collection, and Cart layouts.

However, to ensure a smooth experience for punchout shoppers, remove the following components from your punchout site:

- Any layouts related to profiles, including Profile, Order History, Purchase List, and Scheduled Order.
- Any of the checkout layouts, including Checkout, Checkout with GiftCard, Checkout Payment, and Checkout Stack.
- All wish list layouts, widgets, and elements.
- Any links to external sites.
- The Login/Registration and My Account elements in the Header widget.
  It is important to remove these components because shoppers who did not arrive at the site via punchout should not be able to log in or complete an order. If the shopper's session expires, do not prompt them to log in again; redirect them to the procurement system, instead.
- The Cart Shipping widget from the Cart layout. Instead, you can add a button to the Cart layout that shoppers click when they want to return to the procurement system. See Add a punchout checkout button to the Order Summary widget for more information.
- The Checkout button element in the Cart Summary widget. You could also modify this button to redirect the shopper to the Cart layout, where they can click a button that sends them back to the procurement site. See Add a punchout checkout button to the Order Summary widget for more information.

Indicate that a logged-in shopper is a punchout shopper

A flag that specifies that a shopper has been redirected from a procurement site is available from the User view model. However, it is not included in the out-of-the-box widgets. If you want to indicate that a shopper is a punchout shopper in a widget, you can do so using code similar to the sample below:

```
widget.user().isPunchout()
```

See Add a punchout checkout button to the Order Summary widget for a code sample that uses this functionality in a method that redirects the punchout shopper back to the procurement system.

Add a punchout checkout button to the Order Summary widget

The Order Summary widget (available for the Cart layout) lets the shopper review their order from the cart page before proceeding to checkout. This section describes how to add a button to the Order Summary widget that a punchout shopper clicks when they are ready to complete their order.

First, customize the Order Summary widget's JavaScript file to handle the button click. In the following example, the `handlePunchoutOrder()` function handles the button click.

```
handlePunchoutOrder: function() {
       var widget = this;
       if(data.cart().items().length > 0) {
```

```
        data.cart().validatePrice = true;
        data.cart().skipPriceChange(true);
        data.redirectToProcurement("SSEURL",false);
      }
      return true;
    },
```

The following method redirects the shopper back to their procurement system.

```
/**
 * Method to redirect punchout user back to procurement using Form
post.
 *
 * Before redirecting, call the server-side extension URL with
orderjson
 * to convert it to CXML and post it to the procurement system.
 */
redirectToProcurement : function(){
  var widget=this;
  if(widget.user().isPunchout()){
    //get request data saved in local storage when shopper was
 * redirected from procurement to Commerce
    var punchoutStorageData =CCRestClient.
getStoredValue(CCConstants.LOCAL_STORAGE_ADDITIONAL_FORM_DATA);
    if(punchoutStorageData){
      var punchoutStorageObject =JSON.parse(punchoutStorageData);
      if(punchoutStorageObject){
        widget.punchoutBuyerCookie
=punchoutStorageObject.buyerCookie?
punchoutStorageObject.buyerCookie:null;
        widget.punchoutBrowserFormPost
=punchoutStorageObject.browserFormPost?
punchoutStorageObject.browserFormPost:null;
      }
    }
    //Creating form element
    var form = document.createElement('form');
    document.body.appendChild(form);
    form.method = 'post';
    //Setting form action URL sent by procurement
  form.action = widget.punchoutBrowserFormPost;
  if(widget.user().orderId() && widget.user().orderId() != ''){
      var contextObj = {};
      var data ={};
      contextObj[CCConstants.ENDPOINT_KEY] =
CCConstants.ENDPOINT_GET_ORDER;
      var filterKey =
widget.cart().storeConfiguration.getFilterToUse(contextObj);
      if (filterKey) {
        data[CCConstants.FILTER_KEY] = filterKey;
      }
      //Get Order Call
      CCRestClient.request(CCConstants.ENDPOINT_GET_ORDER, data,
        function(data) {
```

```
            //Success function of get order
            //Call to SSE to convert Order Json to CXML
  $.ajax({
     type: 'post',
     url: 'http://example.com:8080/ccstorex/custom/v1/punchOut/
punchOutComplete',
              data: JSON.stringify(data),
              headers:{"X-BuyerCookie":widget.punchoutBuyerCookie},
              contentType: "application/json",
              dataType: 'text',
              success: function (response) {
     //setting CXML order in form element
     var input = document.createElement('input');
     input.type = 'hidden';
     input.name = "cxml-urlencoded";
     input.value = response;
     form.appendChild(input);
     //cleanup in Commerce before form submission
     CCRestClient.
clearStoredValue(CCConstants.LOCAL_STORAGE_ADDITIONAL_FORM_DATA);
                CCRestClient.clearStoredValues();
                widget.user().clearUserData();
                widget.cart().clearCartForProfile();

                form.submit();
              }
          });

   },
   function(data) {
     navigation.goTo(widget.contextData.global.links['404'].route);
   },
   widget.user().orderId());
  }else{
  //If there is no order id, redirect with empty cart.
  //removing punchout related data from localstorage and empty cart;

CCRestClient.clearStoredValue(CCConstants.LOCAL_STORAGE_ADDITIONAL_FORM_DATA)
;
    CCRestClient.clearStoredValues();
    widget.user().clearUserData();
    widget.cart().clearCartForProfile();
    form.submit();
  }
 }
}
```

Display information about a punchout shopper in a widget

When a procurement system directs a punchout shopper to your store, it can also pass the shopper's name and email address. Commerce maintains this data in local storage, where widgets can access it.

The following sample customizes the JavaScript file for the Contact Login for Managed Accounts element, which is available for the Header widget, to display the punchout

shopper's name and email address. For more information about this element, see Create Page Layouts that Support Different Types of Shoppers.

```
//Changes to be added in onLoad method of element.js file in contact-
login-for-managed-accounts-v2 element of header widget
onLoad : function(widget) {
    var self = this;
    var afterLogIn = false;
    if(widget.user().isPunchout()){
// Get user details from local storage punchout form post data and
// override firstName,lastName and Email
        var punchoutStorageData =
CCRestClient.getStoredValue(CCConstants.LOCAL_STORAGE_ADDITIONAL_FORM_D
ATA);
        if (punchoutStorageData) {
            var punchoutStorageObject =
JSON.parse(punchoutStorageData);
            if (punchoutStorageObject) {
                var punchoutUserFirstName =
punchoutStorageObject.firstName;
                if(punchoutUserFirstName){

widget.user().firstName(punchoutStorageObject.firstName);

widget.user().loggedInUserName(punchoutStorageObject.firstName);
                }
                if(punchoutStorageObject.lastName){

widget.user().lastName(punchoutStorageObject.lastName);
                }
              if(punchoutStorageObject.emailAddress){

widget.user().emailAddress(punchoutStorageObject.emailAddress);
                }
            }
        }
    }
}
```

36
Perform Bulk Export and Import

In an environment where Oracle Commerce interacts with an external system, you may want to exchange data. Oracle Commerce includes REST web services APIs that allows you to select data items and export or import them in bulk.

This framework allows you to perform large data transfers between different environments or systems of items such as accounts, profiles and promotions.

The following is a list of bulk import and export IDs that are available:

ID	Comment	Supports Import	Supports Export
`Profiles`	Both shopper and account-based profile data.	Yes	Yes
`AccountsV2`	Accounts data.	Yes	Yes
`Products`	Product data.	Yes	Yes
`ProductsV2`	Product data. This is faster than the Products plugin as this import generates a `failedAssociationRecordFile` for any associations that failed even though the product updates are successful.	Yes	Yes
`Catalogs`	Catalog data.	Yes	Yes
`Inventory`	Inventory data, including location-based inventory.	Yes	Yes
`Collections`	Collections data.	Yes	Yes
`Promotions`	Promotions data.	Yes	Yes
`Prices`	Prices data.	Yes	Yes
`Addresses`	Address data for profiles and accounts.	Yes	No
`Relationships`	Relationship data between profiles and accounts.	Yes	No

Note: All export and import APIs discussed in this section refer to bulk import and bulk export APIs. For information on individual export and import, refer to Import and Export Catalog Items and Inventory.

Understand Bulk Exporting And Importing

To perform an export or an import, the REST APIs consist of two endpoints that create or read a data file.

The `exportProcess` exports data, while the `importProcess` imports data. These endpoints allow you to generate or read data files in two modes:

- Archived mode
- Standalone mode

Archived mode

When you initiate an export or import in archived mode, you work with a single archive file that can contain more than one data file, allowing you to manipulate several data files at a time. This is the default mode for performing exports and imports.

The data file in archived mode is in ZIP format and must include the `content.json` file, which contains information about the other files in the data ZIP file. The following is an example of a `content.json` file:

```
[
  {
    "fileName": "Profiles.json",
    "format": "json",
    "id": "Profiles"
  },
  {
    "fileName": "Accounts.json",
    "format": "json",
    "id": "Accounts"
  }
]
```

Each JSON in the JSON array corresponds to the data file with the following parameters:

- `fileName` - The name of the data file.
- `id` - The operation ID.

Note that each data file inside the ZIP must be less than 100MB. The ZIP file will be rejected if any data file size, after extraction, is greater than 100MB.

For example, the following export request is in archived mode. It generates two different files that are combined into a single archive file:

```
POST /ccadmin/v1/exportProcess
        {
          "fileName": "profile.zip",
          "items": [
            {
              "id": "Profiles",
              "format": "json"
            },
            {
              "id": "Accounts",
              "format": "json"
            }
          ]
        }
```

As displayed in the above example, the request contains the `fileName` parameter, which represents the name of the archive file. It also contains the `items`, `id` and `format` parameters.

Standalone mode

When you initiate an export or import in standalone mode, you work with a single data file. To transfer data in the standalone mode, use the `mode` parameter to identify the process mode type as `standalone`.

The following is an export request in standalone mode, which creates a single file:

```
POST /ccadmin/v1/exportProcess
{
  "fileName" : "product.json",
  "mode" : "standalone",
  "id" : "Products",
  "format" : "json"
}
```

As displayed in the above example, the request contains the `fileName` parameter, which represents the name of the data file, as well as the `id` and `format` parameter.

Status Files

When you receive either export or import results, the payload provides you with links that you can use to download the export and/or status files. For example, when exporting, the export `fileLink` URL provides information on how many accounts were in the exported file. The `metaLink` URL identifies successful and unsuccessful counts. If there are any failures during export or import, they will be displayed in the `metaLink` file.

Note: Uploaded import files, as well as generated export files, are stored for a period of time before they are automatically removed from the system.

Export data endpoints

The following information shows you how you can perform an export. For detailed information on the REST APIs used in this section, refer to the REST API documentation.

To export data, perform the following steps:

1. Trigger the export using the `executeExport` endpoint.
2. Monitor the export status using the `getExportProcess` endpoint. Once the export has finished, download the exported and status files.
3. You can stop the export process by using the `abortExportProcess` endpoint if necessary.

An export can be initiated using the following steps:

1. Initiate the export by using the `exportProcess` endpoint. Provide the name of the file that will contain the exported data. The following example is of an archived export:

```
[
  {
    "fileName": "Profiles.json",
    "format": "json",
    "id": "Profiles",
```

```
      "params" : {
         "q" : "id eq <id>"
      }
  },
  {
    "fileName": "Accounts.json",
    "format": "json",
    "id": "Accounts",
       "params" : {
          "q" : "id eq <id>"
       }
  }
]
```

The following example is of a standalone export:

```
POST /ccadmin/v1/exportProcess
     {
       "fileName": "product.json",
       "mode": "standalone",
       "accounts": "Products",
       "format": "json",
       "params" : {
          "q" : "id eq <id>"
       }
     }
```

Note that the number of records in the export should be limited to 100 thousand. Exporting large quantities of data can cause performance issues. You can do limit the record number by using the q parameter.

2. You can review the export progress by using the process ID returned from the `exportProcess` API. For example:

```
GET /ccadmin/v1/exportProcess/{processId}
```

3. Once the export job is done, the resulting payload provides the links you can use to download the export and status files. The following is an example of a response for a completed export in archive mode:

```
{
  "progress": "succeeded",
  "startTime": "2020-05-22T07:28:04.242Z",
  "links": [
    {
      "rel": "meta",
      "href": "<base_url>/export/rKjSiAtgt8WCzJ1YHXrAc5mTQ_10000/
exportStatus.zip"
    },
    {
      "rel": "file",
      "href": ""<base_url>/export/rKjSiAtgt8WCzJ1YHXrAc5mTQ_10000/
datafile.zip"
    },
```

```
    {
      "rel": "self",
      "href": ""<base_url>/exportProcess/rKjSiAtgt8WCzJ1YHXrAcmTQ_10000?
fileName=datafile.zip"
    }
  ],
  "endTime": "2020-05-22T07:28:05.434Z",
  "completed": true,
  "requestStatus": 200
}
```

The file URL can be used to download the exported data file. The meta URL is the status file that contains the following data:

- `startTime` - Indicates the start time of the export process.
- `endTime` - Indicates that the time that the export process ended.
- `successCount` - This indicates the number of records that were successfully exported.
- `failureCount` - The number of records that failed to successfully export.
- `failureExceptions` - This provides error details if there are any errors that were created during the export process.

A sample status file resembles the following:

```
{
  "endTime" : 1587782346639,
  "startTime" : 1587782346602,
  "failureCount" : 1,
  "successCount" : 202,
  "failureExceptions" : [Some error message]
}
```

Abort an export

An export can be aborted any time after triggering the export and before completion of the export. Once the export is aborted, the export process terminates and there are no partially exported files available.

To abort an export, use the `POST /ccadmin/v1/exportProcess/{processId}/abort` REST endpoint.

Import data endpoints

The following information shows how you can perform an import.

When you perform an import, you perform the following steps:

1. Upload the data file.
2. Trigger the import using the `executeImport` endpoint.
3. Monitor the import status using the `getImportProcess` endpoint. Once the import has finished, you then download the error and status files.
4. You can end the import process using the `abortImprotProcess` endpoint if necessary.

1. Use the following endpoint to upload your import file:

   ```
   POST /ccadmin/v1/files
   ```

   Use the following form parameters with this endpoint:

   - `fileUpload` - The file to upload.
   - `fileName` - An optional file name.
   - `uploadType` - Use the `bulkImport` parameter.

2. Initiate the import by using the `importProcess` endpoint. The following is an example of an archive import:

   ```
   {
     "fileName" : "profilesAndAccounts.zip"
   }
   ```

   Refer to the Archived Mode section for information on what is contained in the ZIP file.

   The following example is of a standalone import:

   ```
   POST /ccadmin/v1/importProcess
        {
          "fileName": "product.json",
          "mode" : "standalone",
          "id"  : "Products",
          "format": "json"
        }
   ```

3. You can review the progress of the import using the process ID that is returned from the `importProcess` API. For example:

   ```
   GET /ccadmin/v1/importProcess/{processId}
   ```

4. Once the import job is complete, the resulting payload provides the links that you can use to download the error and status files. The following is an example of a completed import in standalone mode:

   ```
   {
       "completedPercentage":100,
       "progress":"succeeded",
       "startTime":"2020-04-04T20:25:01.982Z",
       "links":[
           {
               "rel":"meta",
               "href":"<baseUrl>/import/ ezzPxcPRmJIlXYOTQNT9HMacA
   _10000/importStatus.json"
           },
           {
               "rel":"failedRecordsFile",
               "href":"<baseUrl>/import/ ezzPxcPRmJIlXYOTQNT9HMacA
   _10000/importFile.json"
   ```

```
        },
        {
            "rel":"failedAssociationRecordsFile",
            "href":"<baseUrl>/ezzPxcPRmJIlXYOTQNT9HMacA_10000/
failedAssociationRecordsFile.json"
        },
        {
            "rel":"self",
            "href":"<baseUrl>/importProcess/
ezzPxcPRmJIlXYCyOTQNT9HMacA_10000"
        }
    ],
    "endTime":"2020-04-04T20:25:02.190Z",
    "completed":true,
    "requestStatus":200
}
```

The meta URL is the status file that contains the following data:

- `startTime` - Indicates the start time of the export process.
- `endTime` - Indicates that the time that the export process ended.
- `successCount` - This indicates the number of records that were successfully exported.
- `failureCount` - The number of records that failed to successfully export.
- `failureExceptions` - This provides error details if there are any errors that were created during the export process.

The status file may look similar to the following:

```
{
  "endTime" : 1587782346639,
  "startTime" : 1587782346602,
  "failureCount" : 1,
  "successCount" : 202,
  "failureExceptions" : [Invalid Phone Number is provided for 133]
}
```

The `failedRecordsFile` contains the errors generated during the import process. The details in this meta file can be used to fix the error records in the `failedRecordsFile`, which can be re-imported once you have fixed the data.

For the ProductsV2 import process, the `failedAssociationRecordsFile` contains the records for which most of the data was correctly processed, but some specific associations failed. The `failedAsociationRecordsFile` can also be re-imported once the data has been fixed.

Abort an import

An import can be aborted any time after triggering the import and before completion of the import. The import is terminated only after the current batch of records has been processed.

To abort an import, use the `POST /ccadmin/v1/importProcess/{processId}/abort` REST endpoint.

Update import data

By default, when performing an import, existing records are updated if they are found in the repository. You can set the update flag to false if you do not want to update existing records as setting this flag to false can increase the speed of the import process if all of the records being imported are new records. For example:

```
{
   "fileName": "product.json",
    "id" : "Products",
    "mode" : "standalone",
    "format" : "json",
    "params" :{
       "update":"false"
    }
}
```

Understand export and import endpoint parameters

The `exportProcess` and `importProcess` endpoints contain the following request parameters:

Parameter	Description
`fileName`	The name of the file to read from or create during the process. This parameter is required except when performing an export or import in standalone mode.
`mode`	Indicates if the export or import is standalone or archived. This parameter is only required if using standalone mode.
`id`	The operation ID. This parameter is required when working in the standalone mode.
`format`	The format to use during the export or import. This parameter is used only when working in the standalone mode.
`items`	Applies only to export. A list of objects to be exported. This array contains the `format` and `id` properties of the export operation.

Parameter	Description

param	Passes additional parameters to plug-ins. For example, you can speed up an import process by not looking for an existing item to update using the following:
	
	`{`
	`  "fileName": "account.json",`
	`  "id" : "bulkAccountHandler",`
	`  "mode" : "standalone",`
	`  "format" : "json",`
	`  "params" : {`
	`  "update":"false"`
	`  }`
	`}`

prettyPrintJson	This is parameter is supported for all import and export IDs. The value can be `true` or `false`, which is the default. If set to `true`, the exported JSON will be formatted. Note that this parameter is supposed only for JSON formats.
headersList	This parameter, which is supported for all import and export IDs, controls the headers of an exported CSV file. Note that this parameter is supported only for CSV files.
exportFromRepository	For Products, ProductsV2, Collections, Catalogs, Prices and Promotions, only published data is exported. To export unpublished data, set the value of this parameter to `publishing`.
q	The `q` parameter filters records for the total set of records. It supports conditional operators, such as `eq`, `ne`, `gt`, `ge`, `lt`, `le`, `AND`, `OR`, etc. Use this parameter to restrict exported records to 100 thousand.
update	The value for this import parameter an be set to `true` or `false`, which is the default. If set to `false`, all records will be marked as create. Using this parameter can speed up the import process when import data contains only new records.
memberAssignmentEmailTemplate	This email template import parameter can be used with Accounts and AccountsV2 IDs to trigger email when a member is added or removed from an account.
accountDeactivatedEmailTemplate	This email template import parameter can be used with Accounts and AccountsV2 IDs to trigger email whenever an account is deactivated.
organization_site_available_v1	This email template import parameter can be used with Accounts and AccountsV2 Ids to trigger email when an account is reactivated or a new contract has been added.

Parameter	Description
`emailTemplate`	This email template import parameter can be used with Profiles IDs to trigger email when a new profile is created.
`siteId`	This email import parameter can be used with Profiles IDs. When triggering email, the links or other details included in the email are taken from the default site. To trigger email from a different site context, pass in a different `siteId` using this parameter.
`roleChangeEmailTemplate`	This email template import parameter is used with Profiles IDs to trigger email roles whenever a profile is changed.
`profileDeactivatedEmailTemplate`	This email template import parameter can be used with Profiles IDs to trigger email whenever a profile is deactivated.
`organizationChangeEmailTemplate`	This email template import parameter can be used by Profiles IDs to trigger email when a profile's account changes.
`triggerWebhook`	By default `Shopper Profile Create` or `Shopper Profile Update` webhooks are not triggered when a profile is created or modified. However, you can use this import parameter with Profiles IDs to trigger the webhook by setting this flag to `true`.

Note that you can also import and export your custom properties.

Notification webhooks for export and import

If you want to receive notification when an export or an import job completes, configure the Export or Import Complete webhooks as required for your external application. These webhooks contain links to status and export files for export processes, and status and failed record files for import processes.

1. In the Oracle Commerce administration interface, click the **Settings** icon.
2. Click **Web APIs**.
3. Open the **Import Complete** or the **Export Complete Event** API.
4. Enter the URL of your external application, as well as any authorization information necessary to access the system. Click **Save**.

Export and import account data

One of the Oracle Commerce items that can be exported and imported is accounts.

This enables you to export or import any accounts that you may have configured in your environment, as outlined in Import and Export Catalog Items and Inventory. When you export or import account data, you are generating or adding a list of accounts and their members.

Note that account-based storefronts may not be activated in your environment.

Account export and import fields

The following account-based commerce fields can be exported or imported:

Field Types	Description
Name	The name of the account.
Description	A description of the account.
Classification	This field is used to identify the type of account.
Account Type	Indicates the type of account such as company, division, department or group.
Contract Information	The information includes fields such as contract ID, terms, name, description, catalog, site and price list used.
Account Details	Account details such as DUNS number, logo, VAT reference number, unique ID and tax reference number.
Addresses	These fields include shipping and secondary addresses of the account. Note that the AccountV2 plug-in provides the format supported by the Integration Cloud Service.
Address type	An optional property. The value is validated when provided against the setup values. The shipping and secondary addresses include the address type field.
Payment Information	This field includes payment methods. Note that payment method types are not required fields, and therefore must be manually added to an account before shoppers can use the field. Additionally, the Boolean property, which is set to `true` by default, `useAllPaymentMethodsFromSite`, allows you to obtain the method from the site, as opposed to individual accounts. Note that if the property is set to `true`, it will override any methods set through the UI.
Shipping Information	This field includes shipping methods. Shipping method type is not a required field and must be manually added to an account before shoppers can use the field. The `useAllShippingMethodsFromSite` Boolean property allows you to obtain the method from the site, as opposed from individual accounts. Note that if the property is set to `true`, the existing shipping methods in the repository will be deleted and any shipping methods passed in during import will be ignored. When a new principal account is created, the property is set to `true` when the property has not been set and there are not shipping methods contained within imported data.
Members	Identifies members of the account.
Approval Settings	Indicates if the account has approval settings and delegated approval administrators. These settings are site specific.

Field Types	Description
Site Information	The `siteOrganizationproperties` property is a map that sets information for the site and organization, such as approval, contract, order price limits, delegated administrators and payment information, as described above.

Account import validation

During the import, the system verifies that the account name or the ID provided in the import file is actually in the repository. If the account name or ID in the given row is found, the repository item is updated with the data provided in the import file. If no data is found in the repository, a new account is created.

The following validations are performed for accounts:

- The DUNS number contains 9 digits.
- The account logo is valid.
- The contract ID is valid.
- Members added to the account have the profile type `b2b_user`.
- The phone number, company, address, city, state, country, and postal code have been provided.
- The postal code is valid.
- The country and state combination are valid.
- The name of the account cannot be “root.”
- The ID of the parent organization is valid.
- All site level fields are valid.
- The shipping and payment methods are valid and are associated with the site.
- The address type is validated against the setup values.

Approval settings during import

When accounts are configured with a list of approvers, specific validations occurs:

- A member cannot be removed from an organization if that member is the last active approver in the organization and approval is required, or the organization has at least one order that is either in the `pending_approval` or the `pending_approval_template` state.
- An organization’s order price limit cannot be updated when delegated approval management is active and the current value of the order’s price limit is different than the repository data.
- An organization’s approval required attribute cannot be updated if delegated approval management is active and the current value of the approval required is different than the repository data.
- You cannot activate an organization’s approval required attribute if no member of the organization has approval authorization.
- Approvals are site-specific and must reference a valid site.

Account email triggers during import

While performing an import, if a member is added or removed from an organization, an email is triggered, notifying you of the new status. To trigger emails during the import, send the specific email template as a parameter in the request payload. The following example shows how you would identify using an `organization_assigned` template:

```
{
  "fileName": "accountImport.json",
  "id": "Accounts",
  "format": "json",
  "mode": "standalone",
  "params": {
    "memberAssignmentEmailTemplate": "organization_assigned_v1"
  }
}
```

Account email triggers in multiple site environments

When working in a multiple site environment, an email is triggered during import that contains all of the site URLs related to the account and profile. Note that up to five sites are listed per account. Additionally, account changes, role updates, approval information, and password information is also included in the email, if applicable. All event information is displayed in a table that identifies the changes that occurred.

When you add a new member to an organization, an email with the name of previous organizations and roles, if any, are added.

Work with sub-accounts

Sub-accounts can be created and updated using the `parentOrganization` field. An account can have a single parent account, as well as its own sub-accounts. A principal account, or one that does not have any parent account, cannot inherit any properties. There is no fixed number of account hierarchy levels.

Note: When a new business account is created as a sub account (that is, as the child account of another account in an account hierarchy) there is no limit to the depth of the hierarchy, but sub accounts inherit account properties only up to the 14th level.

Sub-accounts can inherit the following derived organization properties if the sub-accounts properties are not specifically designed:

- `derivedUniqueId`
- `derivedDunsNumber`
- `derivedOrganizationLogo`
- `derivedVatReferenceNumber`
- `derivedTaxReferenceNumber`
- `derivedDescription`
- `derivedType`

Sub-accounts can be moved under other accounts if there are no circular references between the accounts. Additionally, a sub-account cannot be set to a parent account of itself. Each sub-account should have its ancestor organization details updated to provide account

information. Note that a root account cannot be associated to another account either as a parent account or as a sub-account.

When working with shipping and payment methods in a sub-account and site pair, the sub-account-based methods and the use of site- based methods are either both inherited or not inherited. For example, the sub-account shipping methods and the use of all site shipping methods properties are both inherited together and cannot be inherited individually. The inherited state of the these sub-account and site pairs are determined by the Boolean properties `useAllShippingMethodsFromSite` and `useAllPaymentMethodsFromSite`. The values used in the API will override those set in the UI.

Account email triggers when account is deactivated

When performing an import, if an account is deactivated, an email is triggered to notify you of the change of the account's status. To trigger emails during the import, process, send the specific email template as a parameter in the request payload. The following example shows how you would use the `accountDeactivatedEmail` template:

```
{
  "fileName": "accountImport.json",
  "id": "Accounts",
  "format": "json",
  "mode": "standalone",
  "params": {
    "accountDeactivatedEmailTemplate":"organization_deactivated_v1"
  }
}
```

Note that when an account has a contract for a specific site, moving an existing account to be a sub-account of that specific site will cause an email to be sent to all members of the sub-account. This is because the contract for the site is inherited. This also occurs if a new account is created as a child of an account using the API, and contacts are added in that same API call.

For detailed information on account configuration and working with accounts, refer to Understand accounts, contacts, and contracts.

Export and import profile data

Profiles contain information about shoppers, such as their name, phone number and email address.

You can export or import profile data, which you may have configured in your environment as outlined in Import and Export Catalog Items and Inventory, allowing you to share the data with external systems. The following section describes the fields and validation that occurs when exporting or importing profile data.

Profile export and import fields

The following table displays the profile fields that can be exported and imported:

Field	Description
firstName, lastName	The name associated with the profile. This includes the first and last name, which are required during import.
email	The email associated with the profile. This field is required when creating a new profile.
userSiteProperties	This is a site level property that contains the receiveEmail flag property for the site. It indicates if the profile is configured to receive email. This defaults to false.
receiveEmail	This is a user item level property that indicates if the profile is configured to receive email. This defaults to false.
registrationDate	This optional field indicates the date that the profile was registered.
profileType	The type of profile being exported or imported. If the profile type is b2b_user, the shipping and billing addresses are not imported. If the profile type is b2c_user, the organization is not imported.
shippingAddresses	The shipping addresses associated with the profile. There can be only one addressed denoted as the default shipping address. If phone numbers are provided in the shipping address, validation occurs to ensure they are in the correct format. Address type is an optional property, and the value is validated when provided against the setup values. Note that phoneNumber, address1, city, state, country and postalCode are required for a shipping address. The postalCode field and city and state combination are validated.
locale	This field, which is required when creating new profiles, indicates the locale associated with the profile. Validation occurs to ensure that the locale is valid.
roles	The roles associated with the profile. Validation occurs to ensure that the role is available with one of the parent organizations.
dateOfBirth	Date of birth associated with the profile. Validation occurs in the format you have chosen, either MM/dd/yy or DD/mm/YY.
shippingAddress, billingAddress	Export only. The contact information for the profile. Address type is an optional property, and the value is validated when provided against the setup values. Note that phoneNumber, address1, city, state, country and postalCode are required for a shipping address. The postalCode field and city and state combination are validated.

Field	Description
parentOrganization	Export only. The organization associated with the profile. The organization ID is used to associate the profile with the organization. **Note:** The parentOrganization can be updated during the bulk import process for account-based commerce profiles.
Price lists (including the list and sales prices, as well as shippingSurcharge)	Export only. This read-only field indicates the price lists associated with the profile.
Price list groups	Export only. This read-only field indicates the price list groups associated with the profile.

Note that the following audience-based profile properties are read only:

- lifetimeSpend
- lifetimeAOV
- numberOfOrders
- lastPurchaseAmount
- firstPurchaseDate
- lastPurchaseDate
- lastVisitDate
- previousVisitDate

For audience-based information, see Define Audiences.

Profile validations

The following validations are performed when importing profiles:

- The buyer role of a profile, which is the default role, must be present.
- If the profile type is b2b_user, the shipping and billing addresses are not included. If the profile type is b2c_user, the parent organization is not included.
- The login field defaults to the profile's email address.
- When creating an initial import file, the email, locale, firstName and lastName contain valid values.
- The receiveEmail flag is disabled by default.
- All phone numbers are in the correct format.
- All addresses contain a phoneNumber, address1, city, state, country and postalCode.
- The country and state combination is valid.
- The site properties have been set. This is the userSiteProperties that contains the receiveEmail property.

Profile approver role validations during import

When profiles are configured as approvers, specific validations occur:

If a profile is the last active approver in the organization and approval is required, or the organization has at least one order that is either in the `pending_approval` or the `pending_approval_template` state, the following modifications cannot be made:

- The profile approver role cannot be removed from the profile.
- The profile's active status cannot be removed.
- The parent organization of a profile cannot be modified.

Bulk profile import with the GDPR

If you are importing profiles and want to set the European Union General Data Protection Regulation (GDPR) consent values, ensure that you have the shopper's consent, which may have been given on the system from which you are exporting profiles. You should also set the consent date in the import file to the consent date that was recorded on the other system.

Note: If you do not have an explicit consent from the shopper, it is best that you do not set the consent fields during a bulk profile import. For detailed information on working with the GDPR, refer to the Manage the use of personal data section.

Create a new password during import

You can generate new passwords when importing profiles. When the import occurs, a token is triggered for the profile. When the shopper selects the token link, the system navigates the shopper to the change password page, where they have the option to provide their user name, and create a new password. The token is a one-time password that expires within a time frame that you configure. Generating a new token does not clear a shopper's existing password.

To allow a shopper to request a new password, use the `emailTemplate` parameter in the request. This provides the name of the email template used during the creation of new profiles, for example, `shopper_acc_reg_v1`. If an email template is not provided, email will not be sent.

The following is a sample profile import with a new password request:

```
{
  "fileName": "profile.zip",
  "items": [
    {
      "id": "bulkProfileHandler",
      "format": "json",
      "params": {
        "emailTemplate": "shopper_acc_reg_v1"
      }
    }
  ]
}
```

Understand triggering emails during import in a multiple site environment

When working in a multiple site environment, an email is triggered during import that contains all of the sites related to the profile. Note that up to five sites are listed per profile. Additionally, account changes, role updates, approval information, and password information is also included in the email, if applicable. All event information is displayed in a table that identifies the changes that occurred.

When you add a profile to an organization, an email with the name of previous organizations and roles, if any, are added.

Note: Email template settings are copied from the default site. They can be disabled and updated as needed.

Trigger webhook during import

When a new profile is created or an existing profile is updated during an import, a webhook can be triggered with the profile information. The `production-registerProfile` webhook is triggered when a profile is created. The `production-updateProfile` webhook is triggered when the profile is updated.

To trigger the webhook during import, you need to set the `triggerWebHook` parameter in the request to true. For example:

```
{
  "fileName": "profileImport.json",
  "id": "Profiles",
  "format": "json",
  "mode": "standalone",
  "params": {
    "triggerWebHook": "true"
  }
}
```

Contact email triggers with deactivated contact

While performing an import, if a contact is deactivated, an email is triggered that notifies the contact of the new status. To trigger emails during the import, set the specific email template as a parameter in the request payload. The following example shows how you would identify using a `profileDeactivatedEmail` template:

```
{
  "fileName": "profileImport.json",
  "id": "Profiles",
  "format": "json",
  "mode": "standalone",
  "params": {
    "profileDeactivatedEmailTemplate":"re_v1"
  }
}
```

Contact email triggers when organization list changes

While performing an import, if a parent organization or secondary organization list is modified or deleted, an email is triggered, notifying the contact of the change. However, the email is only sent if the contact is active and is a member of an organization of at least one active account, or is an account that has an assigned contract. To trigger emails during the import, set the specific email template as a parameter in the request payload.

The following example shows how you would identify using an `organizationChangeEmailTemplate` template:

```
{
  "fileName": "profileImport.json",
  "id": "Profiles",
  "format": "json",
  "mode": "standalone",
  "params": {
    "profileDeactivatedEmailTemplate":"re_v1"
  }
}
```

Export and import product data

Product data can be exported or imported from external systems.

Once you have set up products and SKUs, as described in Create and work with products and Create and work with SKUs, you can share the data.

Products are associated with Master View or standard catalogs or collections. For detailed information on the difference between these two types of catalogs, refer to Understand catalogs.

When you create a product the product is linked, by default, to the master `defaultCategoryForProducts` category. If you indicate that the category is null, or you set the `orphaned` attribute to `true`, the product you create is identified as orphaned. If you provide a valid a standard catalog ID, the product is linked to the `defaultCategoryForProducts` category of the specific catalog.

Products cannot be linked to root categories of any catalog. If you provide root categories in the `parentCategories` attribute, the product is linked to the `defaultCategoryForProducts` of that specific catalog. However, if you provide both `parentCategories` and a `catalogId`, the `catalogId` is ignored.

Note that the `catalogId` and orphaned attributes are not exported, as these properties are used only when performing an import.

Product-specific export and import fields

The following sections describe the fields that can be exported and imported for products. Note that both the product and SKU data are imported into and exported to a single file. The product and SKU fields have been separated in the following tables for the sake of clarity.

Product import issues

If you import a large number of products, you could experience some performance issues if the products are not categorized properly. To avoid these issues, consider the following:

1. A collection should not hold too many products.
2. Avoid creating too many products without any category.
3. Avoid creating too many price groups.

Field Type	Description
Product price information	Fields that contain pricing information, such as `listPrice`, `listVolumePrice`, `listPrices`, `listVolumePrices`, `salePrice`, `saleVolumePrice`, `salePrices`, `saleVolumePrices`, `shippingSurcharge`, `shippingSurcharges` and `taxCode`. You can upload pricing information into multiple price list groups.
Tax information	The tax code of the product.
Product Add-Ons	Add-on product associated with the product with the property `addOnProducts`. Add-Ons are not supported at SKU level. Only at product level.
Product description information	Fields that describe the product, such as `height`, `weight`, `length`, `brand`, `type` and `displayName`.
Product image information	The product image description field `productImages` that contains `path`, `metadata`, `repositoryId`, `name`, `url` and `tags`. Also describes additional information about images, such as `productImagesMetadata`, `primaryImageTitle` and `primaryImageAltText`. Note that you cannot import images along with the corresponding product. However, you can map images that you have already uploaded to the products that you are importing. To do this, refer to the Import image assignments.
Additional product information	Fields that contain information such as `unitOfMeasure`, `onlineOnly`, `configurable`, `discountable`, `orderable`, `active`, `notForIndividualSale`, `orderLimit`, `variantValuesOrder` and `CountryOfOrigin`.
Associated SKU descriptions	Fields that contain descriptions about associated SKUs, such as `childSKUs` and `defaultProductListingSku`.
Additional related product fields	Fields that contain additional product and related fields descriptions, such as `description`, `longDescription`, `relatedProducts`, `relatedArticles` and `relatedMediaContent`.
Date and time	Fields that provide dates and times, such as `salePriceStartDate`, `salePriceEndDate`, `creationDate`, `arrivalDate`, `dateAvailable` and `daysAvailable`.

Field Type	Description
Linking information	Fields that provide information on linking fields. For example, `parentCategories`. Note that `parentCategories` is imported first and during product import, it is linked to the corresponding parent categories using its `repositoryId`.

SKU-specific export and import fields

The following fields can be exported and imported for SKUs.

Field Type	Description
SKU price information	Fields that contain pricing information, such as `listPrice`, `listVolumePrice`, `listPrices`, `listVolumePrices`, `salePrice`, `saleVolumePrice`, `salePrices`, and `saleVolumePrices`.
SKU image information	The SKU image description field images that uses `path`, `metadata`, `repositoryId`, `name`, `url` and `tags`. Note that you cannot import images along with the corresponding SKU. However, you can map images that you have already uploaded to the SKUs that you are importing. To do this, refer to the Import image assignments.
Product related information	Fields that describe related product data, such as `productFamily` and `productLine`.
Date and time	Fields that provide dates and times, such as `salePriceStartDate`, `salePriceEndDate`, `creationDate`, `arrivalDate`, `dateAvailable` and `daysAvailable`.
Boolean fields	Fields that describe SKU properties, such as `active`, `discountable`, and `configurable`.
Dynamic SKU information	Fields that contain information specific to dynamic SKUs, such as `color` and `resolution`.

Product and SKU validations

The following list describes some of the validation performed when importing products:

- The product ID is valid and not null. Valid characters are alphanumeric, as well as underscore (_) and dash (-).
- The `listPrices` and `salePrices` properties are either maps or null. If it is a map, each price corresponding to a price group ID is either a number or null.
- The `listVolumePrices` and `saleVolumePrices` properties are either a map or null. If it is a map, each price corresponding to a price group ID is either a map or null.
- The `listPrices` and `listVolumePrices` properties cannot contain a non-null entry for a particular price list group ID i.e. a particular product or SKU cannot have both list and list volume prices for a particular `pricelistgroup`.

- The `salePrices` and `saleVolumePrices` properties cannot contain a non-null entry for a particular price list group ID i.e. a particular product or SKU cannot have both sale and sale volume prices for a particular `pricelistgroup`.
- The `listPrice`, which should be a positive number, should be available for the mandatory price list groups.
- If a site not account-enabled, the `listPrice` of a product cannot be null.
- The `listPrice` cannot be less than the `salePrice`.
- The `shippingSurcharge` should be a positive number.
- If `categoryId` is null the `categoryItem` property will not be set.
- The `productImages` property tags should be null or a map and if not null, each of its elements should be an instance of map or string.
- The `childSKUs.configurationMetadata.name` property must be a string and is a required.
- The `propertychildSKUs.configurationMetadata.value` property must be a string and is optional.

The following validations are also performed when importing SKUs:

- The corresponding catalog and product IDs should be valid and not null.
- The active property should be Boolean or String.
- SKU properties cannot be null, and should be unique. Additionally, the SKU ID should be valid, and a SKU with the same ID cannot pre-exist in the repository.
- Inbound SKU data should have all of the mandatory variant properties.

SKUs can be bundled together to create a new SKU. SKU bundles have the following validation:

- A SKU cannot be deleted if it is part of a SKU bundle.
- You can only bundle one level of SKUs.
- A SKU cannot be a member of multiple bundles, and you cannot have bundles referenced within bundles.
- SKUs can only be a part of the same bundle once.

For detailed information on working with products and SKUs, refer to Create and work with SKUs .

Support for hierarchical price list groups

If you have set up price groups by using base price groups, this hierarchy is supported in bulk product import. Note that the following validations are not performed in bulk import for performance reasons:

1. The validation that `salePrice` value is lesser than list price value is not performed when there is price hierarchy. The price list group for the sale and list prices are added and inherit the parent price list group.
2. The validation that `salePrice` is given but the list price is missing is not performed when there is price hierarchy. The price list group for the sale price is added and inherits the parent price list group.

For more information about setting up price groups, see Configure Price Groups.

Export and import catalog data

Catalog data describes the structure of a catalog and specifies the catalog type (independent, filtered, or legacy).

By default, Commerce includes a single catalog called Product Catalog. However, merchants can create and assign additional catalogs. Commerce supports the following types of catalogs:

- Independent catalogs (version 2) are not dependent on any other catalog in your environment. The default Product Catalog is an independent catalog.
- Filtered catalogs (version 3) provide custom views into the Product Catalog.
- Legacy catalogs (version 1) are an older type of secondary catalogs that provide custom views into the Product Catalog.

For detailed information about the different types of catalogs see Understand catalogs.

During an import, the `supportVersion1Catalogs` attribute determines which type of catalog to use. If this attribute is set to `false`, and a catalog version is not provided, an independent catalog will be created. If the `supportVersion1Catalogs` attribute is set to `true`, then you can create both an independent and legacy catalog. If you set the `catalogVersion` property value to 1, a legacy catalog is created. If you set the `catalogVersion` property value to 2, an independent catalog is created. Note that the `catalogVersion` attribute is added to export data, and cannot be updated.

The `fixedParentCategories`, `ancestorCategories`, `parentCategoriesForCatalog` and `computedCatalogs` attributes are updated in the repository for each child category that is linked or to be unlinked from a catalog, including any of its child categories.

Top level root categories (both navigation and non-navigation) are made when creating a catalog and cannot be moved or deleted from the catalog. Root categories, however, are not made when creating a legacy catalog. The default Product Catalog cannot be deleted but it may be updated and exported.

The following is a sample request for a catalog import:

```
{
  "displayName": "Classical Movies Catalog",
  "rootCategories": [
    {
      "id": "category_1"
    },
    {
      "id": "category_2"
    }
  ],
  "id": "ClassicalMoviesCatalog"
}
```

The `catalogVersion`, `rootNavigationCategory` and `defaultCategoryforProducts` attributes are exported. You can export all catalogs, including the default master catalog, in the repository along with `rootCategories`. The `rootCategories` attribute in the response contains navigation and non-navigation categories for standard catalogs, and any number of

categories for Master View catalogs, and cannot be updated. The `fixedChildCategories` for navigation and non-navigation categories are exported for all catalogs, except for Master View catalogs.

The following is an example of a response:

```
{
  "catalogVersion": 2,
  "defaultCategoryForProducts": {
    "id": "category_1"
  },
  "rootNavigationCategory": {
    "id": "rootCategory_1"
  },
  "displayName": "Classical Movies Catalog",
  "rootCategories": [
    {
      "fixedChildCategories": [],
      "id": "nonNavigableCategory_1"
    },
    {
      "fixedChildCategories": [
        {
          "id": "cat40013"
        },
        {
          "id": "cat60023"
        }
      ],
      "id": "rootCategory_1"
    }
  ],
  "id": "ClassicalMoviesCatalog"
}
```

The following section describes the fields and validation that occurs when exporting or importing catalog data.

Catalog export and import fields

The following fields can be exported and imported for catalogs.

Field Name	Description
`displayName`	The name of the catalog.
`Id`	The ID of the catalog.
`rootCategories`	An array of categories associated with the catalog. From this field, you can export the `Id` fields of the categories.

Field Name	Description

translations	This field indicates the different names for an attributed based on the language. The current displayName is supported. The following is an example of a response:
	
	translations": {
	"items": [
	{
	"displayName": "Classical Movies Catalog",
	"lang": "en"
	},
	{
	"displayName": "[DE]Classical Movies Catalog[DE]",
	"lang": "de"
	}
	]
	}

catalogVersion	Specifies the type of catalog. 1 specifies a legacy catalog, 2 specifies an independent catalog, and 3 specifies a filtered catalog.
defaultCategoryForProducts	Identifies the default collection for a product. By default this links the product to the default collection in a site's default catalog. If this attribute is null, the product is orphaned.
rootNavigationCategory	Identifies the navigation category for a catalog.

When there is a relationship between entities such as a collection and products, relationship updates are Full updates. New relationships override existing ones during the import or export process.

During import, you can link independent collections to catalogs. When you create a Master View catalog, the top level root categories are not created. By default, the master catalog's navigation category is linked to the rootNavigationCategory attribute. Additionally, you can link any of the Master View catalogs to collections that are only available in the default master catalog. Errors will occur if you try to link a collection that is not in the master catalog.

When you create a catalog, the top level root categories are created; this is not the case with Master View Catalogs.

Catalog validations

The following validations are performed when importing catalogs:

- The required displayName is valid.
- The required catalogId is valid and must contain only alphanumeric, underscore (_) or dash (-) characters.
- The rootCategories array should have valid categories.

- The `catalogVersion` attribute value is validated based on the `supportVersion1Catalogs` attribute setting.
- The `defaultCategoryForProducts` attribute is validated to ensure that you are not linking a collection to a root category of a standard catalog.

Special handling of large products import

You may experience some performance issues if you import more than 100,000 products and the products are not properly categorized. To avoid import performance issue, please follow these guidelines:

1. A collection should not hold too many products. Try to restrict it to 10,000 products per collection
2. Avoid creating too many price list groups.

Import Parameter

You should pass the parameter `fullPublishMode` as `true` when initiating a large import. Use this option with care since the recording of the changes are disabled to speed up the import process. Full publish becomes mandatory after the imports are performed using this parameter. Note that "full publish" means that all changes will be published.

For example, a sample request payload to trigger initial data import:

```
{
  "mode": "standalone",
  "id": "Products",
  "format": "json",
  "params": {
    "fullPublishMode": "true",
    "update": "false"
  }
}
```

Note: `fullPublishMode=true` should only be used for initial data imports.

`publishChangeLists`

The `publishChangeLists` endpoint can be used to perform to publish all changes:

```
/ccadmin/v1/publishingChangeLists/publish {"operationType":"publish",
"skipDependencyCheck":"true"}
```

You can optionally set `skipDependencyCheck:"true"` to skip any other dependency checks from other modifications that are not necessary for publish operations.

Bulk import and export filtered catalogs

You can bulk import and export filtered catalogs. For detailed information on working with filtered catalogs, refer to Configure catalogs with optional collections.

Export and import category data

Data from categories (collections) can be exported or imported from external systems.

Once you have set up these items, as outlined in Understand catalogs, you can share their data.

Collections can link to an independent collection or a product. Navigation and non-navigation categories of any standard catalog can update all attributes.

The `fixedChildProducts` attribute for all root categories should be empty as a product cannot be linked directly to root categories. If you are trying to update the `fixedChildProducts` attribute while performing a root category update, link the product to the `defaultCategoryForProducts` category of the specific catalog, if present, or ignore if not present.

Note that catalog navigation and non-navigation categories cannot be a child category of any other category. When you create a category, it is linked to the root navigation category. If you set `orphaned=true`, the category is created as an orphan. If a valid standard catalog ID is provided, the category is linked to the navigation category of the specified catalog. Note that an error occurs if the Master View catalog ID is provided because a new category cannot be linked to a Master View catalog.

When you remove a category from the default master catalog, you must remove the same category from all of the Master View catalogs, if present, as the Master View catalogs should contain only the categories available in the master catalog.

The following is an example of a request:

```
{
  "catalogId": "cat_1",
  "orphaned": false,
  "displayName": "Drama",
  "fixedChildCategories": [
    {
      "id": "cat40011"
    },
    {
      "id": "cat40010"
    }
  ],
  "active": true,
  "id": "cat40015",
  "fixedChildProducts": []
}
```

Note that when exporting a `catalogId`, orphaned attributes will not be exported.

The following is an example of a response:

```
{
  "longDescription": null,
  "childProductsCount": 0,
  "route": "/drama/category/cat40015",
```

```
  "categoryImages": [],
  "displayName": "Drama",
  "categoryPaths": [
    "/Cloud Catalog/Storefront Navigation/Movie Store Root/Drama",
    "/QA Movie and Games Catalog/Storefront Navigation/Movie Store
Root/Drama"
  ],
  "translations": {
    "items": [
      {
        "longDescription": null,
        "displayName": "Drama",
        "description": null,
        "lang": "en"
      }
    ]
  },
  "fixedChildCategories": [
    {
      "displayName": "Award Winners",
      "id": "cat40011"
    },
    {
      "displayName": "Sports",
      "id": "cat40010"
    }
  ],
  "seoDescriptionDerived": "Drama",
  "active": true,
  "categoryIdPaths": [
    "cloudCatalog>cat100058>cat40013>cat40015",
    "cloudLakeCatalog>rootCategory>cat40013>cat40015"
  ],
  "id": "cat40015",
  "fixedChildProducts": []
}
```


The following section describes the fields and validation that occurs when exporting or importing category data.

Categories export and import fields

The following fields are exported and imported for categories:

Field Name	Description
`longDescription`	The description of the category.
`fixedChildProducts`	Any child products associated with the category. The `Id` field is used to identify the fixed child products.
`categoryImages`	Images associated with the category.
`displayName`	The name that is displayed with the category.
`Id`	The ID of the category.

Field Name	Description
`fixedChildCategories`	Any child categories associated with the category. From this field, you can access the `Id` and `displayName` fields for fixed child categories.
`seoDescriptionDerived`	Indicates if the description of the SEO has been derived.
`active`	Indicates if the category is active.
`route`	Non-writable. For export only.
`categoryIdPaths`	Non-writable. For export only.
`categoryPaths`	Non-writable. For export only.

Category validations

Additionally, the following validations are performed when importing categories:

- The required category ID is not null and contains only alphanumeric, underscore (_) or dash (-) characters.
- The required image name and path are valid.
- The products for given product IDs in `fixedChildProducts` are available in the repository.
- The new parent category of child categories is valid.
- The `fixedChildCategory` array contains valid categories.
- The display name is valid.

For detailed information on catalogs and categories, refer to Understand catalogs.

Export and import inventory data

It is common to import inventory data from external systems.

Once you have set up your inventory, as outlined in Understand catalogs, you can import inventory data. The following section describes the fields and validation that occurs when exporting or importing inventory data.

Inventory export and import fields

The following fields are exported and imported for inventories:

Field Name	Description
`locationId`	The location of the inventory item.
`skuNumber`	The ID of the SKU.
`displayName`	The display name of the product.
`stockLevel`	Indicates the number of products available for purchase.
`stockThreshold`	The threshold of the stock level that triggers a warning event.
`preorderLevel`	Indicates the number of preordered SKUs.
`preorderThreshold`	The threshold at which the status of the SKU changes from preordered to out of stock.

Field Name	Description
backorderLevel	Indicates the number of backordered SKUs.
backorderThreshold	The threshold at which the status of the SKU changes from backordered to out of stock.
availabilityDate	The date at which the SKU becomes available.
availabilityStatus	The status of the SKU, for example, PREORDERED, BACKORDERED or OUT_OF_STOCK.
availableToPromise	A collection of date/quality pairs that represent when and how much inventory can be promised.

Inventory validations

Additionally, the following validations are performed when importing inventories:

- The `skuNumber` is a valid SKU ID.
- The `locationId` is valid.
- The required `stockLevel` is valid when creating the import, and should be a non-negative long number.
- The `stockThreshold` is a non-negative long number.
- The `preorderLevel` is a non-negative long number.
- The `preorderThreshold` is a non-negative long number.
- The `backorderLevel` is a non-negative long number.
- The `backorderThreshold` is a non-negative long number.
- The `availabilityDate` is a valid date.

For detailed information on working with inventories, refer to Understand catalogs.

Export and import promotion data

Promotion data can be exported or imported from external systems.

Once you have set up promotions, as outlined in Understand promotions, you can share promotion data. The following section describes the fields and validation that occurs when exporting or importing promotion data.

Promotions export and import fields

The following fields can be exported and imported for promotions:

Field Name	Description
audiences	A list of audiences for which a promotion is specifically designed. This includes the audience ID, display name and if audiences are enabled. You can have multiple audiences.
displayName	The name of the promotion.

Field Name	Description
enabled	Describes if the promotion is enabled. If enabled, the promotion takes effect according to the specified usage period.
endDate	The date that the promotion stops being delivered.
excludedPromotions	Promotions that are excluded from a promotion.
id	The promotion ID.
priceListGroups	The pricelist groups used by the promotion.
priority	The priority of the promotion.
shippingMethods	The shipping method used by the promotion.
sites	A list of sites associated with the promotion. This identifies the site ID and the site name. You can configure multiple sites.
stackingRules	Any stacking rules used by the promotion.
startDate	The date the promotion begins.
templateValues	Placeholder values that are used when creating a template.
type	The type of discount this promotion provides.
description	A description of the promotion.
templateName	The name of the template used with the promotion.
templatePath	The path of the template used with the promotion.
parentFolder	The parent folder of the promotion.
qualifiedMessages	The list of qualified messages for a promotion.
unqualifiedMessages	The list of unqualified messages for a promotion.
closenessQualifiers	The list of closeness qualifiers for a promotion. This data will be imported only if the specific promotion supports it.

Promotion validations

The following validations are performed when creating a new promotions import:

- The `templateName`, `templatePath`, `displayName`, and promotion ID are available and are valid. The shipping method, if provided, should also be valid.
- The `startDate` is not greater than the `endDate`.
- The `promotionTemplates` should be valid and are available in the template path.
- The `Id` field inside `excludedPromotions` cannot be null and that promotions with the ID are valid.
- The `promotionId` cannot be the same as any other `excludedPromotionsId`.
- `TemplateValues` do not contain any keys other than `AllowedUIKeys`, and contain all of the keys listed in `RequireUIKeys`.
- The promotion does not contain any coupon codes.

- The `siteId` and the `audienceId` are valid.
- Included and excluded promotion types are compatible with the promotion:
  - If `templatePath=item`, allowed promotion types are `item`, `order` or `shipping`
  - if `templatePath=order`, then allowed promotion types are `order` or `shipping`
  - if `templatePath=shipping`, then the allowed promotion type is `shipping`

The following validations are performed when updating an existing promotions import:

- The promotion ID is there and valid.
- The `templatePath`, `templateName` and `type` of existing promotions have not been changed.
- If enabling a promotion that does not contain a coupon, ensure that there is no previous coupon associated with it.

For detailed information on working with promotions, refer to Understand promotions.

Export and import price data

With this feature you can import and export the prices for products and SKUs and as part of import, you can create, update and delete the prices.

You can import and export prices for products and SKUs. While importing price data, you can create, update and delete prices. For example, you may want to delete prices so that the price hierarchy can be obtained from the parent. Or you might want to remove a sale price to activate the list price.

The `Prices` plugin allows you to do the following:

- Create and update prices for products and SKUs from different price list groups.
- Provide prices for only the SKUs of a product with price changes.

When you issue a bulk request to import prices (and optionally update, create or delete prices) with the `importOperationCode` property, the following applies:

- You can import and export prices using a JSON or CSV format for both standalone and bundle modes.
- The import process supports both simple and complex pricing schemes.
- The bulk import process supports the `priceListGroupId` and `type` properties, in addition to the `priceListId` property.
- You can configure the bulk price export using the `q` parameter.

The following is an example of a price import process:

```
POST http://localhost:9080/ccadmin/v1/importProcess
{
  "mode": "standalone",
  "fileName": "BulkImportPrices.json",
  "format": "json",
  "uploadType": "bulkImport",
  "id": "Prices"
}
```

The following table lists the fields that are supported by the plugin, along with validations.

Property Name	Type	Description	Required
`importOperationCode`	String	Supported for import only. The following values are supported: `merge` (default): Creates a new record if no matching record is found, otherwise it updates it. `create`: Creates a record. If a matching record is found, an error occurs. Can be used to improve performance with new records. `delete`: Deletes the record.	No
`productId`	string	The product ID.	Yes
`skuId`	string	The SKU ID.	No
`priceListId`	string	The price list ID.	Yes, but it is not required if a `priceListgroupId` and `type` are provided.
`priceListGroupId`	string	The `pricelist` group ID.	No, unless the `pricelistId` property has not been provided.
`type`	string	The price type.	No, unless the `pricelistId` property has not been provided.
`startDate`	date	The start date of the price's availability.	No
`endDate`	date	The date that the price expires.	No
`pricingScheme`	string	The pricing scheme contains the following values: `listPrice`: This indicates a simple price. The `listPrice` property must contain data. `bulkPrice` or `tieredPrice`: These properties indicate a volume price.	Yes
`listPrice`	double	The product/SKU price. This must be provided when the `pricingScheme` property equals the `listPrice` property.	No, unless the `pricingScheme` property is set to `listPrice`.

Property Name	Type	Description	Required
complexPrice	repository object	The volume price information that contains an array of levels.	No, unless the pricingScheme property is set to bulkPrice or tieredPrice.

Understand complex price levels

The complex price entities are defined in the following table:

Property Name	Type	Description
quantity	long	The quantity needed to enable the price.
price	double	The price that is used when the quantity meets the value defined in the quantity property.

Import address data

By using this plugin, you can perform bulk imports of addresses for shopper-based and account-based profiles.

You can integrate your account-based and customer-based data with an external system.

Import address data

The import address plugin allows you to synchronize address data between Commerce and an external system. This plugin allows you to perform the following:

- Import data from an external system in JSON and CSV format in standalone or bundle modes, similar to other bulk plugins.
- Work with custom attributes and address types.
- Delete addresses for an account or profile using the importOperationCode attribute.
- Identify an address as the default shipping or billing address for an profile or account using the isDefaultBillingAddress and isDefaultShippingAddress attributes.

The following is an example of a bulk address import record:

```
{
"contactInfo": [
  {
   "address1": "21 Cedar Ave",
   "city": "Syracuse",
   "country": "US",
   "createdBy": "admin",
   "firstName": "Kim",
   "id": "se-980031",
```

```
    "importOperationCode": "create",
    "isDefaultBillingAddress": true,
    "isDefaultShippingAddress": true,
    "lastName": "Anderson",
    "parentId": "110026",
    "phoneNumber": "212-555-1977",
    "postalCode": "13202",
    "state": "NY",
    "parentType": "contact",
    "addressType": [
      "at100105",
      "at100106"
    ]
  }
]
}
```

Note that:

- Export is not supported.
- To link an address to a account-based contact, the contact must to have the `profileAddressManager` role.

You can link an address to a profile or account by providing the following properties in the data file:

Property Name	Type	Description	Required
`address1`	string	The first address line of the address.	Yes
`address2`	string	The second address line of the address.	No
`address3`	string	The third address line of the address.	No
`addressType`	array	The predefined values for this property can contain address types such as shipping or billing.	For account address types, this property is required. However, for profile address types, if no address type is provided, it defaults to a generic value.
`city`	string	The city of the address.	Yes
`companyName`	string	The company name associated with the address.	No
`country`	string	The country code of the address.	Yes
`county`	string	The county of the address.	No
`createdBy`	string	Creator of the address.	No

Property Name	Type	Description	Required
`externalAddressId`	string	The ID used by the external system to identify the address. Ensure that this field contains a unique `externalAddressId`.	No
`externalParentId`	string	The external ID of the parent address. For contact address types, this ID is provided by the `customerContactId` property, for account address types, it is provided by the `externalOrganizationId` property. Ensure that this field contains valid a `externalOrganizationId` value.	No. However, if the `parentId` property is not provided, an `externalParentId` is required.
`faxNumber`	string	The fax number associated with the address.	No
`firstName`	string	The first name of the profile associated with the address.	No
`id`	string	The ID of the address being imported. Ensure that this field contains a unique `addressId`.	Yes
`importOperationCode`	string	The following values are supported: `merge` (default): Creates a new record if no matching record is found, otherwise it updates the record. `create`: Creates the record. If a matching record is found, an error is generated. Can be used for performance gains for new records. `delete`: Deletes the record.	No
`isDefaultBillingAddress`	boolean	This flag is set to `true` if the address being imported is the default billing address of the parent.	No

Property Name	Type	Description	Required
`isDefaultShippingAddress`	boolean	This flag is set to `true` if the address being imported is the default shipping address of the parent.	No
`middleName`	string	The middle name of the profile associated with the address.	No
`parentId`	string	The ID of the parent address. Ensure that this field contains valid `profileId` and `organizationId` values.	No. However, if the `parentId` property is not provided, an `externalParentId` is required.
`parentType`	string	The type of the parent address. This can be set to either `contact` or `account`, depending on which address is being linked.	Yes
`phoneNumber`	string	The phone number associated with the address.	No
`postalCode`	string	The postal code of address.	Yes
`prefix`	string	The prefix associated with the address.	No
`province`	string	The province code of the address.	No
`state`	string	The state code of the address.	Yes
`suffix`	string	The suffix associated with the address.	No

Import relationship data

This plugin, which is available only in account-based environments, is used for importing the relationships between contacts and accounts.

You can import account and profile relationship data from an external system. Note that only account-based contacts can be linked with profiles.

When you import relationship data using the `Relationships` plugin, you can perform the following:

- Create, update and delete a relationship between an account and a profile using the `importOperationCode` attribute.
- Generate email and webhooks whenever the relationship between a profile and an account changes.

Account and profile relationships are established by providing the following properties:

Property Name	Type	Description	Required
`accountId`	string	The ID of the account organization. Ensure that there is a valid `organizationId` value associated with this account.	Yes, if an `externalAccountId` has not been provided.
`contactId`	string	The ID of the account-based contact. Ensure that there is a valid account-based profile ID associated with this property.	Yes, if an `externalContactId` has not been provided.
`externalAccountId`	string	The external ID of the account, which is referenced by the `externalOrganizationId` property. This property provides the value of the `customerContactId`, which represents the contact ID in an external system.	Yes, if an `accountId` has not been provided.
`externalContactId`	string	The external ID of the contact, which is referenced by the `customerContactId` property.	Yes, if a `contactId` property has not been provided.
`importOperationCode`	string	The following values are supported: `merge` (default): Creates a new record if no matching record is found, otherwise it updates the record. `create`: Creates the record. If a matching record is found, an error is generated. Can be used for performance gains for new records. `delete`: Deletes the record.	No

Property Name	Type	Description	Required
`primary`	boolean	Denotes whether the account is the primary or secondary organization of the contact. When primary is set to `true`, the account is added as the parent organization of the contact. When set to `false`, the account is set as a secondary organization. When a primary value is not provided and the relationship already exists (primary or secondary), it is not updated. If there is no relationship, the account will be added as a secondary organization. If you do not provide this property for an existing relationship, the existing relationship remains.	No

Export and import CSV files

When importing and exporting data, you may want to use CSV files. This allows you to list the data in multiple lines with headers.

The first row of the CSV contains the headers. These headers are used to identify the complete path to the property, since the CSV format supports complete hierarchy of the record.

Understand CSV headers

When working with CSV headers, remember the following:

1. The top-level properties use the property name as the header name. For example, the property and header name would both be `stringProperty1`.
2. A child item uses dot separators to refer to its properties. For example, the property of a child item would be `complexProperty.stringProperty1`.
3. When working with list properties, the first header holds the sequence number of each item followed by the individual property. For example:

```
listProperty.row#, listProperty.springProperty1,
listProperty.numberProperty1
```

4. Map properties use the first header to hold the keys of each entry, followed by individual properties. For example:

   ```
   mapProperty.key#, mapProperty.stringProperty1,
   mapProperty.numberProperty1
   ```

   Note that each entry in a map or list will be displayed in separate CSV rows. This may make a single record in a CSV file spread across multiple CSV rows.

The following is an example of a list data CSV file with records spreading across multiple rows:

ID	listProperty.row#	listProperty.id	listProperty.property1	listProperty.property2
ID123	0	childpropId1	Property1data1	Property2data1
ID123	1	childpropId2	Property1data2	Property2data2
ID123	2	childpropId3	Property1data3	Property2data3

You can export a CSV file with default headers using the `csv` in the `format` parameter. For example:

```
{
  "fileName": "profile.json",
  "mode" : "standalone",
  "id" : "Profiles",
  "format" : "csv"
}
```

You can export all attributes, including custom attributes by setting the `headersList` parameter to `ALL`. The following example shows how to include all headers in a standalone mode:

```
{
  "fileName": "profile.json",
  "mode" : "standalone",
  "id" : "Profiles",
  "format" : "csv"{
   "fileName": "profile.json",
   "mode" : "standalone",
   "id" : " Profiles",
   "format" : "csv",
   "params": {
          "headersList": "ALL"
    }
 }
}
```

To export a specific list of attributes, use the following format:

```
{
   "fileName": "profile.json",
   "mode" : "standalone",
   "id" : " Profiles",
   "format" : "csv",
   "params": {
          "headersList":
"firstName,lastName,shippingAddress.postalCode,shippingAddress.country"
    }
 }
```

You can also use the default list of attributes and include additional attributes by using a '+' sign before the list of additional headers. For example:

```
{
   "fileName": "profile.json",
   "mode" : "standalone",
   "id" : " Profiles",
   "format" : "csv",
   "params": {
          "headersList": "+addedfield1,addedfield2,addedfield3.subField1"
    }
 }
```

If you do not specify a set of headers, the export includes the default set of headers for the Oracle Commerce item.

Note: Custom properties for collections are supported for both CSV and JSON formats.

1. For JSON export, the custom properties are automatically included.
2. For CSV export, the `headersList` should include the required custom property names or they can pass "ALL" (as in the previous example) to get all the custom properties.
3. During import, the custom properties can be passed in both CSV and JSON format.

Understand the JSON format

A data file in the JSON format contains an array of JSON items, with each JSON item considered to be a data record. The following is an example of a JSON format:

```
{
  "products": [
    {
      "displayName": "Product 1 Name",
      "id": "product1",
      "listPrice": 35,
      "childSKUs": [
        {
          "id": "SKU of Product1"
        }
      ]
    },
```

```
    {
      "displayName": "Product 2 Name",
      "id": "product2",
      "listPrice": 45,
      "childSKUs": [
        {
          "id": "SKU of Product2"
        }
      ]
    }
  ]
}
```

When working with JSON files, remember the following:

1. A single JSON record can contain the complete hierarchy of data. For example, a profile can have a list of addresses, accounts and create cards, etc.
2. The record that needs to be updated can include any updated partial data except for maps and lists. If simple properties, such as strings or numbers, are not passed during the update, they will not be reset to `null` or to `0`.
3. For map and list properties, if an existing entry is not passed during an update, it is deleted. For example, if a profile contains three addresses and only the first two addresses are passed in with the data file, the third address will be deleted.
4. A simple property value can be set to null so that during an update, it removes the previous value. For example:

   ```
   {
     ...
     "property1" : null
     ...
   }
   ```

5. A list property can be cleared in JSON format by passing an empty array. For example:

   ```
   {
     ...
     "listProperty1" : []
     ...
   }
   ```

Default headers for CSV export and import

The following section describes the fields that are included in the header by default when exporting or importing data.

Default headers for accounts

The default header list for accounts contains:

- id
- name
- description

- type
- customerType
- taxReferenceNumber
- dunsNumber
- contract.terms.terms
- contract.displayName
- contract.description
- contract.catalog.id
- contract.priceListGroup.id
- contract.externalContractReference
- members.row#
- members.firstName
- members.lastName
- members.id
- members.email
- relativeRoles.row#
- relativeRoles.function
- secondaryAddresses.key#
- secondaryAddresses.country
- secondaryAddresses.phoneNumber
- secondaryAddresses.address2
- secondaryAddresses.city
- secondaryAddresses.address1
- secondaryAddresses.companyName
- secondaryAddresses.postalCode
- secondaryAddresses.id
- secondaryAddresses.state
- billingAddress.country
- billingAddress.phoneNumber
- billingAddress.address2
- billingAddress.city
- billingAddress.address1
- billingAddress.companyName
- billingAddress.postalCode
- billingAddress.id
- billingAddress.state
- shippingAddress.country

- shippingAddress.phoneNumber
- shippingAddress.address2
- shippingAddress.city
- shippingAddress.address1
- shippingAddress.companyName
- shippingAddress.postalCode
- shippingAddress.id
- shippingAddress.state

Default headers for profiles

The default header list for profiles contains:

- dateOfBirth
- middleName
- receiveEmail
- lastName
- locale
- id
- lastActivity
- registrationDate
- email
- login
- firstName
- shippingAddress.middleName
- shippingAddress.item-id
- shippingAddress.lastName
- shippingAddress.state
- shippingAddress.address1
- shippingAddress.address2
- shippingAddress.address3
- shippingAddress.companyName
- shippingAddress.repositoryId
- shippingAddress.suffix
- shippingAddress.city
- shippingAddress.country
- shippingAddress.postalCode
- shippingAddress.faxNumber
- shippingAddress.phoneNumber
- shippingAddress.county

- shippingAddress.prefix
- shippingAddress.firstName

Default headers for products

The default header list for products contains:

- id
- displayName
- description
- longDescription
- type
- keywords.row#
- keywords.keywords
- listPrices.key#
- listPrices.listPrices
- salePrices.key#
- salePrices.salePrices
- shippingSurcharges.key#
- shippingSurcharges.shippingSurcharges
- listVolumePrices.key#
- listVolumePrices.complexPrice.levels.row#
- listVolumePrices.complexPrice.levels.quantity
- listVolumePrices.complexPrice.levels.price
- listVolumePrices.pricingScheme
- saleVolumePrices.key#
- saleVolumePrices.complexPrice.levels.row#
- saleVolumePrices.complexPrice.levels.quantity
- saleVolumePrices.complexPrice.levels.price
- saleVolumePrices.pricingScheme
- parentCategories.row#
- parentCategories.displayName
- parentCategories.id
- productImages.row#
- productImages.description
- productImages.url
- productImages.id
- productImages.path
- productImages.type
- productImages.name

- primaryImageTitle
- smallImageURLs
- primaryLargeImageURL
- fullImageURLs
- sourceImageURLs
- primarySourceImageURL
- mediumImageURLs
- largeImageURLs
- thumbImageURLs
- primaryMediumImageURL
- primaryImageAltText
- primarySmallImageURL
- primaryFullImageURL
- primaryThumbImageURL
- seoUrlSlugDerived
- seoKeywordsDerived
- seoDescriptionDerived
- seoTitleDerived
- brand
- defaultProductListingSku.id
- childSKUs.row#
- childSKUs.id
- childSKUs.barcode
- childSKUs.active
- childSKUs.bundleLinks.row#
- childSKUs.bundleLinks.item.id
- childSKUs.bundleLinks.quantity
- childSKUs.listPrices.key#
- childSKUs.listPrices.listPrices
- childSKUs.salePrices.key#
- childSKUs.salePrices.salePrices
- childSKUs.listVolumePrices.key#
- childSKUs.listVolumePrices.complexPrice.levels.row#
- childSKUs.listVolumePrices.complexPrice.levels.quantity
- childSKUs.listVolumePrices.complexPrice.levels.price
- childSKUs.listVolumePrices.pricingScheme
- childSKUs.saleVolumePrices.key#

- childSKUs.saleVolumePrices.complexPrice.levels.row#
- childSKUs.saleVolumePrices.complexPrice.levels.quantity
- childSKUs.saleVolumePrices.complexPrice.levels.price
- childSKUs.saleVolumePrices.pricingScheme

Default headers for categories

The default header list for categories contains:

- longDescription
- categoryImages.row#
- categoryImages.description
- categoryImages.url
- categoryImages.id
- categoryImages.path
- categoryImages.type
- categoryImages.name
- displayName
- id
- fixedChildProducts.row#
- fixedChildProducts.id
- seoDescriptionDerived
- fixedChildCategories.row#
- fixedChildCategories.id
- fixedChildCategories.displayName

Default headers for inventory

The default header list for inventory contains:

- preOrderLevel
- backorderThresdhold
- displayName
- skuNumber
- backorderLevel
- availabilityStatus
- availableToPromise.quantity
- availableToPromise.availableDate
- availableToPromise.quantityWithFraction
- availableToPromise.inventoryId
- preorderThreshold
- availabilityDate

- locationId
- stocklevel
- stockThreshold

Default headers for promotions

The default header list for promotions contains:

- template
- templateName
- templatePath
- endDate
- displayName
- sites.row#
- sites.id
- sites.name
- audiences.row#
- audiences.id
- audiences.displayName
- audiences.deleted
- audiences.enabled
- global
- templateValues.no_of_items_to_discount
- templateValues.discount_value
- templateValues.discount_type_value
- templateValues.sort_order
- templateValues.no_of_items_to_buy
- templateValues.spend_value
- templateValues.sort_by
- templateValues.discountStructure
- templateValues.gwpItem.autoRemove
- templateValues.gwpItem.giftType
- templateValues.gwpItem.giftId
- templateValues.condition_psc_value.includedCategories.row#
- templateValues.condition_psc_value.includedCategories.includedCategories
- templateValues.condition_psc_value.includedProducts.row#
- templateValues.condition_psc_value.includedProducts.includedProducts
- templateValues.condition_psc_value.excludedProducts.row#
- templateValues.condition_psc_value.excludedProducts.excludedProducts
- templateValues.condition_psc_value.excludedCategories.row#

- templateValues.condition_psc_value.excludedCategories.excludedCategories
- templateValues.condition_psc_value.includedSkus.row#
- templateValues.condition_psc_value.includedSkus.includedSkus
- templateValues.condition_psc_value.excludedSkus.row#
- templateValues.condition_psc_value.excludedSkus.excludedSkus
- templateValues.condition_psc_value.sameAsCondition
- templateValues.optional_offer_psc_value.includedCategories.row#
- templateValues.optional_offer_psc_value.includedCategories.includedCategories
- templateValues.optional_offer_psc_value.includedProducts.row#
- templateValues.optional_offer_psc_value.includedProducts.includedProducts
- templateValues.optional_offer_psc_value.excludedProducts.row#
- templateValues.optional_offer_psc_value.excludedProducts.excludedProducts
- templateValues.optional_offer_psc_value.excludedCategories.row#
- templateValues.optional_offer_psc_value.excludedCategories.excludedCategories
- templateValues.optional_offer_psc_value.includedSkus.row#
- templateValues.optional_offer_psc_value.includedSkus.includedSkus
- templateValues.optional_offer_psc_value.excludedSkus.row#
- templateValues.optional_offer_psc_value.excludedSkus.excludedSkus
- templateValues.optional_offer_psc_value.sameAsCondition
- templateValues.offer_psc_value.includedCategories.row#
- templateValues.offer_psc_value.includedCategories.includedCategories
- templateValues.offer_psc_value.includedProducts.row#
- templateValues.offer_psc_value.includedProducts.includedProducts
- templateValues.offer_psc_value.excludedProducts.row#
- templateValues.offer_psc_value.excludedProducts.excludedProducts
- templateValues.offer_psc_value.excludedCategories.row#
- templateValues.offer_psc_value.excludedCategories.excludedCategories
- templateValues.offer_psc_value.includedSkus.row#
- templateValues.offer_psc_value.includedSkus.includedSkus
- templateValues.offer_psc_value.excludedSkus.row#
- templateValues.offer_psc_value.excludedSkus.excludedSkus
- templateValues.offer_psc_value.sameAsCondition
- templateValues.PSC_value.includedCategories.row#
- templateValues.PSC_value.includedCategories.includedCategories
- templateValues.PSC_value.includedProducts.row#
- templateValues.PSC_value.includedProducts.includedProducts
- templateValues.PSC_value.excludedProducts.row#

- templateValues.PSC_value.excludedProducts.excludedProducts
- templateValues.PSC_value.excludedCategories.row#
- templateValues.PSC_value.excludedCategories.excludedCategories
- templateValues.PSC_value.includedSkus.row#
- templateValues.PSC_value.includedSkus.includedSkus
- templateValues.PSC_value.excludedSkus.row#
- templateValues.PSC_value.excludedSkus.excludedSkus
- description
- priority
- type
- excludedPromotions.row#
- excludedPromotions.id
- priceListGroups.row#
- priceListGroups.id
- translations.items.row#
- translations.items.description
- translations.items.displayName
- translations.items.lang
- id
- startDate
- shippingMethods.row#
- shippingMethods.shippingMethods
- stackingRule
- parentFolder.row#
- parentFolder.id

Delete bulk import or export files from repository

Each time a bulk import or export is performed, files are generated and are saved in the publishing queue. These files are not needed by Storefront and the total files size can eventually grow in size to become detrimental to remain in the system.

If the files for bulk import are uploaded using the `files` endpoint by passing the `{"uploadType" : "bulkImport"}` parameter, the files will not be added to the publishing queue and they will be automatically deleted. It is highly recommended that you use this parameter for uploading the bulk import files.

If the files are uploaded without the `{"uploadType" : "bulkImport"}` parameter, they are saved in the publishing queue. These files are not needed by the Storefront and the total file sizes can eventually grow very large and become detrimental to your system.

To delete these files, contact your Oracle representative to open a Service Request (SR) to being the process of deleting these files. Once your Oracle representative has finished configuration changes, you can delete the files uploaded for import, generated by bulk import, or exported from repository using the `deleteFiles` endpoint, for example:

```
POST /ccadminui/v1/files/deleteFiles
Input Example:
{
   "deletePaths": [
      "/import/importPath1",
      "/import/importPath2"
   ],
   "recursive": "true"
}
```

Note: It is recommended that you delete the files before publishing, so that they are not published. However, you can also delete published files. In this case, these the deleted files will appear in Publishing queue with "DELETE" in red text, and they need to be published. If you are having trouble finding your uploaded files, you may not have imported them to the `/import` folder. When importing files include the folder name in the filename property in the payload, for the appropriate destination, for example; filename: `/import/<filename.csv>`

Convert registered shoppers to account-based shoppers

You can perform a one-time bulk conversion of registered shoppers to account-based shoppers.

This section details this process.

To convert a set of registered shoppers to account-based shoppers follow these steps:

1. Export registered shoppers using bulk export API to a CSV or JSON format. See the section Perform Bulk Export and Import for more information.
2. Open the exported file and change the `profileType` from registered `(b2c_user` or null) to account-based (`b2b_user`).
3. Associate each shopper with an active account.
4. Change other fields as desired - for example, adding roles, or removing profile addresses.
5. Import the changed file (CSV or JSON) using bulk import API with the input parameter `isProfileMigration` set to `true`. See the section Perform Bulk Export and Import for more information.

Only the administrator can access the export and import APIs.

Notes on the conversion process

The following table notes issues to be aware of when you convert a shopper's type from registered to account-based.

Conversion area	Conversion Notes
`Profiletype`	After conversion, this field's value will reflect that the shopper is now account-based (`b2b_user`).

Conversion area	Conversion Notes
Lifetime value properties The following properties exist for Registered shoppers only: • Lifetime spend • Lifetime average order value (AOV) • Number of orders • Last purchase amount • First purchase date • Last purchase date	When you convert a registered shopper to an account-based shopper, the shopper's lifetime value properties are changed. The following properties are set to 0: • Lifetime spend • Lifetime average order value (AOV) • Number of orders • Last purchase amount The following properties are set to null: • First purchase date • Last purchase date If you have set up any kind of custom reporting based directly on these properties, reports that span the timeframe of the migration may reflect the fact that these properties were zeroed/nulled for the converted shoppers.
GDPR - Right to be forgotten: • Deletion of profile • Redaction of PII from many entities	Under GDPR (General Data Protection Regulation), a registered shopper or an account-based shopper may ask to be forgotten. You have a legally specified amount of time to redact the shopper's PII in the system and delete the shopper's profile. You determine exactly what PII needs to be redacted, based on guidance from your own legal counsel. For account-based shoppers, there is potentially PII in more places than for registered shoppers. For more information, see the section Delete shopper information.
Account-based roles such as account-based Buyer, Administrator, Approver, Account Address Manager, and Profile Address Manager roles. A registered shopper cannot have these roles.	During conversion, you can assign converted shoppers to account-based roles.
Profile addresses - including: • Shipping address • Billing address	You can delete shopper profile addresses during the conversion process by nulling the address columns in the spreadsheet of exported profiles before re-importing the profiles. If the profile addresses are left intact, the converted shopper will be able to check out with those addresses even if the merchant did not give the shopper (or any shopper) the Profile Address Manager role and does not want account-based shoppers using personal addresses.

Conversion area	Conversion Notes
ORDERS - Active orders - including: • Orders pending payment • Quoted orders • Incomplete orders (carts)	Converted shoppers will not be able to see any orders that they submitted or carts they created when they were a registered shopper. An Agent will not be able to see a cart that the shopper had created as a registered shopper An order that the shopper had submitted as a registered shopper and that does not need intervention will continue to process. If an order that the shopper had submitted as a registered shopper is payment pending or has been quoted, an Agent will be able to see and cancel the order, but cannot pay and submit it. The shopper's orders can be accessed via an Admin API.
ORDERS: Fulfilled orders and order history Returns, exchanges, cancellations	Shoppers will not be able to see any orders that they submitted as a registered shopper. If an order that the shopper had submitted as a registered shopper has been fulfilled, an Agent will be able to see the order and process it as usual – for example, process a return. In addition, you can access the orders using an Admin API.
ORDERS: Scheduled orders	After conversion, any scheduled orders that the shopper had submitted as a registered shopper will be deactivated.
ORDERS: Abandoned cart fields and emails	The shopper will no longer receive abandoned order emails for their registered shopper carts, since they can no longer access their registered shopper carts.
Active coupons / promotions	There is a unusual situation that could occur during the conversion: • A registered shopper enters coupon code, but does not check out. • The shopper is converted to an account-based shopper. • The promotion remains granted to the shopper and stored in the profile. • The converted shopper qualifies for promotion, it is applied, and the shopper checks out having used a registered shopper-intended promotion for account-based shopping. You should be aware of this potential issue and manage it using business processes. One possible solution: For each promotion, specify the price groups to which the promotion applies. A shopper who is converted from registered to account-based will switch price groups, and will only qualify for promotions associated with account-based price groups.

Conversion area	Conversion Notes
Purchase lists - including: • Lists the shopper created and kept private • Lists the shopper created and shared • Lists shared with the shopper	Purchase lists that a converted shopper created when they were a registered shopper and shared with other registered shoppers will no longer be shared upon conversion. After conversion, the only purchase lists shared with the shopper will be those that are shared with the whole account, for any account to which the shopper is being added.
Audience membership	Audiences based on lifetime value properties will stop capturing shoppers who were converted, because the conversion process will have zeroed/nulled these shoppers' lifetime value properties, since these properties are not populated for account-based shoppers. Audiences based on the following profile address properties will no longer capture any shoppers who were converted, if you chose to null out these profile properties as part of the conversion: • Billing address city, country, postal code, state • Shipping address city, country, postal code, state Audiences based on the following profile properties will not be affected: • Date of birth • Email address • First visit date • Gender • Previous visit date • Receive email • Registration date Newly converted shoppers may begin to show up in audiences that are based on account properties, since these shoppers will now belong to accounts.
SSO	If the registered shopper had been signing in via single sign-on (SSO), then after conversion, they will only be able to log in if they are a member of an active account.
Active flag	You can change this field during conversion. An account-based shopper's active status applies across all accounts to which they belong. An account-based shopper must be active in order to be able to log in.

Conversion area	Conversion Notes
Administrator access to shopper profile data	When you convert a shopper from registered to account-based, the shopper will be somewhat more exposed to inspection by administrators. A registered shopper's profile properties are visible only to the shopper. Once converted into account-based shopper, the account administrator can now view attributes of the shopper profile. If you want to prevent Administrators and Delegated Administrators from accessing a shopper profile property, you would need to use role-based property-level access. Property-level access control can ensure that a given profile property's values can be viewed only by the shopper, not by an Administrator or Delegated Administrator. You can also use existing endpoints to clear the values of specific profile properties once exported, and import them with null values. Administrator access to profile properties is as follows: • Merchant Administrators can use the `updateProfile` API to update a shopper's profile properties (custom and static). This applies to all properties (including profile addresses) except `profileType`. • Merchant Administrators have a UI for viewing a shopper's name, email address, and custom profile property values. • Custom and static profile properties that are not `internalOnly` are exposed in the Storefront view model. • A Delegated Administrator has a UI for viewing the name and email address of a shopper in his account. • A Delegated Administrator can use the `updateMember` API to update the profile properties of a shopper in his account. This applies to all profile properties except for the shopper's account memberships and `profileType`.
Shopper's access to sites	Note that, as an account-based shopper, the shopper will only be able to access sites that have contracts with the account.
Catalogs and pricelists	Note that, as an account-based shopper, the shopper should see catalogs and pricelists based on the account's contract with the site.
Webhooks	Changes to shoppers' profile properties during conversion may result in a high volume of webhook calls upon re-import. You can use the existing attribute `triggerWebHook` to control whether webhooks are triggered on re-import.

Conversion area	Conversion Notes
Emails to the shoppers	After the conversion, the usual Account Assignment emails are sent to converted shoppers who are active and assigned to active accounts. Other emails may be triggered as well. The triggering of each email template can be controlled using an existing parameter on import.

Improve performance in large bulk imports

The performance of the import of very large numbers of products can be improved with the use of an alternate plug-in.

If you are trying to import very large numbers of products, you can improve performance by using an alternate plug-in, ProductsV2. Oracle recommends using ProductsV2 during your initial load of data. You can choose between the Products and ProductsV2 plug-ins during the import process. The ProductsV2 plug-in takes advantage of multi-threaded processing. The major difference between Products and ProductsV2 is the file `failedAssociationsRecordsFile`, which appears in an import response when the ProductsV2 plug-in is used as the import ID. For Import Status, `SuccessCount` and `Failure Count` do not depend on `Category` or `Add on` product failures. As a result, if `Category` or `Add On` products are invalid in the passed import data, the products will still be created. After your import, even if the success count matches the product count passed, you should always check `failedAssociationsRecordsFile` for any association failures. If you find failure records, correct `failedAssociationsRecordsFile` and re-import.

How to use the ProductsV2 plug-in

The ProductsV2 plug-in can be used in the same way as the Products plug-in. For more information, see Perform Bulk Export and Import. The following examples show how to use the ProductsV2 plug-in in Standalone and Bundle mode.

Standalone mode code sample

```
POST http://localhost:9080/ccadminui/v1/importProcess
Standalone mode Request:
{
    "mode": "standalone",
    "fileName": "<filename>",
    "format": "json",
    "id": "ProductsV2"
}
```

Note: For CSV request formats, specify `"csv"` as the `"format"` value.

Bundle mode code sample In bundle mode, place files to be imported in zip format and place the content.json file inside the zip which describes importing files details. For example: `[{ "fileName": "example.csv", "format": "csv", "id":`

```
"ProductsV2" },{ "fileName": "example2.csv", "format": "csv", "id":
"ProductsV2" }]

POST http://localhost:9080/ccadminui/v1/importProcess
Bundle mode Request:
{
  "mode": "bundle",
  "fileName": "<filename.zip>"
}
```

Note: For CSV request formats, specify `"csv"` as the `"format"` value.

Sample response

```
{
  "completedPercentage": 100,
  "progress": "succeeded",
  "startTime": "2019-09-24T18:31:45.214Z",
  "links": [
    {
      "rel": "meta",
      "href": "http://example.com:3021/file/v5804828446398654872/import/
cEIWX7B43XJVrfwZm7ygfK4nr8_10001/importStatus_ProductsV2.json"
    },
    {
      "rel": "failedRecordsFile",
      "href": "http://example.com:3021/file/v3721096615427138463/import/
cEIWX7B43XJVrfwZm7ygfK4nr8_10001/importFile.csv"
    },
    {
      "rel": "failedAssociationRecordsFile",
      "href": "http://example.com:3021/file/v3721096615427138463/import/
cEIWX7B43XJVrfwZm7ygfK4nr8_10001/importFile_failedAssociations.csv"
    },
    {
      "rel": "self",
      "href": "http://example.com:3021/ccadminui/v1/importProcess/
cEIWX7B43XJVrfwZm7ygfK4nr8_10001?fileName=Bulk_Import100K.zip"
    }
  ],
  "endTime": "2019-09-24T18:35:39.725Z",
  "completed": true,
  "requestStatus": 200
}
```

37

Create a Credit Card Payment Gateway Integration

You can use tools that Commerce provides to create custom integrations with payment gateways.

As discussed in Configure Payment Processing in Using Oracle Commerce, Oracle Commerce provides support for a number of payment gateways as built-in integrations. In addition, you can create custom integrations with other payment gateways. The integrations you create appear as options on the **Payment Gateways** tab of the **Payment Processing** page in the administration interface.

To create a custom integration with a credit card payment gateway, you create an extension for accessing the gateway, and configure the Credit Card Payment function webhook. When a shopper places an order, the webhook calls a specified payment service URL and sends the payment-related data in a JSON request. The external system then sends a response that indicates success or failure and other information about the transaction.

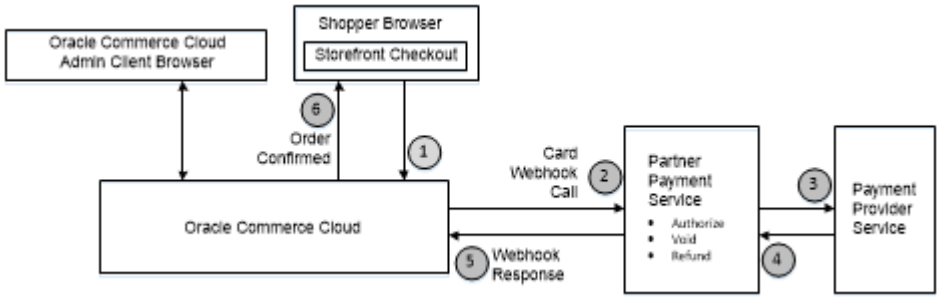
Determine which payment webhook to use

In addition to the Credit Card Payment webhook, which supports only credit cards, Commerce includes the Generic Payment webhook, which supports a variety of different payment methods, including credit cards. See Supported payment methods and transaction types for information about the other payment methods available.

Note that the Credit Card Payment webhook does not support the 3D-Secure protocol, and does not support the use of stored credit cards. To create a credit card payment gateway that supports 3D-Secure or the use of stored credit cards, you must use the Generic Payment webhook. See the Create a Generic Payment Gateway Integration chapter for more information.

Understand the credit card payment gateway workflow

The following diagram illustrates the credit card payment gateway workflow:



Create a credit card extension

Extensions let you add functionality to your store or settings to your administration interface.

For example, you can use extensions to upload custom widgets and widget elements to your store, or to add settings for a custom payment gateway to the **Payment Processing** page's **Payment Gateways** tab.

Before you develop an extension, you must generate an ID that you will include in your extension file. After you develop the extension, you install it by uploading it to Oracle Commerce as a ZIP file. For a payment gateway, the directory structure of the ZIP file looks like this:

```
<extension-name>
   ext.json
   gateway/
      <gateway ID>/
         gateway.json
         config/
            config.json
            locales/
               <locale>.json
```

The JSON files in this structure are used to set various properties that configure the behavior of the extension. These files are discussed below, using a sample credit card payment gateway extension to illustrate their contents. You can find additional information about extensions in Create an Extension.

`ext.json`

The `ext.json` file contains metadata for the extension. For example:

```
{
  "extensionID": "c2e6a60e-579a-4190-af3e-5edc0cd8a725",
  "developerID": "999999",
  "createdBy": "Demo Corp.",
  "name": "DemoPaymentGateway",
  "version": 1,
  "timeCreated": "2016-01-01",
  "description": "Demo Payment Gateway"
}
```

Note that the extension ID must match the value generated on the Extensions page in the administration interface. See Install the extension and configure the gateway for more information.

`gateway.json`

The `gateway.json` file configures the following gateway settings:

- `provider` – A label describing the payment provider.
- `paymentMethodTypes` -- A list of the payment method types supported for the gateway.
- `transactionTypes` -- A list of supported transaction types for each supported payment type. For a credit card payment gateway, valid values are `authorization`, `void`, and `refund`.

In the following example, the Demo Payment Provider is configured to permit only credit card transactions, and to support `authorization`, `void`, and `refund` transactions:

```
{
  "provider": "Demo Payment Provider",
  "paymentMethodTypes": ["card"],
  "transactionTypes": {
    "card": ["authorization", "void", "refund"]
  }
}
```

config.json

The `config.json` file creates user interface elements in the administration interface for configuring gateway settings. For example:

```
{
  "configType": "payment",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "instances" : [
    {
      "id": "agent",
      "instanceName": "agent",
      "labelResourceId": "agentInstanceLabel"
    },
    {
      "id": "preview",
      "instanceName": "preview",
      "labelResourceId": "previewInstanceLabel"
    },
    {
      "id": "storefront",
      "instanceName": "storefront",
      "labelResourceId": "storefrontInstanceLabel"
    }
  ],
  "properties": [
    {
      "id": "merchantId",
      "type": "stringType",
      "name": "merchantId",
      "helpTextResourceId": "merchantIdHelpText",
      "labelResourceId": "merchantIdLabel",
      "defaultValue": "merchant id",
      "required": true
    },
    {
      "id": "paymentMethodTypes",
      "type": "multiSelectOptionType",
      "name": "paymentMethodTypes",
      "required": true,
      "helpTextResourceId": "paymentMethodsHelpText",
      "labelResourceId": "paymentMethodsLabel",
```

```
      "defaultValue": "card",
      "options": [
         {
           "id": "card",
           "value": "card",
           "labelResourceId": "cardLabel"
         }
       ]
    },
    {
      "id": "includeOrderInWebhookPayload",
      "type": "booleanType",
      "name": "includeOrderInWebhookPayload",
      "helpTextResourceId": "includeOrderHelpText",
      "labelResourceId": "includeOrderLabel",
      "defaultValue": true,
      "public": true
    }
  ]
}
```

Notice the following settings in the example above:

- The `configType` property specifies the type of configuration the file contains. For a payment gateway, the value of this property should be `payment`.
- The `instances` property specifies an array of different instances of the resource being configured, which makes it possible to have multiple groups of the same settings in the administration interface. In the example above, there are separate settings created for the storefront, the Agent Console, and preview.
- The `includeOrderInWebhookPayload` property creates a checkbox for specifying whether or not to include the order data in the webhook call.
- The file specifies a number of resource properties. The labels used in the user interface are mapped to the resource IDs in the *`<locale>`*`.json` files, as described below.

***<locale>*.json**

You create a separate *`<locale>`*`.json` file for each language supported in your administration interface. For example, you might have an `en.json` file for English, `fr.json` for French, and `de.json` for German. These files contain labels that appear in the administration interface when you select the Payment Gateways tab. For example:

```
{
  "resources": {
    "paymentMethodsLabel": "Payment Methods",
    "merchantIdLabel": "Merchant ID",
    "cardLabel": "Credit/Debit Card",
    "title": "Demo Payment Gateway Config",
    "description":"Demo Payment Gateway configuration",
    "agentInstanceLabel": "Agent Configuration",
    "previewInstanceLabel": "Preview Configuration",
    "storefrontInstanceLabel": "Storefront Configuration"
```

```
    "merchantIdHelpText": "Enter your Merchant ID",
    "paymentMethodsHelpText": "Select your payment method"
    "includeOrderLabel": "Include order data in webhook call?"
    }
}
```

The values of the properties in the file are applied to the corresponding resource ID in the `config.json` file. For example, the value of the `paymentMethodsLabel` property is used to set the value of the `labelResourceID` property of the JSON object in `config.json` that specifies the user interface controls.

Install the extension and configure the gateway

Once you create the extension, you need to install it and configure the payment gateway.

Install the extension

To install the extension, do the following in the administration interface:

1. Click the **Settings** icon.
2. Click **Extensions** and display the **Developer** tab.
3. Click **Generate ID**. In the dialog, fill in the extension name and click **Save**. A new extension ID is created.
4. Set the `extensionID` property in the `ext.json` file to the value of the ID.
5. Package the extension in a ZIP file. (See Create a credit card extension.)
6. In the **Installed** tab, click **Upload Extension**. Select the ZIP file.

Once the extension is uploaded, it appears in the list of installed extensions.

Enable the gateway

To enable the new payment gateway:

1. Click the **Settings** icon.
2. Select the site you want to configure the gateway for.
3. Click **Payment Processing** and display the **Payment Gateways** tab.
4. Select the payment gateway integration you installed from the **Service Type** drop-down list.
5. Select the **Payment Gateway Enabled** checkbox.
6. Configure any other settings required by the integration. For example, in the extension shown in Create a credit card extension, there are checkboxes for enabling credit card support under Preview Configuration, Agent Configuration, and Storefront Configuration.

When you enable a custom credit card payment gateway for a site, make sure that the other credit card gateways (for example, CyberSource and Chase Paymentech Credit Cards) are disabled on that site. Only one credit card gateway integration should be enabled for an individual site.

Configure the Credit Card Payment webhook

When you create an integration for a credit card gateway, the integration uses the Credit Card Payment webhook to send authorization requests to the gateway. To configure the webhook:

1. Click the **Settings** icon.
2. Click **Web APIs** and display the **Webhook** tab.
3. Select the **Credit Card Payment** webhook that you want to configure. Note that there are separate Preview and Production versions of the webhook.
4. In the **URL** field, enter the URL for accessing the payment gateway. The URL must use HTTPS.
5. Under **Basic Authorization**, fill in the username and password for accessing your gateway account.
6. If your gateway requires any additional HTTP request headers, click **Add New Header Property** and fill in the property name and value.
7. Click **Save**.

Note that webhook settings are not site-specific. The configuration you supply applies to all sites that use this webhook.

Credit card payment properties

When the Credit Card Payment webhook executes, it sends a JSON request body to the payment gateway.

The request body contains a request that contains information about the order and about the method of payment. The gateway processes the request and returns a JSON response body that contains information about the transaction, including whether the transaction succeeded.

The set of properties in the request and response bodies, including the subobjects, vary depending on the type of transaction. For credit card gateways, there are three transaction types supported: `authorization`, `void`, and `refund`.

Credit card payment request properties

This section describes the top-level properties and the properties of subobjects sent in the JSON request body of the Credit Card Payment webhook. Note that if the `includeOrderInWebhookPayload` property in the gateway extension's `config.json` file is set to `true`, the order is also included in the request. See Order Submit webhook for information about the order properties.

Top-level properties

The following table describes the top-level properties that Oracle Commerce sends in the webhook request.

Property	Description
paymentId	The ID of the internal payment group
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens.

Property	Description

transactionType	A code indicating the type of transaction. For the Credit Card Payment webhook, this must be one of the following numeric values:
	
	`0100 (authorization)`
	`0110 (void)`
	`0400 (refund)`

transactionTimestamp	The timestamp of the transaction, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

channel	The area of the system where the payment-processing request originated. Valid values are:
	
	`storefront`
	`agent`
	`preview`

paymentMethod	The payment method. For the Credit Card Payment webhook, the value must be `card`.
orderId	The ID of the order associated with the payment
amount	The amount to be authorized, as a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
currencyCode	The ISO 4217 currency code
locale	The shopper's locale, taken from the order. If no locale is set, the default locale from the storefront is used.
siteURL	The URL of the site on which the order was placed
siteId	The ID of the site on which the order was placed
gatewayId	The ID of the payment gateway
retryPaymentCount	The number of times payment has been retried for the order

`auxiliaryProperties`

The following table describes the properties of the `auxiliaryProperties` object in the request.

Property	Description
authenticationMethod	Either `guest` (for an anonymous shopper) or `local` (for a logged-in shopper)

Property	Description
shopperAccountPaymentAccountFirstUseDate	The timestamp of when the card used for payment was saved, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

`cardDetails` properties

The following table describes the properties of the `cardDetails` object in a credit card payment request. The values of these properties are used to authorize the payment.

Property	Description
expirationMonth	A two-digit number indicating the month the credit card expires (for example, 07 for July)
expirationYear	A four-digit number indicating the year the credit card expires (for example, 2019)
cvv	The three-digit or four-digit security code verifying the credit card
number	The credit card number

type	The credit card type. Valid values are:
	
	`visa`
	`mastercard`
	`amex`
	`discover`
	`diners`
	`jcb`
	`elo`
	`dankort`
	`cartebleue`
	`cartasi`

holderName	The complete name of the holder of the credit card

`billingAddress` properties

The following table describes the properties of the `billingAddress` object in the request. The billing address is the address of the customer to whom the order is charged.

Property	Description
lastName	The last name of the customer
postalCode	The postal code in the address (for example, the zip code in the United States)
phoneNumber	The phone number associated with the address
email	The email address associated with the address

Property	Description
state	The state in the address
address1	The first line of the address. Typically the street and number
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order
firstName	The first name of the customer
city	The city in the address
country	The country in the address

`shippingAddress` properties

The following table describes the properties of the `shippingAddress` object in the request. The shipping address is the address of the person (not necessarily a customer) receiving the order.

Property	Description
lastName	The last name of the order recipient
postalCode	The postal code in the address (for example, the zip code in the United States)
phoneNumber	The phone number associated with the address
email	The email address associated with the address
state	The state in the address
address1	The first line of the address. Typically the street and number
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order
firstName	The first name of the order recipient
city	The city in the address
country	The country in the address

`profile` properties

The following table describes the properties of the `profile` object in the request. These values are associated with the customer purchasing the order.

Property	Description
id	The ID of the customer profile
phoneNumber	The phone number from the customer profile
email	The email address from the customer profile

`profileDetails` properties

The following table describes the properties of the `profileDetails` object in the request. These values are associated with the customer purchasing the order. Note that for account-based commerce shoppers, this object may also include `parentOrganization`, `currentOrganization`, and `secondaryOrganizations` subobjects.

Property	Description
id	The ID of the customer profile
lastName	The last name of the customer profile
firstName	The first name of the customer profile
middleName	The middle name of the customer profile
email	The email address from the customer profile
taxExempt	Indicates whether the customer tax-exempt status; either `true` or `false`
taxExemptionCode	For a customer with tax-exempt status, the exemption code
profileType	The type of profile; either `b2c_user` or `b2b_user`
receiveEmail	Indicates whether the customer agrees to receive email; either `yes` or `no`
registrationDate	The timestamp of when the profile was created, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
lastPasswordUpdate	The timestamp of when the password for the profile was last updated, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

Sample authorization request

The following is an example of an authorization request sent by the Credit Card Payment webhook to a payment gateway:

```
{
     "transactionId": "o30446-pg30417-1458555741310",
     "currencyCode": "USD",
     "paymentId": "pg30417",
     "locale": "en",
     "siteURL": "https://www.example.com",
     "gatewaySettings": {
          "paymentMethodTypes": "card",
          "filteredFields": ["paymentMethodTypes"]
     },
     "cardDetails": {
          "expirationMonth": "02",
          "expirationYear": "2022",
          "cvv": "234",
          "number": "4111111111111111",
          "type": "visa",
          "holderName": "Test Shopper"
     },
     "amount": ""000000122526",
     "transactionType": "0100",
     "transactionTimestamp": "2019-11-21T10:22:21+0000",
     "siteId": "siteUS",
     "billingAddress": {
```

```
            "lastName": "Shopper",
            "postalCode": "01242",
            "phoneNumber": "617-555-1977",
            "email": "tshopper@example.com",
            "state": "MA",
            "address1": "1 Main Street",
            "address2": "",
            "firstName": "Test",
            "city": "Cambridge",
            "country": "US"
      },
      "channel": "storefront",
      "shippingAddress": {
            "lastName": "Shopper",
            "postalCode": "01242",
            "phoneNumber": "617-555-1977",
            "email": "tshopper@example.com",
            "state": "MA",
            "address1": "1 Main Street",
            "address2": "",
            "firstName": "Test",
            "city": "Cambridge",
            "country": "US"
      },
      "orderId": "o30446",
      "paymentMethod": "card",
      "gatewayId": "gatewayDemo",
      "profile": {
            "id": "110454",
            "phoneNumber": "617-555-1977",
            "email": "tshopper@example.com"
      },
      "profileDetails": {
            "id": "110454",
            "lastName": "Shopper",
            "firstName": "Test",
            "taxExempt": false,
            "profileType": "b2c_user",
            "receiveEmail": "no",
            "registrationDate": "2019-10-15T06:50:51.000Z",
            "lastPasswordUpdate": "2019-10-15T06:50:51.000Z"
      }
     "retryPaymentCount": 0,
     "auxiliaryProperties": {
            "authenticationMethod": "local",
            "shopperAccountPaymentAccountFirstUseDate":
"2019-10-17T06:14:06.004Z"
      },
}
```


Credit card payment response properties

This section describes the top-level properties and the properties of subobjects that should be returned in the JSON response body of the Credit Card Payment webhook.

Top-level properties

The following table describes the top-level properties that Oracle Commerce expects in the webhook response.

Property	Description
paymentId	The ID of the internal payment group. Must match the value from the request.
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens.

transactionType	A code indicating the type of transaction. For the Credit Card Payment webhook, this must be one of the following numeric values:
	
	`0100 (authorization)`
	`0110 (void)`
	`0400 (refund)`
	
	Must match the value from the request.

transactionTimestamp	The timestamp of the transaction in Oracle Commerce, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ` Must match the value from the request.
hostTransactionTimeStamp	The timestamp of the transaction from the gateway (in milliseconds).
paymentMethod	The payment method. For the Credit Card Payment webhook, the value must be `card`.
orderId	The ID of the order associated with the payment. Must match the value from the request.
amount	The amount authorized. This must match the exact amount requested. Any other amount will cause an error, as Oracle Commerce does not support partial authorizations for credit card payments. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
currencyCode	The ISO 4217 currency code. This is expected to match the value in the request.
gatewayId	The ID of the payment gateway. Must match the value from the request.
siteId	The ID of the site on which the order was placed. Must match the value from the request.
additionalProperties	Key/value pairs for additional properties sent by the merchant.

`authorizationResponse` properties

The following table describes the properties of the `authorizationResponse` object in the response. The values of these properties indicate whether the transaction was authorized successfully.

Property	Description

responseCode	The authorization decision from the payment provider as interpreted by the merchant. For the Credit Card Payment webhook, this must be one of the following values:
	
	`1000 (success)`
	`9000 (decline)`

responseDescription	Information from the payment gateway about the response
responseReason	Information about why the authorization succeeded or failed
authorizationCode	The authorization code from the payment provider
hostTransactionId	The transaction reference ID from the payment gateway
merchantTransactionId	The transaction reference ID from the merchant
token	The payment token used by the payment provider

Sample authorization response

The following is an example of an authorization response sent to the Credit Card Payment webhook by a payment gateway:

```
{
     "orderId": "o30446",
     "currencyCode": "USD",
     "transactionId": "o30446-pg30417-1458555741310",
     "paymentId": "pg30417",
     "amount": "000000122526",
     "transactionType": "0100",
     "hostTransactionTimestamp": "1447807667046",
     "transactionTimestamp": "2019-11-21T10:22:21+0000",
     "paymentMethod": "card",
     "gatewayId": "gatewayDemo",
     "siteId": "siteUS",
     "authorizationResponse": {
          "responseCode": "1000",
          "responseReason": "1001",
          "responseDescription": "1002",
          "authorizationCode": "s001",
          "hostTransactionId": "h001"
     },
     "additionalProperties": {
          "sampleProperty1": "An additional property whose value will be
stored."
     }
}
```

38

Create a Generic Payment Gateway Integration

For a more general solution than the one provided by the Credit Card Payment webhook, you can create an integration that uses the Generic Payment webhook to exchange data with providers of a variety of different payment types.

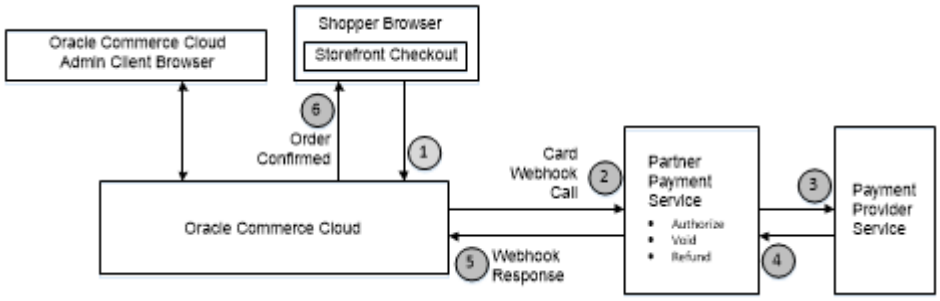
Using the Generic Payment webhook, you can integrate with payment providers for:

- credit cards
- cash payments
- gift cards
- invoices and purchase orders
- store credit
- web checkout systems

This chapter provides an overview of the gateway integrations you can create using the Generic Payment webhook. Subsequent chapters cover specific types of payment providers. Note that the loyalty point payment method uses the Custom Currency webhook.

Understand the generic payment gateway architecture

The following diagram illustrates the generic payment gateway architecture:



Supported payment methods and transaction types

Creating a gateway integration using the Generic Payment webhook is similar to creating an integration using the Credit Card Payment webhook. However, the Generic Payment webhook supports a wider range of options in order to handle a variety of payment methods.

The following table summarizes the available payment methods and the transaction types they support. Note that in addition to the methods listed here for the Generic Payment webhook, Commerce supports loyalty point payments using the Custom Currency webhook.

Payment Method	Supported Transaction Types
`card`	`authorization` – approve payment for an order `void` -- cancel an order or a payment `refund` -- issue a credit to the shopper after a return
`cash`	`initiate` -- create an order to be completed later `cancel` -- cancel an order or a payment
`generic`	`initiate` -- create an order to be completed later `retrieve` -- return an initiated order to complete it `authorization` -- approve payment for an order `void` -- cancel an order or a payment `refund` -- issue a credit to the shopper after a return
`physicalGiftCard`	`balanceInquiry` -- return current available balance `authorize` -- approve payment for an order `void` -- cancel an order or a payment `refund` -- issue a credit to the shopper after a return
`invoice`	`authorization` -- approve payment for an order
`storeCredit`	`balanceInquiry` -- return current available balance `authorize` -- approve payment for an order `void` -- cancel an order or a payment `refund` -- issue a credit to the shopper after a return

The payment and transaction types are specified in the `gateway.json` file. For example:

```
{
  "provider": "Sample Payment Gateway",
  "paymentMethodTypes": ["physicalGiftCard", "cash"],
  "transactionTypes": {
    "physicalGiftCard": ["balanceInquiry", "authorize", "void",
"refund"],
    "cash": ["initiate", "cancel"]
  }
}
```

User interface configuration controls that appear in the Payment Processing page of the administration interface are specified in the `config.json` file. For example:

```
{
  "configType": "payment",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "instances" : [
    {
      "id": "agent",
      "instanceName": "agent",
      "labelResourceId": "agentInstanceLabel"
    },
    {
      "id": "preview",
      "instanceName": "preview",
      "labelResourceId": "previewInstanceLabel"
    },
    {
      "id": "storefront",
      "instanceName": "storefront",
      "labelResourceId": "storefrontInstanceLabel"
    }
  ],
  "properties": [
    {
      "id": "paymentMethodTypes",
      "type": "multiSelectOptionType",
      "name": "paymentMethodTypes",
      "required": false,
      "helpTextResourceId": "paymentMethodsHelpText",
      "labelResourceId": "paymentMethodsLabel",
      "defaultValue": "physicalGiftCard",
      "displayAsCheckboxes": true,
      "public": true,
      "options": [
         {
           "id": "cash",
           "value": "cash",
           "labelResourceId": "cashPayLabel"
         },
         {
           "id": "physicalGiftCard",
           "value": "physicalGiftCard",
           "labelResourceId": "physicalGiftCardPayLabel"
         }
       ]
    },
    {
      "id": "includeOrderInWebhookPayload",
      "type": "booleanType",
      "name": "includeOrderInWebhookPayload",
      "helpTextResourceId": "includeOrderHelpText",
      "labelResourceId": "includeOrderLabel",
      "defaultValue": true,
```

```
      "public": true
    }
  ]
}
```

Notice that in this example the `type` attribute of the `paymentMethodTypes` property is set to `multiSelectOptionType`, which means that multiple methods can be selected (for example, `physicalGiftCard` and `cash`). By default the control created for selecting the methods is a drop-down list, but setting `displayAsCheckboxes` to `true` specifies that a set of checkboxes should be used instead.

The `includeOrderInWebhookPayload` property creates a checkbox for specifying whether or not to include the order data in the webhook call.

Send custom properties to a payment gateway

When payment transaction data is sent to a payment provider by the Generic Payment webhook, the provider processes the payment information and returns information about the transaction.

The webhook sends out a predefined set of properties in the request, and expects to receive another predefined set of properties back in the response. For some providers, however, you may need to send additional data in the request, and the provider may include additional data in the response. This section describes how you can send and receive additional data that is not included in the predefined properties.

Include custom properties in the REST call

Depending on the payment provider you integrate with, there may be additional payment data that you want to send. If so, you can include this data in the `createOrder` REST request. Each `payments` object in the request can include a `customProperties` subobject that you can use to send additional data as key/value pairs. For example:

```
...
"payments": [
  {
    "endYear": 2018,
    "cardTypeName": "Visa",
    "nameOnCard": "Fred Smith",
    "customProperties": {
      "monthlyCharge": "$77",
      "numberOfPayments": "12"
    },
    "cardCVV": "123",
    "type": "card",
    "cardType": "visa",
    "endMonth": "02",
    "cardNumber": "4055011111111111"
  }
],
...
```

Send custom properties in the webhook request

The custom properties from the REST request are then included in the `customProperties` object in the webhook call to the payment provider. For example:

```
{
     "transactionId": "o60412-pg60411-1465342612829",
     "currencyCode": "USD",
     "paymentId": "pg60411",
     "siteId": "siteUS",
     "locale": "en",
     "customProperties": {
       "monthlyCharge": "$77",
       "numberOfPayments": "12"
     },
     "gatewaySettings": [{
       "paymentMethodTypes": "card",
       "filteredFields": ["paymentMethodTypes"]
     }],
     "amount": "000000007700",
     "transactionType": "0100",
...
```

Note that for gift card payments, in addition to the top-level `customProperties` object, each `paymentRequests` object also has a `customProperties` object. See Integrate with a Gift Card Payment Gateway for more information.

Return custom properties in the webhook response

The webhook can return custom properties from the payment provider as an `additionalProperties` object in the response. This data is saved with the payment group for the order. The webhook can also return a `customPaymentProperties` object that specifies a list of the properties in the `additionalProperties` object that should be returned to the storefront in the response to the original `createOrder` request. For example:

```
{
     "orderId": "o60412",
     "paymentId": "pg60411",
     "siteId": "siteUS",
     "merchantTransactionId": "324a5107-8fe5-4dd7-aa1f-8b7e2e0ec8df",
     "hostTransactionId": "o60412-pg60411-1465342612829",
     "transactionTimestamp": "2016-06-07T23:36:52+0000",
     "hostTimestamp": "2016-06-07T23:36:52+0000",
     "transactionType": "0100",
     "additionalProperties": {
          "interestRate": "0.05",
          "remainingPayments": "5",
          "latePayment": false,
      },
     "customPaymentProperties": ["remainingPayments", "latePayment"],
     "amount": "000000007700",
     "currencyCode": "USD",
...
```

Incorporate 3D-Secure support

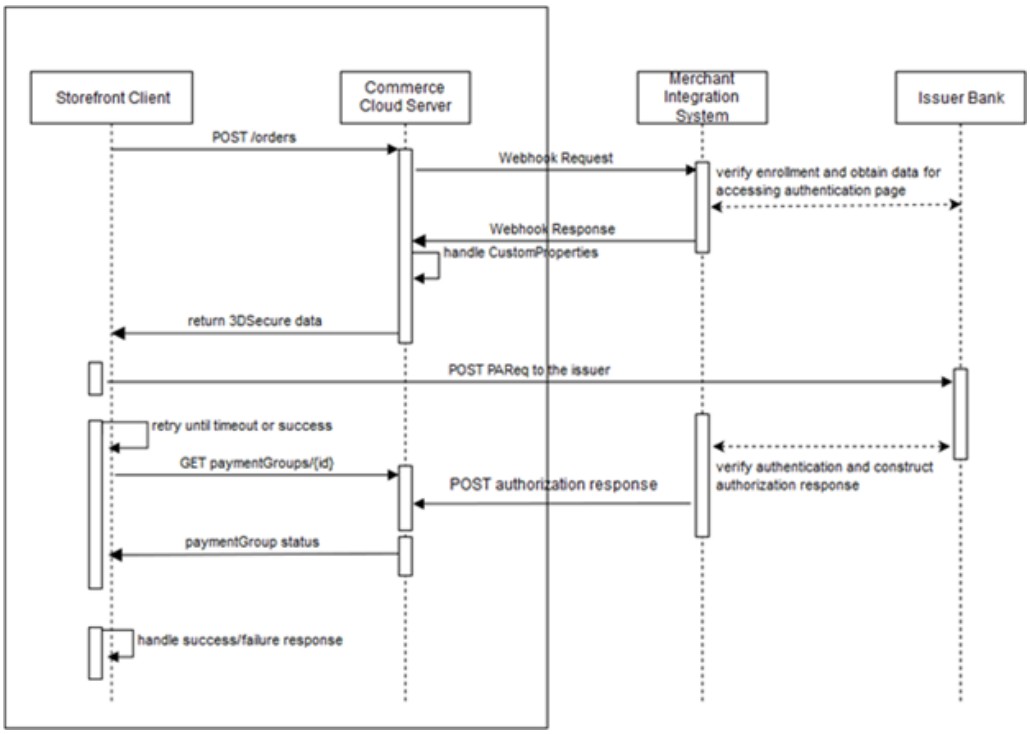
The generic payment gateway's credit card payment method includes optional support for 3D-Secure for shopper verification.

If 3D-Secure is required, then before the merchant authorizes a payment, the shopper is redirected to a page provided by the card issuer for authentication. If the shopper authenticates successfully, the card issuer and merchant then determine whether to authorize the transaction.

Note that whether 3D-Secure is required depends on the merchant and the card issuer, and is not controlled by Oracle Commerce. The gateway described in this section invokes 3D-Secure only when it is required, and can process non-3D-Secure payments as well.

Understand 3D-Secure support

The following diagram illustrates how credit card payments are handled in a generic payment gateway integration that implements 3D-Secure support:



The payment processing involves the following steps:

1. When the shopper clicks Place Order, the storefront invokes the `createOrder` endpoint of the Store API. The endpoint sends the order information to the Commerce server.
2. When the server receives the order submission, it invokes the Generic Payment webhook, which posts an authorization request to the merchant.
3. The merchant and the card issuer communicate to determine whether 3D-Secure is required, and whether the shopper is enrolled in the card issuer's 3D-Secure program. If 3D-Secure is required and the shopper is enrolled, the card issuer sends an ACS (access control server) URL to the merchant.

4. The merchant sends the webhook response to the Commerce server. If 3D-Secure is required for the transaction, the merchant includes the response code `10000 (PAYER_AUTH_REQUIRED)`. The merchant supplies the ACS URL and other data needed for invoking the 3D-Secure authentication page on the card issuer's website.
5. The Commerce server sends the data it receives in the webhook response, including the 3D-Secure data, to the storefront in the `createOrder` endpoint response. The `uiIntervention` property for the payment group is set to `true` in the response to indicate that 3D-Secure authentication is required.
6. The storefront posts a payment request to the card issuer's website to invoke the 3D-Secure authentication page. The request includes data returned from the merchant in the webhook response.
7. The card issuer displays the authentication page.
8. The shopper fills out the authentication form and submits it.
9. The card issuer and the merchant communicate to determine if the shopper authenticated successfully, and if so, whether to authorize the transaction.
10. The merchant constructs an authorization response and sends it to the Commerce server using the `POST /ccstore/v1/payment/genericCardResponses` endpoint.
11. Meanwhile, after posting the payment request to the issuer's website, the storefront begins polling the Commerce server using the `getPaymentGroup` endpoint to detect when the server receives the authorization response from the merchant.
12. When the Commerce server receives the authorization response from the merchant, the server includes the data from the merchant in the `getPaymentGroup` endpoint response it sends to the storefront.

These steps are described in greater detail below.

Note: 3D-Secure is not applicable to payment requests that originate from the Oracle Commerce Agent Console. If the value of the `channel` property in a Generic Payment webhook request is `agent`, the merchant should map the transaction appropriately in the gateway so the card issuer bypasses 3D-Secure.

Create the gateway extension

As discussed in Supported payment methods and transaction types, the payment and transaction types are specified in the `gateway.json` file. For a credit card gateway that supports 3D-Secure, the `gateway.json` file should be similar to the following:

```
{
  "provider": "Generic Card 3DS Provider",
  "paymentMethodTypes": ["card"],
  "transactionTypes": {
    "card": ["authorization", "void", "refund"]
  },
  "processors" : {
    "card": "card3ds"
  }

}
```

Note that the `card3ds` processor is needed to provide 3D-Secure support.

In addition to configuring user interface controls, the `config.json` file must include a shared secret key specified by the merchant. The key is used to generate a signature that the `POST /ccstore/v1/payment/genericCardResponses` endpoint uses for authentication:

```
...
    {
      "id": "secretKey3DS",
      "type": "passwordType",
      "name": "secretKey",
      "helpTextResourceId": "secretKeyHelpText",
      "labelResourceId": "secretKeyLabel",
      "defaultValue": "5ad0f437X6af6X4d4eXb08cX729a310843ce",
      "required": true,
      "public": true
    },
...
```

See Generate the signature for more information about how the secret key is used.

Send the webhook response

If 3D-Secure is required for a transaction, the merchant returns a `responseCode` value of `10000 (PAYER_AUTH_REQUIRED)` when it sends the Generic Payment webhook response to the Commerce server. The payment group is not updated.

The merchant uses the `additionalProperties` map in the webhook response to supply data needed for invoking the 3D-Secure authentication page on the card issuer's website. This data typically includes values such as `acsURL` (the issuer's URL to direct the shopper to for authentication), `paReq` (the payer authentication request), `MD` (merchant data), and `TermURL` (the URL to return the shopper to after authentication). The exact set of properties, and the names for these properties, may differ depending on the card issuer. The merchant can also include `maxRetryCount` and `delayInMillis` properties as part of the `additionalProperties` map to configure the storefront's polling behavior.

The webhook also returns a `customPaymentProperties` array that specifies a list of the properties from the `additionalProperties` map that should be returned to the storefront. For example:

```
{
  "transactionType": "0100",
  "orderId": "o140451",
  "siteId": "siteUS",
  "channel": "preview",
  "locale": "en",
  "currencyCode": "USD",

  "authorizationResponse": {
      "additionalProperties": {
           "delayInMillis": "10000",
           "payerAuthEnrollReply_proxyPAN": "1078787",
           "amount": "000000003499",
           "orderId": "o140451",
```

```
            "channel": "storefront",
            "maxRetryCount": "5",
            "locale": "en",
            "transactionId": "o140451-pg140415-1480662437847",
            "transactionTimestamp": "2016-12-02T07:07:17+0000",
            "transactionType": "0100",
            "payerAuthEnrollReply_paReq": "eNpVkctuNBgiESLAKGWnXGsJiUPcByU/
n3thQd",
            "paymentId": "pg140415",
            "payerAuthEnrollReply_acsURL":
               "http://www.example.com/ccstore/v1/genericCardAuth3DS",
            "paymentMethod": "card",
            "displayMessage": "Please wait .....",
            "payerAuthEnrollReply_xid": "Skh0MTRsZGsxYXZPbEd4a2I1VjA=",
            "TermUrl":
               "http://www.example.com/ccstore/v1/payment/
genericCardResponses",
            "currencyCode": "USD",
            "gatewayId": "gateway3DS"
             },
      "customPaymentProperties": ["delayInMillis",
          "payerAuthEnrollReply_proxyPAN", "amount", "decision", "orderId",
          "channel", "maxRetryCount", "locale", "transactionId",
          "transactionTimestamp", "transactionType",
"payerAuthEnrollReply_paReq",
          "paymentId", "payerAuthEnrollReply_acsURL", "paymentMethod",
          "displayMessage", "reasonCode", "payerAuthEnrollReply_xid",
"TermUrl",
          "currencyCode", "gatewayId"],
      "responseCode": "10000"
  }

}
```


Authorize the payment

The Commerce server sends the data from the webhook response, including the 3D-Secure data, to the storefront in the `createOrder` endpoint response. The storefront then posts a payer authentication request to the card issuer's website using data received from the merchant. The issuer displays an authentication page in an inline frame on the Commerce storefront. (See Create a custom payment authorization widget for information about how to customize the storefront to do this.)

After the shopper fills out the authentication form and submits it, the card issuer and the merchant communicate to determine if the shopper authenticated successfully, and if so, whether to authorize the transaction. The merchant then sends an authorization response to the Commerce server, using the `POST /ccstore/v1/payment/genericCardResponses` endpoint.

The following table describes the `POST` properties:

Property	Description
transactionType	Match value from webhook request.
currencyCode	Match value from webhook request.

Property	Description
locale	Match value from webhook request.
channel	Match value from webhook request.
orderId	Match value from webhook request.
signedKeys	A comma-separated list of the properties that are used to generate the signature. See Generate the signature.
signature	The Base64 signature returned by the merchant. See Generate the signature.
authorizationResponse	A JSON map of key/value pairs containing authorization data

For example, the body of the `POST` might include:

```
<input id="transactionType" name="transactionType" type="hidden"
value="0100"/>
<input id="currencyCode" name="currencyCode" type="hidden"
value="USD"/>
<input id="locale" name="locale" type="hidden" value="en"/>
<input id="channel" name="channel" type="hidden" value="preview"/>
<input id="orderId" name="orderId" type="hidden" value="o120419"/>
<input id="signedKeys" name="signedKeys" type="hidden"
value="transactionType,

currencyCode,locale,channel,orderId,paymentId,transactionId,paymentMeth
od,
  gatewayId,amount,merchantTransactionId,authCodes"/>
<input id="signature" name="signature" type="hidden"
  value="5ad0f437X6af6X4d4eXb08cX729a310843ce"/>
<input id="authorizationResponse" name="authorizationResponse"
type="hidden"
  value="authorization_response_JSON_map"/>
```

The following table lists the properties of the JSON map that the `authorizationResponse` property in the `POST` is set to. All properties are required unless specified otherwise:

Property	Description
paymentId	Match value from webhook request.
transactionId	Match value from webhook request.
transactionTimestamp	Match value from webhook request.
paymentMethod	Match value from webhook request.
gatewayId	Match value from webhook request.
siteId	Must match the value from the request.
amount	The amount authorized. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`

Property	Description
merchantTransactionId	The transaction reference ID from the merchant
merchantTransactionTimestamp	The timestamp of the transaction from the merchant (in milliseconds)
hostTransactionId	The transaction reference ID from the payment gateway (optional)
hostTransactionTimestamp	The timestamp of the transaction from the gateway, in milliseconds (optional)

responseCode	The authorization decision from the payment provider as interpreted by the merchant. Must be one of the following values:
	
	`1000 (success)`
	`4000 (sale complete)`
	`9000 (decline)`

responseReason	Information about why the authorization succeeded or failed
responseDescription	Information from the payment gateway about the response
authCode	The authorization code for the transaction
token	The payment token used by the payment provider (optional)
additionalProperties	Key/value pairs for additional properties sent by the merchant (optional)
customPaymentProperties	A list of the properties in the additionalProperties object that should be returned to the storefront (optional)

The following is an example of the JSON map that is supplied as the value of the `authorizationResponse` property in the `POST`. Note that you need to use HTML entities to replace certain characters such as quotation marks before including the map in the `POST`:

```
{
   "paymentId": "pg130411",
   "transactionId": "o120419-pg130411-1478862352044",
   "transactionTimestamp": "2016-08-05T12:24:54+0000",
   "paymentMethod": "card",
   "gatewayId": "gateway3DS",
   "siteId": "siteUS",
   "amount": "000000009349",
   "merchantTransactionId": "mID1470399894815",
   "merchantTransactionTimestamp": "1470399894815",
   "hostTransactionId": "hID1470399894715",
   "hostTransactionTimestamp": "1470399894715",
   "responseCode": "1000",
   "responseReason": "AuthResponseReason",
   "responseDescription": "AuthResponseDescription",
   "authCode": "AUTH-ACCEPT",
   "token": "token-success",
   "additionalProperties":
```

```
      {
      "sample-addnl-property-key1": "sample-payment-property-value1",
      "sample-addnl-property-key2": "sample-payment-property-value2"
      },
   "customPaymentProperties": ["sample-addnl-property-key2"]
}
```

Generate the signature

The merchant uses the shared secret key to generate a signature that it supplies when it sends the authorization response using the `POST /ccstore/v1/payment/genericCardResponses` endpoint. When the Commerce server receives authorization response, it applies the same logic that the merchant uses to calculate the signature, and accepts the authorization only if both signatures match.

The signature is generated on the merchant server by performing an `HmacSHA256` hash of the `signedKeys` properties using the shared secret key. The minimum recommended set of properties to include in `signedKeys` is:

```
signedKeys=transactionType,currencyCode,locale,channel,orderId,paymentI
d,
transactionId,paymentMethod,gatewayId,amount,merchantTransactionId,auth
Code
```

Using the properties listed in `signedKeys`, construct a comma-separated list of key/value pairs. For example, using the `signedKeys` value from the `authorizationResponse` data in the example above, the list would be:

```
transactionType=0100,currencyCode=USD,locale=en,channel=preview,orderId
=o120419,
paymentId=pg130411,transactionId=o120419-pg130411-1478862352044,
paymentMethod=card,gatewayId=gateway3DS,amount=000000009349,
merchantTransactionId=mID1470399894815,authCode=AUTH-ACCEPT
```

Note: Do not include any URL-encoded characters in the list.

Using the list of key/value pairs and the shared secret key, perform the hash to generate the signature. For example:

```
...

// secretKey - shared secret Key provided by merchant in the gateway
extension
// dataToSign - comma separated key/value string
SecretKeySpec secretKeySpec = new
SecretKeySpec(secretKey.getBytes("UTF-8"),
"HmacSHA256");
Mac mac = Mac.getInstance("HmacSHA256");
mac.init(secretKeySpec);
byte[] rawHmac = mac.doFinal(dataToSign.getBytes("UTF-8"));
Base64.getEncoder().encodeToString(rawHmac).replace("\n", "");
```

Retrieve the authorization response

After posting the payment request to the issuer's website, the storefront begins polling the Commerce server using the `getPaymentGroup` endpoint to determine if the server has received the authorization response from the merchant. When the Commerce server receives the authorization response, it includes the data from the merchant in the `getPaymentGroup` endpoint response it sends to the storefront.

Create a custom payment authorization widget

In order for your storefront to use 3D-Secure, you need to write a custom widget and use it to replace the CyberSource Payment Authorization Widget on the Payer Authentication Layout. The custom widget must manage various communications between the storefront, the Commerce server, and the credit card issuer.

The `widget.json` file should be similar to the following:

```
{
  "name": "Generic Card 3DS Widget",
  "javascript": "genericCard3DS",
  "jsEditable": true,
  "global": false,
  "i18nresources": "genericCard3DS",
  "imports": [
    "payment",
    "paymentauthorization",
    "order",
    "site"
   ],
  "pageTypes": ["payment"]
}
```

Write the widget JavaScript

The widget's JavaScript code should implement the following logic:

- Listen for the `ORDER_AUTHORIZE_PAYMENT` event and initiate payer authentication.
- Populate the authentication form and submit it to the issuer's URL.
- Publish a `PAYMENT_GET_AUTH_RESPONSE` event to trigger polling the Commerce server to detect when it receives the authorization response from the merchant.
- Handle timeout and error cases.

This section includes examples of code that implements these operations.

Listening for the `ORDER_AUTHORIZE_PAYMENT` event and initiating payer authentication:

```
$.Topic(pubsub.topicNames.ORDER_AUTHORIZE_PAYMENT).subscribe(function(obj) {
  if (obj[0].details) {
    widget.authDetails = obj[0].details;
    widget.createSignatureIfIframeIsLoaded(widget.authDetails, 0);
  }
});
```

Populating the authentication form and submitting it to the issuer's URL:

```
widget.injectFormValuesForPayerAuth(
  uiIntervenedPaymentGroup.customPaymentProperties);
widget.injectActionURL(

uiIntervenedPaymentGroup.customPaymentProperties.payerAuthEnrollReply_a
csURL);
widget.postForm();
```

Publishing a `PAYMENT_GET_AUTH_RESPONSE` event to trigger polling the Commerce server to detect when it receives the authorization response from the merchant:

```
var messageDetails = [{message: "success",
orderid: authDetails.orderDetails.id,
orderuuid: authDetails.orderDetails.uuid,
paymentGroupId: uiIntervenedPaymentGroup.paymentGroupId,
numOfRetries:
uiIntervenedPaymentGroup.customPaymentProperties.maxRetryCount,
delay: uiIntervenedPaymentGroup.customPaymentProperties.delayInMillis
  }];

$.Topic(pubsub.topicNames.PAYMENT_GET_AUTH_RESPONSE).publish(messageDet
ails);
```

Handling the error cases, such as being unable to access the issuer URL, or receiving an authentication error from the issuer. For example:

```
widget.handleErrors = function() {
  try {
    $.Topic(pubsub.topicNames.ORDER_SUBMISSION_FAIL).publish([{message:
 "fail"}]);
  }
  catch(e) {
    log.error('Error Handling Order Fail');
    log.error(e);
  }
  try {
    widget.handleTimeout();
  }
  catch(e) {
    log.error('Error Handling Timeout');
    log.error(e);
  }
};
```

Support stored credit cards

When you create a Generic Payment gateway integration for credit card payments, you can enable the integration to allow logged-in shoppers to store card data in their profiles, and then later access the stored cards when they place orders.

Commerce does not store the complete credit card data. Instead, when a shopper stores a credit card, the payment processor associated with the gateway sends back a token that is stored with the shopper's profile. When the shopper places orders in the future, he or she is given a list of saved cards and can select the one to use. The token associated with the selected card is then sent to the gateway, which retrieves the card data and sends the authorization request to the payment processor.

Tokens are stored on a per-gateway basis. This has important implications in an environment running multiple sites, because different sites may use different gateway extensions. For example, if you have two sites that require different gateway configurations for credit cards, you must create a separate gateway extension for each site. In this case, a card stored on one site will not automatically be stored for the other, because the gateways require separate tokens. (The shopper can store the same card on each site separately, though.) But if both sites use the same gateway configuration, they can share the same gateway extension. In this case, a card stored on one site will also be stored for the other.

Save and use stored credit cards

This section describes the workflow supported for storing and using saved credit cards. Note that you must implement this logic on your storefront; the widgets included with Commerce cannot handle stored cards by default.

Store a credit card when placing an order

The following is the logic you implement for storing a credit card when a shopper places an order:

1. When a shopper pays for an order with a credit card that has not been previously saved by the current payment gateway, the checkout page provides an option for storing the card. The shopper selects the option to indicate that the card should be saved.
2. When the shopper submits the order, the storefront invokes the `createOrder` or `updateOrder` endpoint of the Store API. The endpoint sends the order information to the Commerce server, along with information about the credit card, which includes a flag indicating the credit card should be saved.
3. When the server receives the order submission, it invokes the Generic Payment webhook, which posts an authorization request to the gateway. The webhook request contains the credit card information that was sent to the server by the endpoint, including the flag indicating the credit card should be saved.
4. The gateway saves the card information and generates a token to associate with the card. It sends the authorization request to the payment provider.
5. The provider sends a response back to the gateway, which it passes on to the Commerce server along with the token.
6. The Commerce server stores the token and passes the authorization response on to the storefront.

Store credit cards without placing an order

Commerce provides an Update Profile store API endpoint that lets you add and store customer credit cards as part of a customer Billing Profile without actually placing an order.

The name of the endpoint is `addCreditCard`. The URI for the endpoint is `POST /ccstore/v1/current/creditCards/`.

The endpoint can be used to invoke Add Card requests multiple times to let you add more than one card to a profile. Each new card is then stored against the profile. The inputs of this endpoint are:

- `cardType`
- `nameOnCard`
- `cardNumber`
- `expiryMonth`
- `expiryYear`

This endpoint triggers the Generic Payment webhook for a Tokenize operation on the payment system. The payment system is expected to return a tokenized value of the card which is then saved against the profile. The endpoint then returns back a stored card ID.

Note: To support adding credit cards directly to a shopper profile, you must configure the Generic Payment webhook and specify the `card3ds` processor in the gateway extension's `config.json` file. Specifying the `card3ds` processor is required, although you do not actually need to implement 3D-Secure support unless your store requires it.

Use a stored card

The following describes the logic you implement to enable shoppers to use stored credit cards for order payments:

1. When the shopper accesses the checkout page, the storefront calls the `listCreditCards` endpoint of the Store API. The Commerce server sends a response that includes information about the shopper's stored credit cards. The storefront displays the cards and provides controls for selecting a card.
2. The shopper selects the card to use. Depending on how the payment gateway is configured, the shopper may need to then provide the CVV.
3. The shopper submits the order. The storefront invokes the `createOrder` or `updateOrder` endpoint of the Store API, which sends the order information to the Commerce server.
4. When the server receives the order submission, it retrieves the token associated with the credit card. It invokes the Generic Payment webhook, which posts an authorization request to the gateway, along with the token.
5. The gateway retrieves the card data and sends the authorization request to the payment provider. The provider sends a response back to the gateway, which it passes on to the Commerce server.
6. The Commerce server passes the authorization response on to the storefront.

The payload of the authorization request includes several properties for tracking information about transactions involving a stored credit card:

- `originOfOrder` -- A top-level property that indicates the source of the order. Valid values: `default`, `scheduledOrder`, `contactCenter`, `punchout`, `purchaseOrder`, `bulk`.
- `storedCardUsed` -- A boolean property of the `cardDetails` object that is set to `true` for transactions involving a stored credit card.
- `additionalSavedCardProperties` -- An object containing custom properties that are sent by the gateway in the webhook response when a card is stored. Each

subsequent time the saved card is used, these properties are updated with values received in the associated webhook response.

The following example shows a portion of an authorization request that includes these properties:

```
{
...
"originOfOrder": "scheduledOrder",
"paymentId": "pg40429",
"cardDetails": {
  "expirationYear": "2029",
  "storedCardUsed": true,
  "number": "411111xxxxxx1111",
  "tokenExpiryDate": "2023-05-14 06:45:35.0",
  "expirationMonth": "11",
  "additionalSavedCardProperties":
   {
     "prop1": "val1",
     "prop2": "val2",
     "prop3": "val3"
   }
,
"type": "visa",
"maskedCardNumber": "xxxxxxxxxxxx1111",
"token": "Token-1557816335786"
...
}
```

Manage cards

The `createOrder` endpoint's payments array includes properties for specifying that the card used for the order should be saved, as well as additional properties for optionally making it the default card for the shopper and for specifying a nickname for the card (for example, WorkAMEX). In addition, the checkout page can call the `listCreditCards` endpoint to display a list of the cards stored for the current site, so a shopper will be able to select the card to use when placing a future order.

You can also use the `listCreditCards` endpoint to display the shopper's stored cards on the Your Account page. In addition, there are several other endpoints that you can use to enable shoppers to update card details and delete cards.

Note that by default the `listCreditCards` endpoint returns only the shopper's cards that are active and apply to the current gateway and current site. This ensures that the cards displayed on the checkout page are all valid for the order being placed. The endpoint also supports query parameters that can be used to return all of a shopper's stored cards, regardless of site, gateway, or active status. These parameters should be used on the Your Account page, as shown in Create a saved credit card widget.

Configure the payment gateway integration to support stored cards

To configure a payment gateway integration that supports storing credit cards, the `config.json` file in the gateway extension can include these settings:

- `isCVVRequiredForSavedCards` – A boolean indicating whether the shopper must supply the CVV when using a stored card. Default is `true`.

- `enabledForScheduledOrders` – A boolean indicating whether the gateway supports using stored cards as payment for scheduled orders. Default is `true`.
- `isCVVRequiredForScheduledOrders` – A boolean indicating whether the shopper must supply the CVV for each instance of a scheduled order. Default is `false`, which allows the instances to be processed without the CVV. See Use stored credit cards for scheduled orders for more information.

Note: Because the scheduler runs on the store server, if you have an Oracle Commerce Agent Console configuration, you should configure the gateway so that both the storefront and the agent instances use the same value for the `isCVVRequiredForScheduledOrders` flag. This prevents the settings used in the storefront from overwriting the scheduled order settings used in the agent environment.

Use stored credit cards for scheduled orders

A shopper can use a stored credit card to pay for scheduled orders. This provides an alternative to sending an invoice to the shopper after each order is placed. Note that the credit card must have already been stored before creating the scheduled order. The ID of the selected credit card is retained on the scheduled order, and is used to retrieve the token that is sent when processing an instance of the order.

Submission of the orders depends on whether the shopper is required to take further action, such as supplying a CVV or 3D-Secure login credentials. If no shopper intervention is required, order instances are submitted automatically in the background, and the shopper is sent an email indicating the order has been submitted. To avoid the need for shopper intervention, you can set the payment gateway's `isCVVRequiredForScheduledOrders` to false. This setting allows scheduled orders to be processed without the CVV.

If the shopper is required to take further action, then when an instance of the order is created, it is placed in the `PENDING_PAYMENT` state and an email is sent to the shopper about the action required. Similarly, if a card-related problem occurs (for example, the card has been deleted or has expired), the order instance moves to the `PENDING_PAYMENT` state and an email is sent to the shopper.

After receiving an email indicating further action is required, the shopper can access the order and do any of the following:

- Enter any required card information (for example, update the expiration date) and submit the order.
- Change the payment method and submit the order. Note that the payment method change applies only to this order instance; future instances of the order still use the credit card associated with the order.
- Cancel the order instance.

Note that for a scheduled order, the value of the `orderId` property in the Generic Payment webhook authorization request is the ID for the individual order instance, not for the order template. Also, the value of the `originOfOrder` property in the request is set to `scheduledOrder`.

Use stored credit cards for orders requiring approval

An account-based commerce shopper can use a stored credit card to pay for an order that requires approval. The card must have already been stored before creating the

order. The shopper's credit card information cannot be seen by delegated administrators or approvers.

Once an order has been created and approved, submission of the order depends on whether the shopper is required to take further action, such as supplying a CVV or 3D-Secure login credentials. If no shopper intervention is required, the order is submitted upon approval, and the shopper is sent an email indicating the order has been submitted.

If the shopper is required to take further action, then once the order is approved, it is placed in the `PENDING_PAYMENT` state and an email is sent to the shopper about the action required. Similarly, if a card-related problem occurs (for example, the card has been deleted or has expired), the order instance moves to the `PENDING_PAYMENT` state and an email is sent to the shopper.

After receiving an email indicating further action is required, the shopper can access the order and do any of the following:

- Enter any required card information (for example, update the expiration date) and submit the order.
- Change the payment method and submit the order.
- Cancel the order instance.

If an order is rejected by an approver, there is no effect on the stored credit card. It remains available for use with other orders.

Customize your storefront to support stored cards

To add support for stored credit cards to your storefront, you must customize some of the widgets. This section describes the fields in the credit card view model that enable access to stored cards, and provides guidance for creating a new widget for managing stored cards on the Your Account page, as well as for customizing the Split Payments widget on the checkout page to enable saving and retrieving stored cards.

View model support for stored cards

The credit card view model includes several fields for working with stored credit cards:

- `nickname` -- Stores a nickname supplied by the shopper to identify the card.
- `isSavedCard` – A boolean used to indicate whether a card has been stored.
- `saveCard` – A boolean used when the shopper enters a new card for an order, indicating whether the card should be saved.
- `setAsDefault` – A boolean that indicates whether the card is the default card.
- `isCVVRequiredForSavedCards` – A boolean whose value is set from the gateway property of the same name. If `true`, the shopper must supply the CVV when using a stored card.

Modify the Split Payments widget

To support storing credit cards, modify the display template of the Split Payments widget to add checkboxes for setting the `saveCard` and `setAsDefault` properties of the credit card view model. In addition, you will need to make changes to the widget's JavaScript to add the ability to select a previously stored card to pay for an order.

The following example shows a function which calls the `listCreditCards` endpoint to retrieve stored cards. It takes the results from this REST call, and for each credit card returned it calls

the view model's `populateData()` function to create a credit card object. Each credit card object is stored as an entry in an `observableArray` named `allCreditCards`:

```
getCreditCardsForProfile: function() {
   var widget = this;
   var inputData = {};
   var url = "listCreditCards";
   var maskedNumberRegex = /\d(?=\d{4})/g;
   var maskedSymbol = "*";
   ccRestClient.request(url, inputData,
       function(data){
       data.creditCards=data.items;
         for (var i = 0; i < data.creditCards.length; i++) {
           var creditCard =
widget.paymentsContainer().createPaymentGroup(
             CCConstants.CARD_PAYMENT_TYPE);
           creditCard.populateData(data.creditCards[i]);
           creditCard.isSavedCard(true);
           widget.allCreditCards.push(creditCard);
         }
         if (data.creditCards.length > 0) {
           widget.resetSelectedSavedCardId();
           widget.allCreditCards()
[0].amount.subscribe(function(newVal) {
             console.log(newVal);
           });
     }
       },
       function(data) {
         console.log("Error while retrieving the credit cards");
    });
}
```

The widget includes fields named `selectedSavedCardId` (to hold the ID of the selected card) and `orderDefaultSavedCardId` (to hold the ID of the default card). The `onLoad()` function contains code that sets the initial values of these fields:

```
onLoad: function(widget) {
   widget.resetSelectedSavedCardId = function() {
      widget.selectedSavedCardId(null);
      for (var i = 0; i < widget.allCreditCards().length; i++) {
        widget.allCreditCards()[i].resetCardCvv();
        widget.allCreditCards()[i].cardCVV.isModified(false);
        if (widget.orderDefaultSavedCardId ==
          widget.allCreditCards()[i].savedCardId()) {
          widget.selectedSavedCardId(widget.orderDefaultSavedCardId);
          break;
        }
      }
      if (widget.selectedSavedCardId() == null) {
        for (var i = 0; i < widget.allCreditCards().length; i++) {
          if (widget.allCreditCards()[i].isDefault() == true) {
            widget.selectedSavedCardId(widget.allCreditCards()
[i].savedCardId());
```

```
            break;
          }
        }
      }
      if (widget.selectedSavedCardId() == null &&
        widget.allCreditCards().length>0) {
        widget.selectedSavedCardId(widget.allCreditCards()[0].savedCardId());
      }
    }

    widget.addCardToPaymentViewModel = function() {
      for (var i = 0; i < widget.allCreditCards().length; i++) {
        if (widget.selectedSavedCardId() ==
          widget.allCreditCards()[i].savedCardId()) {
          var newCard = widget.paymentsContainer().createPaymentGroup(
          CCConstants.CARD_PAYMENT_TYPE)
          newCard.populateData(ko.mapping.toJS(widget.allCreditCards()[i]));
          newCard.cardCVV(widget.allCreditCards()[i].cardCVV());
          newCard.isSavedCard(true);
          if (widget.allCreditCards()[i].cardCVV() === undefined) {
                widget.allCreditCards()[i].cardCVV.isModified(true);
          }
          widget.paymentViewModel(newCard);
        }
      }
    };
...
}
```

The widget's `beforeAppear()` function clears the values of these fields and calls the `getCreditCardsForProfile()` function to repopulate the fields with the current data:

```
beforeAppear: function (page) {
   var widget = this;
   widget.orderDefaultSavedCardId = null;
   widget.allCreditCards.removeAll();
   widget.getCreditCardsForProfile();
...
}
```

Create a saved credit card widget

In addition to modifying the Split Payments widget as described above, you will need to create a new widget for displaying and modifying saved credit cards on the Your Account page.

This widget includes a function called `getCreditCardsForProfile()` that is called by the `beforeAppear()` function. This is similar to the `getCreditCardsForProfile()` function in the updated Split Payments widget, except that when it calls the `listCreditCards` endpoint, it uses the `allCards=true`, `allGateways=true`, and `allSites=true` query parameters so that all of the shopper's saved credit cards are displayed:

```
getCreditCardsForProfile: function() {
  var widget = this;
```

```
  var inputData = {"allCards":true, "allGateways":true,
"allSites":true};
  var url = "listCreditCards";
  var maskedNumberRegex = /\d(?=\d{4})/g;
  var maskedSymbol = "*";
  CCRestClient.request(url, inputData,
      function(data){
      data.creditCards=data.items;
        for (var i = 0; i < data.creditCards.length; i++) {
          var creditCard = new CreditCard();
          creditCard.populateData(data.creditCards[i]);
          creditCard.isSavedCard(true);
          widget.allCreditCards.push(creditCard);
        }
      },
      function(data) {
        console.log("Error while retrieving the credit cards");
      });
}
```

The widget should also include a function that calls the `updateCreditCard` endpoint to modify card nicknames and to change which card is the default, and a function that calls the `removeCreditCard` endpoint to delete a credit card.

39

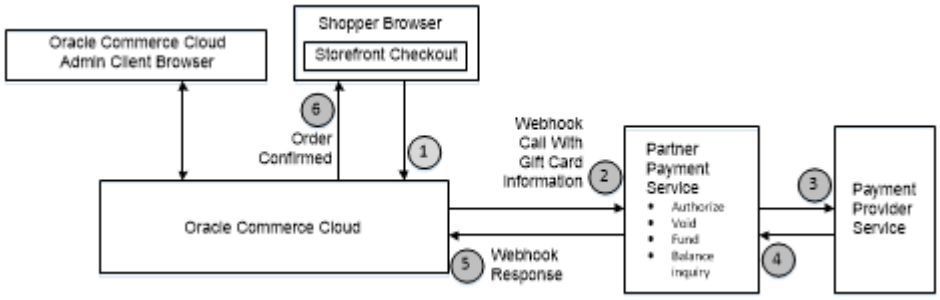
Integrate with a Gift Card Payment Gateway

Oracle Commerce provides support for building integrations with gift card providers.

This section describes with how to integrate with a gift card payment gateway.

Understand the gift card payment gateway workflow

The following diagram illustrates the gift card payment gateway workflow:



Create a gift card extension and configure the webhook

To create a custom integration with a gift card payment gateway, you perform the following steps:

1. Create the gateway extension. See Gift card extension details for information specific to this extension.
2. Upload the extension to the administration interface.
3. Enable the gateway for the sites that require it. Be sure to disable any other gift card payment gateways for those sites.
4. Configure the Generic Payment webhook by specifying the gateway URL and the username and password. Note that webhook settings are not site-specific. The configuration you supply applies to all sites that use this webhook.

Gift card extension details

The format of a payment gateway extension is described in the Create a Credit Card Payment Gateway Integration chapter. For a gift card gateway, the `gateway.json` file should be similar to the following:

```
{
  "provider": "Custom Gift Card",
  "paymentMethodTypes": ["physicalGiftCard"],
  "transactionTypes": {

    "physicalGiftCard": ["balanceInquiry", "authorize", "void", "refund"]
  },
```

```
  "processors" : {
    "physicalGiftCard": "genericGiftCard"
  }
}
```

The `config.json` file should be similar to the following:

```
{
  "configType": "payment",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "instances" : [
    {
      "id": "agent",
      "instanceName": "agent",
      "labelResourceId": "agentInstanceLabel"
    },
    {
      "id": "preview",
      "instanceName": "preview",
      "labelResourceId": "previewInstanceLabel"
    },
    {
      "id": "storefront",
      "instanceName": "storefront",
      "labelResourceId": "storefrontInstanceLabel"
    }
  ],
  "properties": [
    {
      "id": "paymentMethodTypes",
      "type": "multiSelectOptionType",
      "name": "paymentMethodTypes",
      "required": false,
      "helpTextResourceId": "paymentMethodsHelpText",
      "labelResourceId": "paymentMethodsLabel",
      "defaultValue": "physicalGiftCard",
      "displayAsCheckboxes": true,
      "public": true,
      "options": [
          {
           "id": "physicalGiftCard",
           "value": "physicalGiftCard",
           "labelResourceId": "physicalGiftCardPayLabel"
          }
       ]
    },
    {
      "id": "giftCardMaxLength",
      "type": "stringType",
      "name": "giftCardMaxLength",
      "helpTextResourceId": "giftCardMaxLengthHelpText",
      "labelResourceId": "giftCardMaxLengthLabel",
      "required": true,
```

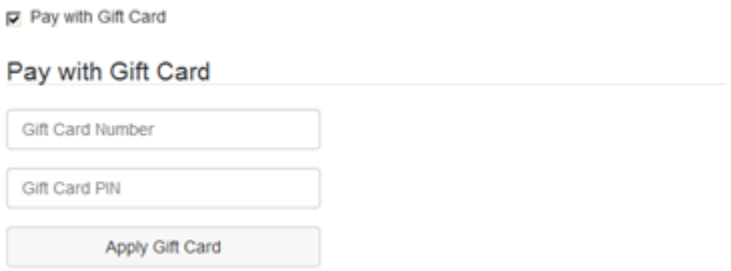
```
      "defaultValue": "19",
      "public": true
    },
    {
      "id": "giftCardPinRequired",
      "type": "booleanType",
      "name": "giftCardPinRequired",
      "helpTextResourceId": "giftCardPinRequiredHelpText",
      "labelResourceId": "giftCardPinRequiredLabel",
      "defaultValue": true,
      "public": true
    },
    {
      "id": "giftCardPinMaxLength",
      "type": "stringType",
      "name": "giftCardPinMaxLength",
      "required": false,
      "helpTextResourceId": "giftCardPinMaxLengthHelpText",
      "labelResourceId": "giftCardPinMaxLengthLabel",
      "defaultValue": "4",
      "public": true
    }
    {
      "id": "includeOrderInWebhookPayload",
      "type": "booleanType",
      "name": "includeOrderInWebhookPayload",
      "helpTextResourceId": "includeOrderHelpText",
      "labelResourceId": "includeOrderLabel",
      "defaultValue": true,
      "public": true
    }
  ]
}
```

The properties in the `config.json` file shown above create controls that appear in the Payment Processing settings in the administration interface. These controls allow the merchant to specify whether a shopper using a gift card is required to supply a PIN, as well as the maximum length of the gift card number and the PIN. The `includeOrderInWebhookPayload` property creates a checkbox for specifying whether or not to include the order data in the webhook call.

Customize the Gift Card widget

By default, the shopper specifies gift card information through the Gift Card widget, which is included on the Checkout Layout with GiftCard.

When the shopper selects Pay with Gift Card on the checkout page, the widget displays fields for entering the gift card number and PIN. For example:



The gateway settings shown in Gift card extension details allow the merchant to specify whether a PIN is required, as well as the maximum length of the gift card number and the PIN. You can also change the behavior of the Gift Card widget by downloading it and customizing it. To download the Gift Card widget as a ZIP file, access the widget template in the **Components** tab in the administration design interface, and click the **Download Source** button. After customizing the new widget, upload it and use it to replace the Gift Card widget in the Checkout Layout with GiftCard. For more information, see Understand widgets.

Gift card payment properties

When the Generic Payment webhook executes, it sends a JSON request body to the payment gateway.

The request body contains information about the order and about the method of payment. The gateway processes the request and returns a JSON response body that contains information about the transaction, including whether the transaction succeeded.

The set of properties in the request and response bodies, including the subobjects, vary depending on the type of transaction. For gift card gateways, there are four transaction types supported: authorize, void, refund, and balance inquiry.

Gift card payment request properties

This section describes the top-level properties and the properties of subobjects sent in the JSON request body of the Generic Payment webhook for gift card transactions. Note that if the `includeOrderInWebhookPayload` property in the gateway extension's `config.json` file is set to `true`, the order is also included in the request. See Order Submit webhook for information about the order properties.

Top-level properties

The following table describes the top-level properties that Oracle Commerce sends in the webhook request.

Property	Description

transactionType	A code indicating the type of transaction. This must be one of the following numeric values:
	
	`0100 (authorize)`
	`0110 (void)`
	`0400 (refund)`
	`0600 (balance inquiry)`

Property	Description

channel	The area of the system where the payment-processing request originated. Valid values are:
	
	`storefront`
	`agent`
	`preview`

orderId	The ID of the order associated with the payment
currencyCode	The ISO 4217 currency code.
locale	The shopper's locale, taken from the order. If no locale is set, the default locale from the storefront is used.
siteId	The ID of the site on which the order was placed
siteURL	The URL of the site on which the order was placed
retryPaymentCount	The number of times payment has been retried for the order
customProperties	Additional key/value pairs from the submitted order to be sent to the provider

`paymentRequests` properties

The following table describes the properties of `paymentRequests` objects that Oracle Commerce sends in the webhook request.

Property	Description
paymentId	The ID of the internal payment group.
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens.
transactionTimestamp	The timestamp of the transaction, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
paymentMethod	The payment method. For a gift card, the value is `physicalGiftCard`.
amount	The expected amount of the transaction. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
gatewayId	The ID of the payment gateway.
customProperties	Additional key/value pairs from the submitted order to be sent to the provider.

`cardDetails` properties

The following table describes the properties of the `cardDetails` object sent in a gift card authorization request or balance inquiry.

Property	Description
giftCardNumber	The number that uniquely identifies the gift card
giftCardPin	The security code for authenticating the gift card

`referenceInfos` properties

The following table describes the properties of the `referenceInfos` objects sent in a gift card void request or refund request. The values of these properties are taken from the original authorization transaction so the merchant can map the void or refund to it.

Property	Description
merchantTransactionId	The transaction reference ID from the merchant
hostTransactionId	The transaction reference ID from the payment gateway

`billingAddress` properties

The following table describes the properties of the `billingAddress` object in the request. The billing address is the address of the shopper to whom the order is charged.

Property	Description
lastName	The last name of the shopper
postalCode	The postal code in the address (for example, the zip code in the United States)
phoneNumber	The phone number associated with the address
email	The email address associated with the address
state	The state in the address
address1	The first line of the address. Typically the street and number.
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order.
firstName	The first name of the shopper
city	The city in the address
country	The country in the address

`shippingAddress` properties

The following table describes the properties of the `shippingAddress` object in the request. The shipping address is the address of the person (not necessarily the shopper) receiving the order.

Property	Description
lastName	The last name of the order recipient
postalCode	The postal code in the address (for example, the zip code in the United States)
phoneNumber	The phone number associated with the address
email	The email address associated with the address
state	The state in the address
address1	The first line of the address. Typically the street and number.
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order.
firstName	The first name of the order recipient
city	The city in the address
country	The country in the address

`profile` properties

The following table describes the properties of the `profile` object in the request. These values are associated with the shopper purchasing the order.

Property	Description
id	The ID of the shopper profile
phoneNumber	The phone number from the shopper profile
email	The email address from the shopper profile

`profileDetails` properties

The following table describes the properties of the `profileDetails` object in the request. These values are associated with the customer purchasing the order. Note that for account-based commerce shoppers, this object may also include `parentOrganization`, `currentOrganization`, and `secondaryOrganizations` subobjects.

Property	Description
id	The ID of the customer profile
lastName	The last name of the customer profile
firstName	The first name of the customer profile
middleName	The middle name of the customer profile
email	The email address from the customer profile
taxExempt	Indicates whether the customer tax-exempt status; either `true` or `false`
taxExemptionCode	For a customer with tax-exempt status, the exemption code
profileType	The type of profile; either `b2c_user` or `b2b_user`
receiveEmail	Indicates whether the customer agrees to receive email; either `yes` or `no`

Property	Description
registrationDate	The timestamp of when the profile was created, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
lastPasswordUpdate	The timestamp of when the password for the profile was last updated, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

Sample authorization request

The following is an example of a gift card authorization request:

```
{
   "transactionType": "0100",
   "currencyCode": "USD",
   "locale": "en",
   "customProperties": { },
   "channel": "storefront",
   "siteId": "siteUS",
   "siteURL": "https://www.example.com",
   "orderId": "o50415",
   "paymentRequests": [
      {
         "transactionId": "o50415-pg50417-1464958982609",
         "paymentId": "pg50417",
         "customProperties": { },
         "gatewaySettings": {
            "paymentMethodTypes":"physicalGiftCard"
         },
         "cardDetails":{
            "giftCardNumber": "12393678",
            "giftCardPin": ""
         },
         "amount":"000000002499",
         "billingAddress": { },
         "transactionTimestamp": "2019-12-03T13:03:02+0000",
         "referenceInfos": { },
         "shippingAddress": { },
         "paymentMethod": "physicalGiftCard",
         "gatewayId": "demoGiftCardGateway",
      }
   ],
   "profile": {
      "id": "120002",
      "phoneNumber": "1234512345",
      "email": "ab@abc.com"
   }
   "profileDetails": {
       "id": "120002",
       "lastName": "Shopper",
       "firstName": "Test",
```

```
        "taxExempt": false,
        "profileType": "b2c_user",
        "receiveEmail": "no",
        "registrationDate": "2019-10-15T06:50:51.000Z",
        "lastPasswordUpdate": "2019-10-15T06:50:51.000Z",
    }
}
```

Gift card payment response properties

This section describes the top-level properties and the properties of subobjects that should be returned in the response body of the Generic Payment webhook for gift card transactions.

Top-level properties

The following table describes the top-level properties that Oracle Commerce expects in the webhook response.

Property	Description

transactionType	A code indicating the type of transaction. This must be one of the following numeric values:
	
	`0100 (authorize)`
	`0110 (void)`
	`0400 (refund)`
	`0600 (balance inquiry)`

currencyCode	The ISO 4217 currency code. This is expected to match the value in the request.
locale	The shopper's locale. This is expected to match the value in the request.
channel	The area of the system where the payment-processing request originated. This is expected to match the value in the request.
orderId	The ID of the order associated with the payment. This is expected to match the value in the request.
siteId	The ID of the site on which the order was placed. Must match the value from the request.

`authorizationResponse`, `voidResponse`, `creditResponse`, and `inquireBalanceResponse` properties

The following table describes the properties of the `authorizationResponse`, `voidResponse`, `creditResponse`, or `inquireBalanceResponse` objects in the webhook response. Only one of these object types is included in each response (the object type corresponding to the transaction type; for example, a `voidResponse` object for a void transaction). All of these object types require the same set of properties. The values of these properties indicate the results of the transaction.

Property	Description

responseCode	The decision from the payment provider as interpreted by the merchant. The acceptable values depend on the transaction type. For an authorization request, the code must be one of the following values:
	
	`1000 (success)`
	`4000 (sale complete)`
	`9000 (decline)`
	
	For a void request, the code must be one of the following values:
	
	`2000 (success)`
	`8000 (decline)`
	
	For a credit (refund) request, the code must be one of the following values:
	
	`3000 (success)`
	`7000 (decline)`
	
	For a balance inquiry, the code must be one of the following values:
	
	`5000 (success)`
	`6000 (decline)`

responseDescription	Information from the payment gateway about the response
responseReason	Information about why the transaction succeeded or failed
hostTransactionId	The transaction reference ID from the payment gateway
merchantTransactionId	The transaction reference ID from the merchant
paymentId	The ID of the internal payment group. Must match the value from the request
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens. Must match the value from the request.
transactionTimestamp	The timestamp of the transaction in Oracle Commerce, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ` Must match the value from the request.

Property	Description
paymentMethod	The payment method. Must match the value from the request. For a gift card, the value is `physicalGiftCard`.
amount	The actual amount of the transaction. This may differ from the amount in the request. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
merchantTransactionTimeStamp	The timestamp of the transaction from the merchant (in milliseconds)
hostTransactionTimeStamp	The timestamp of the transaction from the gateway (in milliseconds)
gatewayId	The ID of the payment gateway. Must match the value from the request
additionalProperties	Key/value pairs for additional properties sent by the merchant

Sample authorization response

The following is an example of a response to a gift card authorization request:

```
{
   "transactionType": "0100",
   "currencyCode": "USD",
   "locale": "en",
   "channel": "storefront",
   "siteId": "siteUS",
   "orderId": "o50415",
   "authorizationResponse":
      [{
         "merchantTransactionTimestamp": "1464958982654",
         "responseCode": "1000",
         "hostTransactionId": "hID1464958982554",
         "transactionId": "o50415-pg50417-1464958982609",
         "paymentId": "pg50417",
         "responseDescription": "AuthResponseDescription",
         "merchantTransactionId": "mID1464958982654v",
         "amount": "000000002999",
         "additionalProperties": {
            "sample-addnl-property-key4": "sample-payment-property-value4",
            "sample-addnl-property-key2": "sample-payment-property-value2",
            "sample-addnl-property-key3": "sample-payment-property-value3",
            "sample-addnl-property-key1": "sample-payment-property-value1"
         },
         "hostTransactionTimestamp": "1464958982554",
         "responseReason": "AuthResponseReason",
         "transactionTimestamp": "2019-12-03T13:03:02+0000",
         "paymentMethod": "physicalGiftCard",
         "gatewayId": "demoGiftCardGateway"
```

```
            }]
        }
```

40

Integrate with a Store Credit Payment Gateway

Oracle Commerce provides support for integrating with store credit systems. Individual shoppers can pay for items using store credits that they have accumulated.

Important: Store credit is not available as a payment option for account-based shoppers.

This section describes how to integrate with a gateway for paying with store credits.

Create a store credit extension and configure the webhook

To create a custom integration with a store credit payment gateway, you perform the following steps:

1. Create the gateway extension. See Store credit extension details for information specific to this extension.
2. Upload the extension to the administration interface.
3. Enable the gateway for the sites that require it.
4. Add a store credit payment option to the checkout page.
5. Configure the Generic Payment webhook by specifying the gateway URL and the username and password. Note that webhook settings are not site-specific. The configuration you supply applies to all sites that use this webhook.

Store credit extension details

The format of a payment gateway extension is described in the Create a Credit Card Payment Gateway Integration chapter. For a store credit gateway, the `gateway.json` file should be similar to the following:

```
{
  "provider": "Store Credits Payment Gateway",
  "paymentMethodTypes": ["storeCredit"],
  "transactionTypes": {
    "storeCredit": ["balanceInquiry", "authorize", "void", "refund"]
  }
}
```

The `config.json` file should be similar to the following:

```
{
  "configType": "payment",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "instances" : [
    {
```

```
      "id": "agent",
      "instanceName": "agent",
      "labelResourceId": "agentInstanceLabel"
    },
    {
      "id": "preview",
      "instanceName": "preview",
      "labelResourceId": "previewInstanceLabel"
    },
    {
      "id": "storefront",
      "instanceName": "storefront",
      "labelResourceId": "storefrontInstanceLabel"
    }
  ],
  "properties": [
    {
      "id": "paymentMethodTypes",
      "type": "multiSelectOptionType",
      "name": "paymentMethodTypes",
      "required": false,
      "helpTextResourceId": "paymentMethodsHelpText",
      "labelResourceId": "paymentMethodsLabel",
      "defaultValue": "storeCredit",
      "displayAsCheckboxes": true,
      "public": true,
      "options": [
         {
           "id": "storeCredit",
           "value": "storeCredit",
           "labelResourceId": "storeCreditPayLabel"
         }
       ]
    },
    {
      "id": "includeOrderInWebhookPayload",
      "type": "booleanType",
      "name": "includeOrderInWebhookPayload",
      "helpTextResourceId": "includeOrderHelpText",
      "labelResourceId": "includeOrderLabel",
      "defaultValue": true,
      "public": true
    }
  ]
}
```

Currency and store credit

Commerce requests the store credit authorization in the order currency value, but it does not convert the order currency to store credits or conversely. The merchant ERP system should return the store credit equivalent value for the requested currency amount. For example, Commerce requests the store credit authorization in $150.00, and the store credit balance is 1500. Only the merchant ERP system can decide the dollar amount of 1500 store credits and whether it is more or less than $150.00. If it is more than $150, the merchant ERP system should return a success response code

along with the remaining store credit balance. Otherwise it should return a decline response code.

Add a Store Credit payment option to the checkout page

To enable paying through store credits, you need to add a store credit payment option to the checkout page:

1. Create a store credit payment widget. Oracle does not provide a ready-to-use widget for store credit.
2. Open the Checkout Layout that you are using on your storefront. (The default is Checkout Layout with GiftCard.)
3. Switch to grid view.
4. Add the store credit payment widget to the layout.
5. Publish your changes.

Store credit payment properties

When the Generic Payment webhook executes, it sends a JSON request body to the payment gateway.

The request body contains information about the order, the method of payment, and the type of transaction being initiated. The gateway processes the request and returns a JSON response body that contains information about the results of the transaction, including whether the transaction succeeded.

The set of properties in the request and response bodies, including the subobjects, vary depending on the type of transaction. For store credit gateways, there are four transaction types supported: authorize, void, refund, and balance inquiry.

Store credit payment request properties

This section describes the top-level properties and the properties of subobjects sent in the JSON request body of the Generic Payment webhook for store credit transactions. Note that if the `includeOrderInWebhookPayload` property in the gateway extension's `config.json` file is set to `true`, the order is also included in the request. See Order Submit webhook for information about the order properties.

Top-level properties

The following table describes the top-level properties that Oracle Commerce sends in the webhook request.

Property	Description

transactionType	A code indicating the type of transaction. This must be one of the following numeric values:
	
	`0100 (authorize)`
	`0110 (void)`
	`0400 (refund)`
	`0600 (balance inquiry)`

Property	Description

channel	The area of the system where the payment-processing request originated. Valid values are:
	
	`storefront`
	`agent`
	`preview`

orderId	The ID of the order associated with the transaction.
currencyCode	The ISO 4217 currency code.
locale	The shopper's locale, taken from the order. If no locale is set, the default locale from the storefront is used.
customProperties	Additional key/value pairs to be sent to the payment provider.
siteURL	The URL of the site on which the order was placed.
siteId	The ID of the site on which the order was placed.
retryPaymentCount	The number of times payment has been retried for the order.

`paymentRequests` properties

The following table describes the properties of `paymentRequests` objects that Oracle Commerce sends in the webhook request.

Property	Description
paymentId	The ID of the internal payment group.
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens.
transactionTimestamp	The timestamp of the transaction, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
paymentMethod	The payment method. For store credit, the value is `storeCredit`.
amount	The expected amount of the transaction. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
gatewayId	The ID of the payment gateway.
customProperties	Additional key/value pairs from the submitted order to be sent to the provider.

`profile` properties

The following table describes the properties of the `profile` object included in the request. These values are associated with the shopper purchasing the order.

Property	Description
id	The Commerce ID of the shopper profile
phoneNumber	The phone number from the shopper profile
email	The email address from the shopper profile
dynamicProperties	The shopper profile dynamic properties if configured

`profileDetails` properties

The following table describes the properties of the `profileDetails` object in the request. These values are associated with the customer purchasing the order.

Property	Description
id	The ID of the customer profile
lastName	The last name of the customer profile
firstName	The first name of the customer profile
middleName	The middle name of the customer profile
email	The email address from the customer profile
taxExempt	Indicates whether the customer tax-exempt status; either `true` or `false`
taxExemptionCode	For a customer with tax-exempt status, the exemption code
profileType	The type of profile; either `b2c_user` or `b2b_user`
receiveEmail	Indicates whether the customer agrees to receive email; either `yes` or `no`
registrationDate	The timestamp of when the profile was created, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
lastPasswordUpdate	The timestamp of when the password for the profile was last updated, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

`storeCredit` properties

The following table describes the property of the `storeCredit` objects sent in an authorization request or balance inquiry.

Property	Description
storeCreditNumber	The number that uniquely identifies the store credit. For balance inquiries, if a store credit number is passed in the request, the amount specific to that store credit is sent back. Otherwise, all the store credits associated with the shopper profile are sent back.

`referenceInfos` properties

The following table describes the properties of the `referenceInfos` objects sent in a void request or refund request. The values of these properties are taken from the original authorization transaction so the merchant can map the void or refund to it.

Property	Description
merchantTransactionId	The transaction reference ID from the merchant.
hostTransactionId	The transaction reference ID from the payment gateway.

`billingAddress` properties

The following table describes the properties of the `billingAddress` object in an authorization request. The billing address is the address of the shopper to whom the order is charged.

Property	Description
lastName	The last name of the shopper.
postalCode	The postal code in the address (for example, the zip code in the United States).
phoneNumber	The phone number associated with the address.
email	The email address associated with the address.
state	The state in the address.
address1	The first line of the address. Typically the street and number.
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order.
firstName	The first name of the shopper.
city	The city in the address.
country	The country in the address.

`shippingAddress` properties

The following table describes the properties of the `shippingAddress` object in a request. The shipping address is the address of the person (not necessarily the shopper) receiving the order.

Property	Description
lastName	The last name of the order recipient.
postalCode	The postal code in the address (for example, the zip code in the United States).
phoneNumber	The phone number associated with the address.
email	The email address associate with the address.
state	The state in the address.
address1	The first line of the address. Typically the street and number.

Property	Description
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order.
firstName	The first name of the order recipient.
city	The city in the address.
country	The country in the address.

Sample balance inquiry request

The following is an example of a store credit balance inquiry request:

```
{
  "orderId": "o78615",
  "profile": {
    "phoneNumber": "617-555-1977",
    "id": "se-570031",
    "email": "john@example.com"
    "dynamicProperties": [
      {
        "label": "Nickname",
        "id": "field1",
        "value": "Jack"
      }
    ]
},
  "channel": "agent",
  "locale": "en",
  "transactionId": "f4dd73a8-d722-407d-9d9d-
e9db11a68ace-00e09e08-171e-4a83-8e75-96132ad61166-1509528103666",
  "transactionTimestamp": "2018-01-06T09:21:43+0000",
  "transactionType": "0600",
  "customProperties": null,
  "paymentId": "00e09e08-171e-4a83-8e75-96132ad61166",
  "gatewaySettings": {
    "paymentMethodTypes": "storeCredit"
  },
  "paymentMethod": "storeCredit",
  "shippingAddress": null,
  "siteId": "siteUS",
  "currencyCode": "USD",
  "gatewayId": "storeCreditPaymentGateway"
}
```

Sample authorization request

The following is an example of a store credit authorization request:

```
{
  "amount": "000000007490",
  "orderId": "o150425",
  "profile": {
    "phoneNumber": "617-555-1977",
```

```
  "id": "se-570031",
  "email": "john@example.com"
  "dynamicProperties": [
    {
      "label": "Nickname",
      "id": "field1",
      "value": "Jack"
    }
  ]
},
"profileDetails": {
     "id": "se-570031",
     "lastName": "Shopper",
     "firstName": "John",
     "taxExempt": false,
     "profileType": "b2c_user",
     "receiveEmail": "no",
     "registrationDate": "2019-10-15T06:50:51.000Z",
     "lastPasswordUpdate": "2019-10-15T06:50:51.000Z",
}
"channel": "agent",
"locale": "en",
"siteURL": "https://www.example.com",
"transactionId": "o150425-pg150422-1509433854097",
"transactionTimestamp": "2019-12-07T07:10:54+0000",
"transactionType": "0100",
"paymentId": "pg150422",
"gatewaySettings": {
  "paymentMethodTypes": "storeCredit"
},
"paymentMethod": "storeCredit",
"shippingAddress": {
  "lastName": "Niel",
  "country": "US",
  "firstName": "John",
  "phoneNumber": "617-555-1977",
  "address2": null,
  "city": "Cambridge",
  "address1": "1 Main St",
  "postalCode": "02142",
  "state": "MA",
  "email": "john@example.com"
},
"siteId": "siteUS",
"billingAddress": {
  "country": "US",
  "lastName": "Niel",
  "firstName": "John",
  "phoneNumber": "617-555-1977",
  "city": "San Francisco",
  "address1": "1 Elm St",
  "postalCode": "91333",
  "state": "CA",
  "email": "john.niel@gmail.com"
},
```

```
  "retryPaymentCount": 0,
  "currencyCode": "USD",
  "gatewayId": "storeCreditPaymentGateway"
}
```

Store credit payment response properties

This section describes the top-level properties and the properties of subobjects that should be returned in the response body of the Generic Payment webhook for store credit transactions.

Top-level properties

The following table describes the top-level properties that should be returned in the response body of the Generic Payment webhook for store credit transactions.

Property	Description

transactionType	A code indicating the type of transaction. This must be one of the following numeric values, and is expected to match the value in the request:
	
	`0100 (authorize)`
	`0110 (void)`
	`0400 (refund)`
	`0600 (balance inquiry)`

currencyCode	The ISO 4217 currency code. This is expected to match the value in the request.
locale	The shopper's locale. This is expected to match the value in the request.
channel	The area of the system where the payment-processing request originated. This is expected to match the value in the request.
orderId	The ID of the order associated with the payment. This is expected to match the value in the request.
siteId	The ID of the site on which the order was placed. Must match the value from the request.

`authorizationResponse`, `voidResponse`, `creditResponse`, and `inquireBalanceResponse` properties

The following table describes the properties of the `authorizationResponse`, `voidResponse`, `creditResponse`, or `inquireBalanceResponse` object in the webhook response. Only one of these objects is included in each response (the object corresponding to the transaction type; for example, a `voidResponse` object for a void transaction). All of these object types require the same set of properties. The values of these properties indicate the results of the transaction.

Property	Description
totalAvailableAmount	The total amount available in all store credits

Property	Description

responseCode	The decision from the payment provider as interpreted by the merchant. The acceptable values depend on the transaction type. For an authorization request, the code must be one of the following values:
	
	`1000 (success)`
	`4000 (sale complete)`
	`9000 (decline)`
	
	For a void request, the code must be one of the following values:
	
	`2000 (success)`
	`8000 (decline)`
	
	For a credit (refund) request, the code must be one of the following values:
	
	`3000 (success)`
	`7000 (decline)`
	
	For a balance inquiry, the code must be one of the following values:
	
	`5000 (success)`
	`6000 (decline)`

responseDescription	Information from the payment gateway about the response.
responseReason	Information about why the transaction succeeded or failed.
hostTransactionId	The transaction reference ID from the payment gateway.
merchantTransactionId	The transaction reference ID from the merchant.
paymentId	The ID of the internal payment group. Must match the value from the request.
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens. Must match the value from the request.
transactionTimestamp	The timestamp of the transaction in Oracle Commerce, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ` Must match the value from the request.
paymentMethod	The payment method. Must match the value from the request. For store credit, the value is `storeCredit`.

Property	Description
amount	The actual amount of the transaction. This may differ from the amount in the request. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
merchantTransactionTimeStamp	The timestamp of the transaction from the merchant (in milliseconds).
hostTransactionTimeStamp	The timestamp of the transaction from the gateway (in milliseconds).
additionalProperties	Key/value pairs for additional properties sent by the merchant.
customPaymentProperties	Keys from the `additionalProperties`.
storeCredits	Store credit details of the shopper profile.

`storeCredit` properties

The following table describes the properties of the objects in the `storeCredits` array.

Property	Description
storeCreditNumber	The number that uniquely identifies the store credit.
availableAmount	Amount available in each store credit

Sample balance inquiry response

The following is an example of a response to a store credit balance inquiry request:

```
{
  "totalAvailableAmount": "500",
  "transactionType": "0600",
  "orderId": "o78615",
  "paymentId": "00e09e08-171e-4a83-8e75-96132ad61166",
  "channel": "agent",
  "paymentMethod": "storeCredit",
  "siteId": "siteUS",
  "locale": "en",
  "inquireBalanceResponse": {
    "hostTransactionTimestamp": "1509528105863",
    "responseReason": "inquireBalanceResponseReason",
    "storeCredits": [
      {
        "storeCreditNumber": "4123654789",
        "availableAamount": "100"
      },
      {
        "storeCreditNumber": "4123654790",
        "availableAamount": "200"
      },
      {
        "storeCreditNumber": "4123654791",
```

```
        "availableAamount": "200"
     }
],
    "customPaymentProperties": [
      "5000addnl-property-key5",
      "5000addnl-property-key2"
    ],
    "responseDescription": "inquireBalanceResponseDescription",
    "merchantTransactionId": "MERCH-TX-1509528105863",
    "hostTransactionId": "HOST-TX-1509528105863",
    "additionalProperties": {
      "5000addnl-property-key5": "5000payment-property-value5",
      "5000addnl-property-key4": "5000payment-property-value4",
      "5000addnl-property-key3": "5000payment-property-value3",
      "5000addnl-property-key2": "5000payment-property-value2",
      "5000addnl-property-key1": "5000payment-property-value1",
    },
    "responseCode": "5000",
    "merchantTransactionTimestamp": "1509528105863"
  },
  "currencyCode": "USD",
  "transactionId": "f4dd73a8-d722-407d-9d9d-
e9db11a68ace-00e09e08-171e-4a83-8e75-96132ad61166-1509528103666",
  "transactionTimestamp": "2018-01-06T09:55:22+0000",
  "gatewayId": "storeCreditPaymentGateway"
}
```


Sample authorization response

The following is an example of a response to a store credit authorization request:


```
{
  "amount": "000000007490",
  "orderId": "o150425",
  "channel": "agent",
  "authorizationResponse": {
    "hostTransactionTimestamp": "1509433854723",
    "responseReason": "authResponseReason",
    "responseDescription": "authResponseDescription",
    "merchantTransactionId": "MERCH-TX-1509433854723",
    "hostTransactionId": "HOST-TX-1509433854723",
    "additionalProperties": {
      "1000addnl-property-key5": "1000payment-property-value5",
      "1000addnl-property-key4": "1000payment-property-value4",
      "1000addnl-property-key1": "1000payment-property-value1",
      "1000addnl-property-key3": "1000payment-property-value3",
      "1000addnl-property-key2": "1000payment-property-value2",
    },
    "responseCode": "1000",
    "merchantTransactionTimestamp": "1509433854723"
  },
  "locale": "en",
  "transactionId": "o150425-pg150422-1509433854097",
  "transactionTimestamp": "2018-01-06T09:47:48+0000",
  "transactionType": "0100",
```

```
  "paymentId": "pg150422",
  "paymentMethod": "storeCredit",
  "siteId": "siteUS",
  "currencyCode": "USD",
  "gatewayId": "storeCreditPaymentGateway"
}
```

41

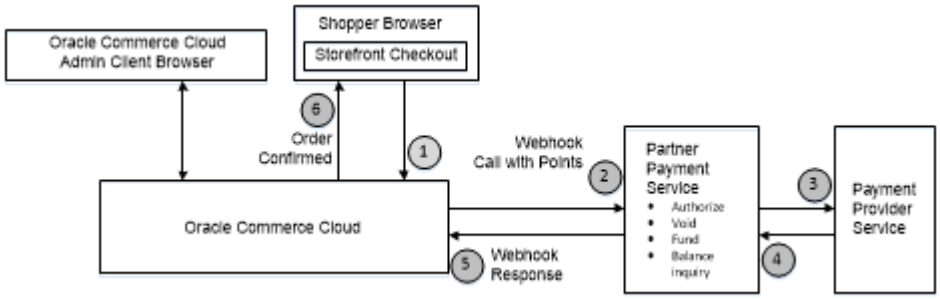
Integrate with a Loyalty Point Payment Gateway

Shoppers who are enrolled in supported loyalty programs can pay for items using loyalty points that they have accumulated in those programs.

This section describes how to integrate with a gateway for paying with loyalty points. See Work with Loyalty Programs for more information about integrating with loyalty programs.

Understand the loyalty point payment gateway workflow

The following diagram illustrates the loyalty point payment gateway workflow:



Create a loyalty point extension and configure the webhook

To create a custom integration with a loyalty point payment gateway, you must create a loyalty point extension and configure the Custom Currency webhook.

To create the integration, you perform the following steps:

1. Create the gateway extension, as described below.
2. Upload the extension to the administration interface.
3. Enable the gateway for the sites that require it.
4. Add a loyalty point payment option to the checkout page.
5. Configure the Custom Currency webhook by specifying the gateway URL and the username and password. Note that webhook settings are not site-specific. The configuration you supply applies to all sites that use this webhook.

Loyalty point extension details

The format of a payment gateway extension is described in the Create a Credit Card Payment Gateway Integration chapter. For a loyalty point gateway, the `gateway.json` file should be similar to the following:

```
{
  "provider": "Loyalty Points Payment Gateway",
  "paymentMethodTypes": ["loyaltyPoints"],
  "transactionTypes": {
    "loyaltyPoints": ["balanceInquiry", "authorize", "void", "refund"]
  },
  "processors" : {
    "loyaltyPoints": "loyaltyPoints"
  }
}
```

The `config.json` file should be similar to the following:

```
{
  "configType": "payment",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "instances" : [
    {
      "id": "agent",
      "instanceName": "agent",
      "labelResourceId": "agentInstanceLabel"
    },
    {
      "id": "preview",
      "instanceName": "preview",
      "labelResourceId": "previewInstanceLabel"
    },
    {
      "id": "storefront",
      "instanceName": "storefront",
      "labelResourceId": "storefrontInstanceLabel"
    }
  ],
  "properties": [
    {
      "id": "paymentMethodTypes",
      "type": "multiSelectOptionType",
      "name": "paymentMethodTypes",
      "required": false,
      "helpTextResourceId": "paymentMethodsHelpText",
      "labelResourceId": "paymentMethodsLabel",
      "defaultValue": "loyaltyPoints",
      "displayAsCheckboxes": true,
      "public": true,
      "options": [
        {
```

```
            "id": "loyaltyPoints",
            "value": "loyaltyPoints",
            "labelResourceId": "loyaltyPointsLabel"
          }
        ]
    },
    {
      "id": "includeOrderInWebhookPayload",
      "type": "booleanType",
      "name": "includeOrderInWebhookPayload",
      "helpTextResourceId": "includeOrderHelpText",
      "labelResourceId": "includeOrderLabel",
      "defaultValue": true,
      "public": true
    }
  ]
}
```

Add a loyalty point payment option to the checkout page

To enable paying through loyalty points, you need to add a loyalty point payment option to the checkout page:

1. Open the Checkout Layout that you are using on your storefront. (The default is Checkout Layout with GiftCard.)
2. Switch to grid view.
3. Add the Loyalty Payment widget to the layout.
4. Publish your changes.

Note that in addition to the Loyalty Payment widget, Commerce includes a Loyalty Details widget that displays information about loyalty programs the shopper is enrolled in and the shopper's current point totals. You may want to add the Loyalty Details widget to your checkout page or to another page, such as the shopper profile page.

Loyalty point payment properties

When the Custom Currency webhook executes, it sends a JSON request body to the payment gateway.

The request body contains information about the order, the method of payment, and the type of transaction being initiated. The gateway processes the request and returns a JSON response body that contains information about the results of the transaction, including whether the transaction succeeded.

The set of properties in the request and response bodies, including the subobjects, vary depending on the type of transaction. For loyalty point gateways, there are four transaction types supported: authorize, void, refund, and balance inquiry.

Loyalty point payment request properties

This section describes the top-level properties and the properties of subobjects sent in the JSON request body of the Custom Currency webhook for loyalty point transactions. Note that if the `includeOrderInWebhookPayload` property in the gateway extension's `config.json` file is

set to `true`, the order is also included in the request (except for balance inquiry requests, which do not have an associated order). See Order Submit webhook for information about the order properties.

Top-level properties

The following table describes the top-level properties that Oracle Commerce sends in the webhook request.

Property	Description

transactionType	A code indicating the type of transaction. This must be one of the following numeric values:
	
	`0100 (authorize)`
	`0110 (void)`
	`0400 (refund)`
	`0600 (balance inquiry)`

channel	The area of the system where the payment-processing request originated. Valid values are:
	
	`storefront`
	`agent`
	`review`

orderId	The ID of the order associated with the transaction
currencyCode	The custom currency code used for the loyalty program. See Create a custom currency for loyalty points for information about this value.
locale	The shopper's locale, taken from the order. If no locale is set, the default locale from the storefront is used.
customProperties	Additional key/value pairs to be sent to the payment provider
paymentId	The ID of the internal payment group
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens.
transactionTimestamp	The timestamp of the transaction, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
paymentMethod	The payment method. For loyalty points, the value is `loyaltyPoints`.
amount	For authorize, void, and refund transactions, the number of loyalty points, expressed as a positive 12-digit number.
retryPaymentCount	The number of times payment has been retried for the order
siteURL	The URL of the site on which the order is placed

Property	Description
siteId	The ID of the site on which the order is placed
gatewayId	The ID of the payment gateway
gatewaySettings	Key/value pairs of properties that configure the payment gateway

`profile` properties

The following table describes the properties of the `profile` object included in the request. These values are associated with the shopper purchasing the order.

Property	Description
id	The Commerce ID of the shopper profile
phoneNumber	The phone number from the shopper profile
email	The email address from the shopper profile

`profileDetails` properties

The following table describes the properties of the `profileDetails` object in the request. These values are associated with the customer purchasing the order.

Property	Description
id	The ID of the customer profile
lastName	The last name of the customer profile
firstName	The first name of the customer profile
middleName	The middle name of the customer profile
email	The email address from the customer profile
taxExempt	Indicates whether the customer tax-exempt status; either `true` or `false`
taxExemptionCode	For a customer with tax-exempt status, the exemption code
profileType	The type of profile; either `b2c_user` or `b2b_user`
receiveEmail	Indicates whether the customer agrees to receive email; either `yes` or `no`
registrationDate	The timestamp of when the profile was created, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
lastPasswordUpdate	The timestamp of when the password for the profile was last updated, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

`loyaltyDetails` properties

The following table describes the properties of the `loyaltyDetails` objects sent in an authorization request or balance inquiry. These values are retrieved from the shopper's profile.

Property	Description
programName	The name of the loyalty program
programId	The ID for the loyal program
membershipId	The shopper's membership ID in the loyalty program

status	The shopper's status in the loyalty program. Valid values are:
	
	`RequestForEnrollment`
	`Enrolled`
	`RequestForUnenrollment`
	`Unenrolled`
	`Failed`

`referenceInfos` properties

The following table describes the properties of the `referenceInfos` objects sent in a void request or refund request. The values of these properties are taken from the original authorization transaction so the merchant can map the void or refund to it.

Property	Description
amount	The number of loyalty points used in the original transaction, expressed as a positive 12-digit number.
currencyCode	The custom currency code used for the loyalty program
locale	The shopper's locale from the original transaction
merchantTransactionId	The transaction reference ID from the merchant
hostTransactionId	The transaction reference ID from the payment gateway

`billingAddress` properties

The following table describes the properties of the `billingAddress` object in an authorization request. The billing address is the address of the shopper to whom the order is charged.

Property	Description
lastName	The last name of the shopper
postalCode	The postal code in the address (for example, the zip code in the United States)
phoneNumber	The phone number associated with the address
email	The email address associated with the address
state	The state in the address

Property	Description
address1	The first line of the address. Typically the street and number
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order
firstName	The first name of the shopper
city	The city in the address
country	The country in the address

`shippingAddress` properties

The following table describes the properties of the `shippingAddress` object in an authorization or balance inquiry request. The shipping address is the address of the person (not necessarily the shopper) receiving the order.

Property	Description
lastName	The last name of the order recipient
postalCode	The postal code in the address (for example, the zip code in the United States)
phoneNumber	The phone number associated with the address
email	The email address associated with the address
state	The state in the address
address1	The first line of the address. Typically the street and number.
address2	The second line of the address. Included as an empty string in the JSON data if no value exists in the order.
firstName	The first name of the order recipient
city	The city in the address
country	The country in the address

Sample balance inquiry request

The following is an example of a loyalty point balance inquiry request:

```
{
  "loyaltyDetails": [
    {
      "programName": "programForFlyer",
      "membershipId": "member0002",
      "programId": "prg10002",
      "status": "Enrolled"
    },
    {
      "programName": "programForYoungster",
      "membershipId": "member0001",
      "programId": "prg10001",
      "status": "Enrolled"
    }
  ],
```

```
  "orderId": "o30417",
  "profile": {
    "phoneNumber": "1234512345",
    "id": "110072",
    "email": "john@example.com"
  },
  "channel": "storefront",
  "locale": "en",
  "transactionId": "o30417-pg30418-1504691722253",
  "transactionTimestamp": "2017-09-06T09:55:22+0000",
  "transactionType": "0600",
  "customProperties": {
    "cust-prop2": "cust-prop2",
    "cust-prop1": "cust-prop1"
  },
  "paymentId": "pg30418",
  "gatewaySettings": {
    "paymentMethodTypes": "loyaltyPoints"
  },
  "paymentMethod": "loyaltyPoints",
  "shippingAddress": {
    "lastName": "Niel",
    "country": "US",
    "firstName": "John",
    "phoneNumber": "1234512345",
    "address2": null,
    "city": "Cambridge",
    "address1": "1 Main St",
    "postalCode": "02142",
    "state": "MA",
    "email": "john@example.com"
  },
  "siteId": "siteUS",
  "currencyCode": "PTS",
  "gatewayId": "demoLoyaltyPointsPaymentGateway"
}
```


Sample authorization request

The following is an example of a loyalty point authorization request:


```
{
  "loyaltyDetails": [
    {
      "programName": "programForYoungster",
      "membershipId": "member0001",
      "programId": "prg10001",
      "status": "Enrolled"
    },
    {
      "programName": "programForFlyer",
      "membershipId": "member0002",
      "programId": "prg10002",
      "status": "Enrolled"
    }
```

```
],
"amount": "000000000200",
"orderId": "o30417",
"profile": {
  "phoneNumber": "1234512345",
  "id": "110072",
  "email": "john@example.com"
},
"profileDetails": {
  "id": "110072",
  "lastName": "Shopper",
  "firstName": "Test",
  "taxExempt": false,
  "profileType": "b2c_user",
  "receiveEmail": "no",
  "registrationDate": "2019-10-15T06:50:51.000Z",
  "lastPasswordUpdate": "2019-10-15T06:50:51.000Z",
 }
"channel": "storefront",
"locale": "en",
"siteURL": "https://www.example.com",
"retryPaymentCount": 0,
"transactionId": "o30417-pg30415-1504691268818",
"transactionTimestamp": "2019-12-06T09:47:48+0000",
"transactionType": "0100",
"customProperties": {
  "cust-prop2": "cust-prop2",
  "cust-prop1": "cust-prop1"
},
"paymentId": "pg30415",
"gatewaySettings": {
  "paymentMethodTypes": "loyaltyPoints"
},
"paymentMethod": "loyaltyPoints",
"shippingAddress": {
  "lastName": "Niel",
  "country": "US",
  "firstName": "John",
  "phoneNumber": "1234512345",
  "address2": null,
  "city": "Cambridge",
  "address1": "1 Main St",
  "postalCode": "02142",
  "state": "MA",
  "email": "john@example.com"
},
"siteId": "siteUS",
"billingAddress": {
  "country": "US",
  "lastName": "John",
  "firstName": "Niel",
  "phoneNumber": "9000054321",
  "city": "San Francisco",
  "address1": "1 Elm St",
  "postalCode": "91333",
```

```
    "state": "CA",
    "email": "john.niel@gmail.com"
  },
  "currencyCode": "PTS",
  "gatewayId": "demoLoyaltyPointsPaymentGateway"
}
```

Loyalty point payment response properties

This section describes the top-level properties and the properties of subobjects that should be returned in the response body of the Custom Currency webhook for loyalty point transactions.

Top-level properties

The following table describes the top-level properties that Oracle Commerce expects in the webhook response.

Property	Description

transactionType	A code indicating the type of transaction. This must be one of the following numeric values, and is expected to match the value in the request:
	
	`0100 (authorization)`
	`0110 (void)`
	`0400 (refund)`
	`0600 (balance inquiry)`

currencyCode	The custom currency code used for the loyalty program. This is expected to match the value in the request.
locale	The shopper's locale. This is expected to match the value in the request.
channel	The area of the system where the payment-processing request originated. This is expected to match the value in the request.
orderId	The ID of the order associated with the payment. This is expected to match the value in the request.
paymentId	The ID of the internal payment group. Must match the value from the request.
paymentMethod	The payment method. Must match the value from the request. For loyalty points, the value is `loyaltyPoints`.
transactionId	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens. Must match the value from the request.
amount	The number of loyalty points used in the transaction, expressed as a positive 12-digit number.

Property	Description
transactionTimestamp	The timestamp of the transaction in Oracle Commerce, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ` Must match the value from the request.
siteId	The ID of the site on which the order was placed. Must match the value from the request.
gatewayId	The ID of the payment gateway. Must match the value from the request.

`authorizationResponse`, `voidResponse`, `creditResponse`, and `inquireBalanceResponse` properties

The following table describes the properties of the `authorizationResponse`, `voidResponse`, `creditResponse`, or `inquireBalanceResponse` object in the webhook response. Only one of these objects is included in each response (the object corresponding to the transaction type; for example, a `voidResponse` object for a void transaction). All of these object types require the same set of properties. The values of these properties indicate the results of the transaction.

Property	Description

responseCode	The decision from the payment provider as interpreted by the merchant. The acceptable values depend on the transaction type. For an authorization request, the code must be one of the following values:
	
	`1000 (success)`
	`4000 (sale complete)`
	`9000 (decline)`
	
	For a void request, the code must be one of the following values:
	
	`2000 (success)`
	`8000 (decline)`
	
	For a credit (refund) request, the code must be one of the following values:
	
	`3000 (success)`
	`7000 (decline)`
	
	For a balance inquiry, the code must be one of the following values:
	
	`5000 (success)`
	`6000 (decline)`

responseDescription	Information from the payment gateway about the response
responseReason	Information about why the transaction succeeded or failed
hostTransactionId	The transaction reference ID from the payment gateway
merchantTransactionId	The transaction reference ID from the merchant
merchantTransactionTimeStamp	The timestamp of the transaction from the merchant (in milliseconds)
hostTransactionTimeStamp	The timestamp of the transaction from the gateway (in milliseconds)
additionalProperties	Key/value pairs for additional properties sent by the merchant. (For an `authorizationResponse` object, we recommend including the `programId`, `programName`, and `membershipId` properties, so they can be included in any subsequent void or refund webhook requests.)
customPaymentProperties	A list of the properties from the `additionalProperties` map that should be returned to the storefront

`loyaltyPrograms` properties

The following table describes the properties of the objects in the `loyaltyPrograms` array of an `inquireBalanceResponse` object.

Property	Description
membershipType	The category or level of the membership; for example, silver, gold, or platinum
pointsBalance	The number of points available
pointsType	The type of points. This should match the currencyCode value described in Create a custom currency for loyalty points.
additionalProperties	Key/value pairs for additional properties sent by the merchant for this program type

`loyaltyPointDetails` properties

The following table describes the properties of the objects in the `loyaltyPointDetails` array of a `loyaltyPrograms` object of an `inquireBalanceResponse` object.

Property	Description
programName	The name of the loyalty program
programId	The ID of the loyalty program
membershipId	The shopper's ID in the loyalty program
additionalProperties	Key/value pairs for additional properties sent by the merchant for this point type

status	The status of the shopper in the loyalty program. Valid values are:
	
	`RequestForEnrollment`
	`Enrolled`
	`RequestForUnenrollment`
	`Unenrolled`
	`Failed`

Sample balance inquiry response

The following is an example of a response to a loyalty point balance inquiry request:

```
{
  "transactionType": "0600",
  "orderId": "o30417",
  "paymentId": "pg30418",
  "channel": "storefront",
  "paymentMethod": "loyaltyPoints",
  "siteId": "siteUS",
  "locale": "en",
  "inquireBalanceResponse": {
    "hostTransactionTimestamp": "1504691722267",
    "responseReason": "inquireBalanceResponseReason",
    "customPaymentProperties": [
```

```
    "cust-prop2",
    "cust-prop1"
  ],
  "responseDescription": "inquireBalanceResponseDescription",
  "loyaltyPrograms": [
    {
      "loyaltyPointDetails": [
        {
          "membershipType": "blue",
          "pointsBalance": "2000",
          "pointsType": "bluePoints",
          "additionalProperties": {
            "propertyName1": "value1",
            "propertyName2": "value2"
          }
        },
        {
          "membershipType": "red",
          "pointsBalance": "6000",
          "pointsType": "redPoints",
          "additionalProperties": {
            "propertyName4": "value4",
            "propertyName3": "value3"
          }
        },
        {
          "membershipType": "standard",
          "pointsBalance": "6000",
          "pointsType": "PTS",
          "additionalProperties": {
            "propertyName6": "value6",
            "propertyName5": "value5"
          }
        }
      ],
      "programName": "programForYoungster",
      "membershipId": "member0001",
      "additionalProperties": {
        "validTill": "2029-11-26T15:57:55.631Z",
        "validFrom": "2017-07-14T15:57:55.631Z"
      },
      "programId": "prg10001"
    },
    {
      "loyaltyPointDetails": [
        {
          "membershipType": "silver",
          "pointsBalance": "2000",
          "pointsType": "rewardPoints",
          "additionalProperties": {
            "propertyName7": "value7",
            "propertyName8": "value8"
          }
        },
        {
```

```
            "membershipType": "gold",
            "pointsBalance": "6000",
            "pointsType": "skyMiles",
            "additionalProperties": {
              "propertyName10": "value10",
              "propertyName9": "value9"
            }
          }
        ],
        "programName": "programForFlyer",
        "membershipId": "member0002",
        "additionalProperties": {
          "validTill": "2030-12-26T15:57:55.631Z",
          "validFrom": "2017-06-14T15:57:55.631Z"
        },
        "programId": "prg10002"
      }
    ],
    "merchantTransactionId": "MERCH-TX-1504691722267",
    "hostTransactionId": "HOST-TX-1504691722267",
    "additionalProperties": {
      "cust-prop2": "cust-prop2",
      "cust-prop1": "cust-prop1",
    },
    "responseCode": "5000",
    "merchantTransactionTimestamp": "1504691722267"
  },
  "currencyCode": "PTS",
  "transactionId": "o30417-pg30418-1504691722253",
  "transactionTimestamp": "2017-09-06T09:55:22+0000",
  "gatewayId": "demoLoyaltyPointsPaymentGateway"
}
```


Sample authorization response

The following is an example of a response to a loyalty point authorization request:


```
{
  "amount": "000000000200",
  "orderId": "o30417",
  "channel": "storefront",
  "authorizationResponse": {
    "hostTransactionTimestamp": "1504691269627",
    "responseReason": "authResponseReason",
    "customPaymentProperties": [
      "programName",
      "membershipId",
      "programId"
    ],
    "responseDescription": "authResponseDescription",
    "merchantTransactionId": "MERCH-TX-1504691269627",
    "hostTransactionId": "HOST-TX-1504691269627",
    "additionalProperties": {
      "programName": "programForFlyer",
      "membershipId": "member0002",
```

```
      "programId": "prg10002"
    },
    "responseCode": "1000",
    "merchantTransactionTimestamp": "1504691269627"
  },
  "locale": "en",
  "transactionId": "o30417-pg30415-1504691268818",
  "transactionTimestamp": "2017-09-06T09:47:48+0000",
  "transactionType": "0100",
  "paymentId": "pg30415",
  "paymentMethod": "loyaltyPoints",
  "siteId": "siteUS",
  "currencyCode": "PTS",
  "gatewayId": "demoLoyaltyPointsPaymentGateway"
}
```

Use Loyalty Points and Pay with alternate currency

Allow shoppers to pay for an order using either loyalty points or in a monetary currency or a mix of currencies.

The property `allowAlternateCurrency` is a boolean introduced at site level and is not accessible with the admin UI. At the time of order creation, `allowAlternateCurrency` is copied to the order. This setting is disabled by default, but can be enabled through the update site admin API, for example:

```
/ccadminui/v1/sites/siteUS

{ "properties": {
    "allowAlternateCurrency": true
   }
}
```

Allowing orders to be paid in mix of currencies

Enable the `allowAlternateCurrency` property to allow a shopper to pay for an order in a mix of currencies, such as:

- A monetary order paid using points and monetary currency.
- A points based order paid using monetary currency and points.

If the `payShippingInSecondaryCurrency` and `payTaxInSecondaryCurrency` setting is enabled in addition to `allowAlternateCurrency`, then there are certain restrictions on the way the order can be paid:

1. If the order is monetary order, then the entire order can be paid in monetary, but can not be paid in points. The shipping and tax total paid in monetary and leftover amount can be paid in either, points or in monetary currency or in a mix of both.
2. If the order is point based order, then the entire order can be paid in points, but can not be paid in monetary. The shipping and tax total paid in monetary and leftover amount can be paid in either, points or in monetary currency or in a mix of both.

`payShippingInSecondaryCurrency` and `payTaxInSecondaryCurrency` impose a restriction on the order, so that the shipping and tax total must be paid in monetary currency.

Choosing alternate currency

If the order is point based order and the `allowAlternateCurrency` flag is enabled, then the alternate currency which can be used to pay for the order will be the secondary currency set at the site level. The shopper is not allowed to choose any other monetary currency.

If the order is monetary based and the `allowAlternateCurrency` flag is enabled, then the alternate currency of points can be selected at the time of payment, but the shopper is allowed to use only one point type for one order. For example, if one points payment is done using, for example "bluePoints" and shopper wishes to make another payment using points, they must only use "bluePoints" in that order. The point type selected has to be sent in payments section of the order payload as "currencyCode". This is required because there is no other way of identifying the alternate currency for monetary orders and to calculate the conversion and remaining amount.

For point based orders, the alternate currency is always the site level secondary currency and need not be mentioned in the order request payload.

Setting exchange rate for amount conversion

For point based orders, the exchange rate is set at the order level at the time of the order creation itself while the pricing is completed.

For monetary orders, the exchange rate is set at order level, only if there is a payment made in points and it is done when the payment is submitted. The exchange rate set at the order level can only be changed if the order is in the pending payment state and all previous point based payments have been voided.

If the `allowAlternateCurrency` flag is disabled and `payTaxInSecondaryCurrency` and `payShippingInSecondaryCurrency` settings are disabled, then orders created are to be paid in single currency. Therefore, an error will result if a monetary order is paid using loyalty points or if a point based order is paid using card or any other monetary payment method. Single currency orders must be paid using payment methods supporting the order's primary currency.

The `calculateRemainingBalance` endpoint is used to calculate the order amount which can be paid in monetary currency and points. More details about this can be found at Payment endpoint for supporting mixed currency orders.

For mixed currency payments for a logged in user at checkout, the `createOrderForLoggedInAtCheckout` property must be enabled in client configuration, for example:

```
PUT /ccadminui/v1/merchant/
clientConfiguration{"createOrderForLoggedInAtCheckout":true}
```

Payment endpoint for supporting mixed currency orders

The `calculateRemainingBalance` endpoint is used as part of the payment framework for mixed currency orders and you can split the order total between custom and monetary currency, and this endpoint returns the order amount which can be paid in points and monetary currency.

The `calculateRemainingBalance` endpoint returns the remaining order amount in primary and alternate currency.

The order states supported are `BEING_AMENDED`, `INCOMPLETE`, `PENDING_PAYMENT`, and `QUOTED`. To use this endpoint, the `allowAlternateCurrency` flag must be enabled.

For example:

```
calculateRemainingBalance{POST ccagent/v1/payment/
calculateRemainingBalance} {POST ccstore/v1/payment/
calculateRemainingBalance }
```

This request contains:

- **amount** - The amount the user wants to pay in primary currency.
- **alternateCurrencyAmount** - The amount the user wants to pay in the alternate currency.
- **alternateCurrencyCode** - The alternate currency code. This is always the secondary currency for point based orders. For point based orders if the property is empty or null, the API assumes the alternate currency to be the secondary currency, but if an incorrect currency code is sent, then the API will throw an exception.

The following is an example of a `calculateRemainingBalance` request:

```
If the primary currency is points.
{
  "amount":20,
  "orderId":"o410438",
  "alternateCurrencyCode":"USD",
  "alternateCurrencyAmount":19
}

If the primary currency is USD
{
  "amount":20,
  "orderId":"o410438",
  "alternateCurrencyCode":"bluePoints",
  "alternateCurrencyAmount":19
}
```

This response contains:

- **amountRemaining** - The remaining order amount to be paid in primary currency.
- **alternateAmountRemaining** - The remaining order amount to be paid in alternate currency.
- **minimumMonetaryCurrencyAmountRemaining** - The minimum remaining order amount which must be paid in monetary currency. This occurs when both the `payTaxInSecondaryCurrency` and `payShippingInSecondaryCurrency` settings are enabled.
- **alternateCurrencyCode** - The alternate currency code given in the input.
- **amount** - The amount the shopper wants to pay in primary currency given in the input.

- **alternateCurrencyAmount** - The amount the shopper wants to pay in the alternate currency given in the input.
- **payments** - The existing payment groups in the order.

The following is an example of a `calculateRemainingBalance` response:


```
{
  "secondaryCurrencyTaxAmount": 0,
  "alternateCurrencyAmountRemaining": "746.0",
  "secondaryCurrencyShippingAmount": 425,
  "orderId": "o410440",
  "payments": [
    {
      "paymentGroupId": "pg530482",
      "amountAuthorized": 20,
      "amount": 20,
      "customPaymentProperties": {
        "1000addnl-property-key5": "1000payment-property-value5",
        "1000addnl-property-key2": "1000payment-property-value2"
      },
      "gatewayName": "demoLoyaltyPointsPaymentGateway",
      "uiIntervention": null,
      "paymentMethod": "customCurrencyPaymentGroup",
      "isAmountRemaining": false,
      "paymentState": "AUTHORIZED",
      "type": "loyaltyPoints",
      "currencyCode": "bluePoints"
    }
  ],
  "alternateCurrencyCode": "USD",
  "payShippingInSecondaryCurrency": true,
  "payTaxInSecondaryCurrency": true,
  "amountRemaining": "20.0",
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:9080/ccagentui/v1/payment/
calculateRemainingBalance"
    }
  ],
  "state": "PENDING_PAYMENT",
  "minimumMonetaryCurrencyAmountRemaining": "406.0",
  "currencyCode": "bluePoints"
}
```

42

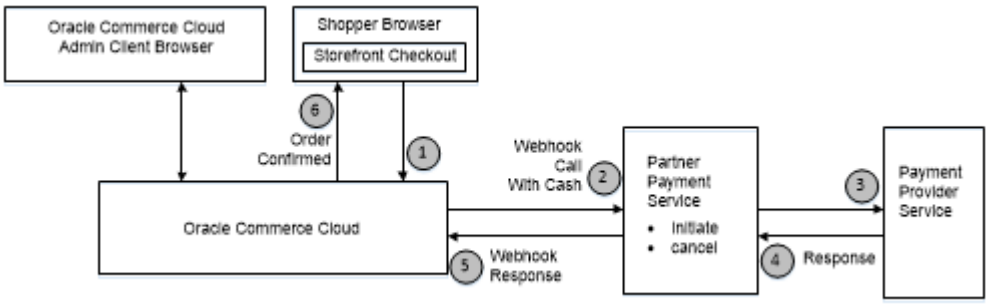
Integrate with a Cash Payment Gateway

This section describes how to integrate with a cash payment gateway, to enable customers to pay with cash after placing an order.

This section describes payment gateway workflows, extensions, webhooks and properties.

Understand the cash payment gateway workflow

The following diagram illustrates the cash payment gateway workflow:



Create a cash payment extension and configure the webhook

To create a custom integration with a cash payment gateway, perform the following steps:

1. Create the gateway extension. See Cash payment extension details for information specific to this extension.
2. Upload the extension to the administration interface.
3. Enable the gateway for the sites that require it.
4. Add a cash payment option to the checkout page.
5. Configure the Generic Payment webhook by specifying the gateway URL and the username and password. Note that webhook settings are not site-specific. The configuration you supply applies to all sites that use this webhook.

Cash payment extension details

The format of a payment gateway extension is described in the Create a Credit Card Payment Gateway Integration chapter. For a cash payment gateway, the `gateway.json` file should be similar to the following:

```
{
  "provider": "Cash Payments",
  "paymentMethodTypes": ["cash"],
  "transactionTypes": {

    "cash": ["initiate", "cancel"]
```

```
  },
}
```

The extension must also create user interface controls (for example, checkboxes) for merchants to specify the countries that cash payments are supported for.

Add a cash payment option to the checkout page

To enable paying through cash, you need to add a cash payment option to the checkout page:

1. Open the Checkout Layout that you are using on your storefront. (The default is Checkout Layout with GiftCard.)
2. Switch to grid view.
3. Add the Payment Gateway widget to the layout.
4. Add the Cash Payment element to the Payment Gateway widget.
5. Publish your changes.

Note that when a shopper checks out, the cash payment option appears on the checkout page only if the country being shipped to appears in the list of countries that cash payments are supported for.

Process the order

When a customer checks out using the cash payment option, the Generic Payment webhook sends a `CASH REQUEST` transaction type to the payment gateway. The provider can respond with the response code 1000, which acknowledges the payment request. After the payment is received, the provider can update the order with payment details using the `updateOrder` endpoint in the Admin API.

If payment has already occurred when the webhook request is received, the provider can send the response code 4000 in the webhook response, indicating that the sale is complete. In such cases, `updateOrder` calls are not needed. If response code 4000 is received, the order is marked as ready for fulfillment.

Cash payment properties

When the Generic Payment webhook executes, it sends a JSON request body to the payment gateway.

The request body contains an authorization request that contains information about the order and about the method of payment. The gateway processes the request and returns a JSON response body that contains information about the transaction, including whether the transaction succeeded. Note that if the `includeOrderInWebhookPayload` property in the gateway extension's `config.json` file is set to `true`, the order is also included in the request. See Order Submit webhook for information about the order properties.

Cash payment request properties

This section describes the top-level properties and the properties of subobjects sent in the JSON request body for cash transactions.

Top-level properties

The following table describes the top-level properties that Oracle Commerce sends in the webhook request.

Property	Description
`referenceNumber`	The ID of the internal payment group
`transactionId`	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens.

`transactionType`	For a cash payment gateway, this must be one of the following strings:
	`CASH REQUEST`
	`CASH CANCEL`

`transactionTimestamp`	The timestamp of the transaction, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

`channel`	The area of the system where the payment-processing request originated. Valid values are:
	`storefront`
	`agent`
	`preview`

`paymentMethod`	For a cash payment gateway, the value must be `cash`
`orderId`	The ID of the order associated with the payment
`amount`	The amount to be authorized, as a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
`currencyCode`	The ISO 4217 currency code
`locale`	The shopper's locale, taken from the order. If no locale is set, the default locale from the storefront is used.
`retryPaymentCount`	The number of times payment has been retried for the order
`siteURL`	The URL of the site on which the order was placed
`siteId`	The ID of the site on which the order was placed
`gatewayId`	The ID of the payment gateway

`profile` properties

The following table describes the properties of the `profile` object in the request. These values are associated with the customer purchasing the order.

Property	Description
id	The ID of the customer profile.
phoneNumber	The phone number from the customer profile.
email	The email address from the customer profile.

`profileDetails` properties

The following table describes the properties of the `profileDetails` object in the request. These values are associated with the customer purchasing the order.

Property	Description
id	The ID of the customer profile
lastName	The last name of the customer profile
firstName	The first name of the customer profile
middleName	The middle name of the customer profile
email	The email address from the customer profile
taxExempt	Indicates whether the customer tax-exempt status; either `true` or `false`
taxExemptionCode	For a customer with tax-exempt status, the exemption code
profileType	The type of profile; either `b2c_user` or `b2b_user`
receiveEmail	Indicates whether the customer agrees to receive email; either `yes` or `no`
registrationDate	The timestamp of when the profile was created, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
lastPasswordUpdate	The timestamp of when the password for the profile was last updated, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`

Sample authorization request

The following is an example of an authorization request sent by the Generic Payment webhook to a cash payment gateway:

```
{
     "transactionId": "o30446-pg30417-1458555741310",
     "currencyCode": "USD",
     "referenceNumber": "pg30417",
     "locale": "en",
     "gatewaySettings": {
          "paymentMethodTypes": "cash",
          "filteredFields": ["paymentMethodTypes"]
     },
     "amount": ""000000122526",",
     "transactionType": "CASH REQUEST",
     "transactionTimestamp": "2019-10-21T10:22:21+0000",
     "channel": "storefront",
     "orderId": "o30446",
     "paymentMethod": "cash",
     "retryPaymentCount": 0,
     "siteURL": "https://www.example.com",
     "siteId": "siteUS",
     "gatewayId": "gatewayDemo",
     "profile": {
          "id": "110454",
```

```
            "phoneNumber": "617-555-1977",
            "email": "tshopper@example.com"
        }
        "profileDetails": {
            "id": "120002",
            "lastName": "Shopper",
            "firstName": "Test",
            "taxExempt": false,
            "profileType": "b2c_user",
            "receiveEmail": "no",
            "registrationDate": "2019-10-15T06:50:51.000Z",
            "lastPasswordUpdate": "2019-10-15T06:50:51.000Z",
        }
}
```

Cash payment response properties

This section describes the top-level properties and the properties of subobjects that should be returned in the JSON response body for cash transactions.

Top-level properties

The following table describes the top-level properties that Oracle Commerce expects in the webhook response.

Property	Description
`referenceNumber`	The ID of the internal payment group. Must match the value from the request.
`transactionId`	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens.

`transactionType`	For a cash payment gateway, this must be one of the following strings:
	`CASH REQUEST`
	`CASH CANCEL`

`transactionTimestamp`	The timestamp of the transaction in Oracle Commerce, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ` Must match the value from the request.
`hostTimeStamp`	The timestamp of the transaction in the gateway (in milliseconds).
`merchantTransactionTimestamp`	The timestamp of the transaction from the merchant (in milliseconds).
`paymentMethod`	For a cash payment gateway, the value must be `cash`.
`orderId`	The ID of the order associated with the payment. Must match the value from the request.

Property	Description
amount	The actual amount collected from the shopper. This may differ from the amount in the request. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as 000000012575.
currencyCode	The ISO 4217 currency code. This is expected to match the value in the request.
siteId	The ID of the site on which the order was placed. Must match the value from the request.
gatewayId	The ID of the payment gateway. Must match the value from the request.
additionalProperties	Key/value pairs for additional properties sent by the merchant.

`authorizationResponse` properties

The following table describes the properties of the `authorizationResponse` object in the response. The values of these properties indicate whether the transaction was authorized successfully.

Property	Description
responseCode	For a cash payment gateway, this must be one of the following values: `1000` (success) `4000` (sale complete) `9000` (decline)
responseDescription	Information from the payment gateway about the response.
responseReason	Information about why the authorization succeeded or failed.
authorizationCode	The authorization code from the payment provider.
hostTransactionId	The transaction reference ID from the payment gateway.
merchantTransactionId	The transaction reference ID from the merchant.
token	The payment token used by the payment provider.

Sample authorization response

The following is an example of an authorization response sent to the Generic Payment webhook by a cash payment gateway:

```
{
     "orderId": "o30446",
     "currencyCode": "USD",
     "transactionId": "o30446-pg30417-1458555741310",
     "referenceNumber": "pg30417",
```

```
     "amount": "000000122526",
     "transactionType": "CASH REQUEST",
     "hostTimestamp": "1447807667046",
     "transactionTimestamp": "2019-10-21T10:22:21+0000",
     "merchantTransactionTimestamp": "1447807667046"
     "paymentMethod": "cash",
     "siteId": "siteUS",
     "gatewayId": "gatewayDemo",

     "authorizationResponse": {
          "responseCode": "1000",
          "responseReason": "1001",
          "responseDescription": "1002",
          "authorizationCode": "s001",
          "hostTransactionId": "h001"
     },
     "additionalProperties": {
          "sampleProperty1": "An additional property whose value will be
stored."
     }
}
```

43

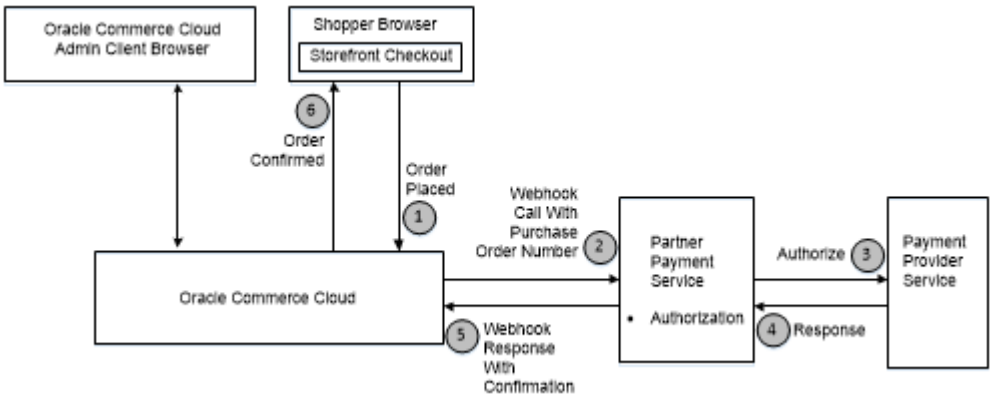
Integrate with an Invoice Payment Gateway

This section describes how to integrate with an invoice payment gateway, to enable merchants to bill customers for payments after orders are placed.

This section contains information on the payment gateway workflow, extensions, and properties.

Understand the invoice payment gateway workflow

The following diagram illustrates the invoice payment gateway workflow:



Create an invoice payment extension and modify the checkout page

To create a custom integration with an invoice payment gateway, you perform the following steps:

1. Create the gateway extension. See Invoice payment extension details for information specific to this extension.
2. Upload the extension to the administration interface.
3. Enable the gateway for the sites that require it.
4. Add an invoice payment option to the checkout page.
5. Configure the Generic Payment webhook by specifying the gateway URL and the username and password. (This step is optional. See Validate the purchase order number.) Note that webhook settings are not site-specific. The configuration you supply applies to all sites that use this webhook.
6. Add invoice payment information to order pages.

Invoice payment extension details

The format of a payment gateway extension is described in the Create a Credit Card Payment Gateway Integration chapter. For an invoice gateway, the `gateway.json` file should be similar to the following:

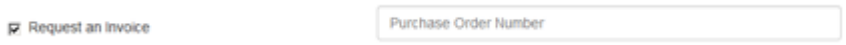
```
{
  "provider": "Invoice Payment Provider",
  "paymentMethodTypes": ["invoice"],
  "transactionTypes": {
    "invoice": ["authorization"]
  }
}
```

Add an invoice payment option to the checkout page

To enable paying through a purchase order, you need to add an invoice option to the checkout page:

1. Open the Checkout Layout that you are using on your storefront. (The default is Checkout Layout with GiftCard.)
2. Switch to grid view.
3. Add the Payment Gateway widget to the layout.
4. Add the Invoice Payment element to the Payment Gateway widget.
5. Publish your changes.

If the invoice gateway is enabled, the checkout page displays a checkbox for optionally requesting an invoice. When the shopper selects this checkbox, a text field appears for specifying a purchase order number:



Validate the purchase order number

When a shopper checks out using a purchase order number, the Store API `createOrder` endpoint includes the purchase order number in the submitted order. The merchant can then issue an invoice for the amount owed. After receiving payment, the merchant should invoke the `updateOrder` endpoint to update the order.

You can optionally configure the Generic Payment webhook to send transaction data, including the purchase order number, to an external service for validation. If the webhook is configured, Commerce waits for the result of the validation to determine whether to submit the order.

Add invoice payment information to order pages

To display information about the invoice payment selection and purchase order number on the order details and order confirmation pages, you need to make changes to the corresponding layouts.

To update the order details page perform the following steps:

1. Open the Order Details Layout.

2. Switch to grid view.
3. Replace the Order Details widget with the Order Details with Additional Info widget.
4. Publish your changes.

To update the order confirmation page:

1. Open the Order Confirmation Layout.
2. Switch to grid view.
3. Replace the Order Confirmation widget with the Order Confirmation with Additional Info widget.
4. Publish your changes.

Invoice payment properties

When the Generic Payment webhook executes, it sends a JSON request body to the payment gateway.

The request body contains information about the order and about the method of payment. The gateway processes the request and returns a JSON response body that contains information about the transaction, including whether the transaction succeeded.

Invoice payment request properties

This section describes the top-level properties and the properties of subobjects sent in the JSON request body of the Generic Payment webhook for invoice transactions. Note that if the `includeOrderInWebhookPayload` property in the gateway extension's `config.json` file is set to `true`, the order is also included in the request. See Order Submit webhook for information about the order properties.

Top-level properties

The following table describes the top-level properties that Oracle Commerce Cloud sends in the webhook request.

Property	Description
`transactionId`	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction time stamp (in milliseconds), separated by hyphens.
`transactionType`	The type of transaction. For an invoice payment gateway, this must be `AUTHORIZE`.
`transactionTimestamp`	The timestamp of the transaction, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ`
`organizationId`	The ID of the organization to be invoiced.
`organizationName`	The name of the organization to be invoiced.
`PONumber`	The purchase order number for the account to be invoiced.
`referenceNumber`	The ID of the internal payment group.

Property	Description

`channel`	The area of the system where the payment-processing request originated. Valid values are:
	`storefront`
	`agent`
	`preview`

`paymentMethod`	The payment method. For an invoice payment gateway, the value must be `invoice`.
`orderId`	The ID of the order associated with the payment.
`amount`	The amount to be authorized, as a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as `000000012575`.
`currencyCode`	The ISO 4217 currency code.
`locale`	The shopper's locale, taken from the order. If no locale is set, the default locale from the storefront is used.
`siteURL`	The URL of the site on which the order was placed.
`siteId`	The ID of the site on which the order was placed.
`gatewayId`	The ID of the payment gateway.
`retryPaymentCount`	The number of times payment has been retried for the order.
`customProperties`	Additional key/value pairs from the submitted order to be sent to the provider.

`profile` properties

The following table describes the properties of the `profile` object in the request. These values are associated with the customer purchasing the order.

Property	Description
`id`	The ID of the customer profile.
`phoneNumber`	The phone number from the customer profile.
`email`	The email address from the customer profile.

`profileDetails` properties

The following table describes the properties of the `profileDetails` object in the request. These values are associated with the customer purchasing the order. Note that for account-based commerce shoppers, this object may also include `parentOrganization`, `currentOrganization`, and `secondaryOrganizations` subobjects.

Property	Description
`id`	The ID of the customer profile
`lastName`	The last name of the customer profile
`firstName`	The first name of the customer profile

Property	Description
middleName	The middle name of the customer profile
email	The email address from the customer profile
taxExempt	Indicates whether the customer tax-exempt status; either true or false
taxExemptionCode	For a customer with tax-exempt status, the exemption code
profileType	The type of profile; either b2c_user or b2b_user
receiveEmail	Indicates whether the customer agrees to receive email; either yes or no
registrationDate	The timestamp of when the profile was created, expressed as an ISO 8601 value in the following format: yyyy-MM-dd'T'HH:mm:ssZ
lastPasswordUpdate	The timestamp of when the password for the profile was last updated, expressed as an ISO 8601 value in the following format: yyyy-MM-dd'T'HH:mm:ssZ

Sample authorization request

The following is an example of a purchase order authorization request:

```
{
   "transactionId": "o40426-pg40413-1466678343221",
   "currencyCode": "USD",
   "organizationName": "ABCD Corp",
   "locale": "en",
   "siteURL": "https://www.example.com",
   "PONumber": "po22222",
   "referenceNumber": "pg40413",
   "customProperties": { },
   "organizationId": "or-300007",
   "siteId": "siteUS",
   "gatewaySettings": {
      "paymentMethodTypes": "invoice",
      "filteredFields": [
         "paymentMethodTypes"
      ]
   },
   "amount": "000000002999",
   "transactionType": "AUTHORIZE",
   "transactionTimestamp": "2019-12-23T10:39:03+0000",
   "channel": "storefront",
   "orderId": "o40426",
   "paymentMethod": "invoice",
   "profile": {
      "id": "130000",
      "phoneNumber": "2342342345",
      "email": "a1@abcdcorp.com"
   },
```

```
   "profileDetails": {
      "id": "130000",
      "lastName": "Shopper",
      "firstName": "Test",
      "taxExempt": false,
      "profileType": "b2c_user",
      "receiveEmail": "no",
      "registrationDate": "2019-10-15T06:50:51.000Z",
      "lastPasswordUpdate": "2019-10-15T06:50:51.000Z",
   }
   "retryPaymentCount": 0,
   "gatewayId": "invoiceGateway"
}
```

Invoice payment response properties

This section describes the top-level properties and the properties of subobjects that should be returned in the JSON response body of the Generic Payment webhook for invoice transactions.

Top-level properties

The following table describes the top-level properties that Oracle Commerce expects in the webhook response.

Property	Description
`transactionId`	The unique ID of the transaction. Consists of the order ID, the payment ID, and the transaction timestamp (in milliseconds), separated by hyphens. Must match the value from the request.
`transactionType`	The type of transaction. For an invoice payment gateway, this must be `AUTHORIZE`.
`transactionTimestamp`	The time stamp of the transaction in Oracle Commerce Cloud, expressed as an ISO 8601 value in the following format: `yyyy-MM-dd'T'HH:mm:ssZ` Must match the value from the request.
`organizationId`	The ID of the organization for the account to be invoiced.
`PONumber`	The purchase order number for the account to be invoiced.
`referenceNumber`	The ID of the internal payment group. Must match the value from the request.
`hostTransactionTimeStamp`	The time stamp of the transaction from the gateway.
`merchantTransactionTimestamp`	The time stamp of the transaction from the merchant (in milliseconds).
`paymentMethod`	The payment method. For an invoice payment gateway, the value must be `invoice`.
`orderId`	The ID of the order associated with the payment. Must match the value from the request.

Property	Description
amount	The amount invoiced. The value of this property is a positive, 12-digit number that is expressed in base currency. For example, $125.75 is represented as 000000012575.
currencyCode	The ISO 4217 currency code. This is expected to match the value in the request.
siteId	The ID of the site on which the order was placed. Must match the value from the request.
gatewayId	The ID of the payment gateway. Must match the value from the request.
additionalProperties	Key/value pairs for additional properties sent by the merchant.

authorizationResponse properties

The following table describes the properties of the authorizationResponse object in the response. The values of these properties indicate whether the transaction was authorized successfully.

Property	Description
responseCode	The authorization decision from the payment provider as interpreted by the merchant. For an invoice payment gateway, this must be one of the following values: 1000 (success) 9000 (decline)
responseDescription	Information from the payment gateway about the response.
responseReason	Information about why the authorization succeeded or failed.
hostTransactionId	The transaction reference ID from the payment gateway.
merchantTransactionId	The transaction reference ID from the merchant.

Sample authorization response

The following is an example of a response to a purchase order authorization request:

```
{
    "merchantTransactionTimestamp": "1447807667046",
    "currencyCode": "USD",
    "transactionId": "o40426-pg40413-1466678343221",
    "PONumber": "po22222",
    "referenceNumber": "pg10415",
    "organizationId": "or-300007",
    "amount": "000000002999",
    "transactionType": "AUTHORIZE",
    "siteId": "siteUS",
    "authorizationResponse": {
```

```
            "hostTransactionId": "HOST-TRANSACTION-ID",
            "responseCode": "1000",
            "responseReason": "1002",
            "responseDescription": "Valid PO Number",
            "merchantTransactionId": "2016-06-23T10:39:03+0000"
        },
        "transactionTimestamp": "2016-05-02T12:14:09+0000",
        "paymentMethod": "invoice",
        "orderId": "o40426",
        "gatewayId": "invoiceGateway"
}
```

44

Integrate with a Web Checkout System

This chapter discusses integration with an external web checkout system such as Stripe Checkout or Amazon Payments.

It also discusses how to initiate, retrieve and complete an order.

Overview of web checkout system integrations

When you integrate with a payment provider, the checkout process is typically managed by the Oracle Commerce storefront.

For example, if you use the Credit Card Payment webhook, the gateway can transmit payment authorization, void, and refund requests to the payment provider. The provider processes these transactions, but does not handle the shopping cart.

Integrations based on the Generic Payment webhook can use the Commerce checkout process as well. For example, in the scenario described in Integrate with a Gift Card Payment Gateway, the Generic Payment webhook functions in a similar way to the Credit Card Payment webhook, and the checkout process is handled by the Commerce storefront.

The Generic Payment webhook, however, also supports integrations with external web checkout systems such as Amazon Payments or Stripe Checkout. When using one of these providers, some or all of the checkout experience, including the user interface, is delegated to it. Commerce passes the entire shopping cart to the external system, which is responsible for managing the checkout process and then sending the results back to Commerce.

To develop an integration with an external web checkout system, you need to do the following:

- Create a gateway extension. In the `gateway.json` file, set `paymentMethodTypes` to `generic`.
- Customize the Commerce storefront widgets to redirect to the external system, and to handle the data that the external system returns.
- Customize the system to be able to transform the Commerce data into the form required by the payment provider.

This chapter describes how to integrate with a web checkout system. This process requires using the initiate and retrieve gateway transaction types to process the cart, and authorization to complete the order. See Supported payment methods and transaction types for information about these transaction types.

Initiate the order

When the shopper invokes checkout with an external checkout system, the Store API `createOrder` endpoint sends the contents of the shopping cart to the Commerce server, and includes a special parameter (`"op":"initiate"`) to specify that the order being created is incomplete.

The server invokes the Generic Payment webhook, which sends an `initiate` transaction request with the cart data to the web checkout system. For example:

```
{
    "transactionId" : "o60412-pg60411-1465342272905",
    "currencyCode" : "USD",
    "paymentId" : "pg60411",
    "locale" : "en",
    "siteURL": "https://www.example.com",
    "customProperties" : { },
    "gatewaySettings" : [ {
      "name" : "paymentMethodTypes",
      "value" : "generic"
    } ],
    "amount" : "000000004999",
    "transactionType" : "0800",
    "items" : [ {
      "id" : "ci6000412",
      "catRefId" : "sku10020",
      "price" : 49.99,
      "rawTotalPrice" : 49.99,
      "description" : "Xbox 360 Controller",
      "quantity" : 1,
      "unitPrice" : null,
      "displayName" : null,
      "options" : [ ],
      "productId" : "prod10017"
    } ],
    "transactionTimestamp" : "2016-06-07T23:31:12+0000",

    ... billing and shipping addresses ...

    "channel" : "storefront",
    "siteId": "siteUS",
    "orderId" : "o60412",
    "paymentMethod" : "generic",
    "profile" : {
      "id" : "140160",
      "phoneNumber" : "617-555-1977",
      "email" : "bshopper@example.com"
    },
    "profileDetails": {
      "id": "140160",
      "lastName": "Shopper",
      "firstName": "Test",
      "taxExempt": false,
      "profileType": "b2c_user",
      "receiveEmail": "no",
      "registrationDate": "2019-10-15T06:50:51.000Z",
      "lastPasswordUpdate": "2019-10-15T06:50:51.000Z",
    },
    "retryPaymentCount": 0,
    "gatewayId" : "demoGenericGateway"
}
```

The checkout system typically creates its own representation of the order and sends a response to Commerce indicating that it was created successfully. This triggers an `ORDER_CREATED_INITIAL` event on the client. You can write widget code to subscribe to this topic. For example:

```
$.Topic(pubsub.topicNames.ORDER_CREATED_INITIAL).subscribe(
   widget.initialOrderCreated.bind(widget));
```

In this example, the widget defines a function called `initialOrderCreated()` that subscribes to the `ORDER_CREATED_INITIAL` topic. The response object, named order, is passed in the event. Your custom widget can retrieve the response and inspect the properties of the order.

Note that when the shopper is directed back to the checkout page from the web checkout system (as described in Retrieve the order), the shopping cart should be retrieved from the Commerce server, not reloaded from local storage. To prevent reloading from local storage, your widget code should set the `ccConstants.SKIP_LOADING_LOCAL_CART` variable to true before initiating the web checkout. For example:

```
widget.initialOrderCreated = function(orderEvent) {
    var widget = this;

storageApi.getInstance().setItem(ccConstants.SKIP_LOADING_LOCAL_CART,true);
    navigation.goTo(widget.links().checkout.route+'?
      param1=test1&param2=test2&orderId='+orderEvent.order.id);
};
```

Setting the variable to `true` prevents the cart from being reloaded from local storage, and causes a `DEFERRED_CART_LOAD` event to be published from the `initCatalog() function` of the `CartViewModel`. When the order is retrieved, the `handlePageChanged()` widget function receives this event. See Retrieve the order for more information.

Use custom properties

A key aspect of the integration involves the use of custom properties in the Generic Payment webhook data. The webhook response to the `initiate` request may include custom properties that the storefront can use to hand off control to the external system.

For example, the response may include a `REDIRECT` property that specifies the URL for redirecting the shopper's browser to the external checkout system:

```
{
     "orderId": "o60412",
     "paymentId": "pg60411",
     "merchantTransactionId": "c9f058f8-ef54-44cc-9c90-dc58269d3667",
     "hostTransactionId": "o60412-pg60411-1465342272905",
     "transactionTimestamp": "2016-06-07T23:31:12+0000",
     "hostTimestamp": "2016-06-07T23:31:14+0000",
     "transactionType": "0800",
     "siteId": "siteUS",
     "additionalProperties": {
          "AddProp1": "AddProp1_value",
          "AddProp2": "AddProp2_value",
          "AddProp3": "AddProp3_value",
          "REDIRECT": "checkout_system_URL"
```

```
    },
    "externalProperties": ["AddProp2", "REDIRECT"],
    "amount": "000000004999",
    "currencyCode": "USD",
    "response": {
        "success": true,
        "code": "Response Code Value",
        "description": "Response description value",
        "reason": "Response reason value"
    }
}
```

See Send custom properties to a payment gateway for more information about using custom properties with the Generic Payment webhook.

Retrieve the order

After processing the payment information, some web checkout systems will issue a `POST` request to the Commerce store to redirect the shopper back to the checkout page to complete the order.

The `POST` request triggers a `PAGE_CHANGED` event. Your widget code should subscribe to this topic. For example:

```
$.Topic(pubsub.topicNames.PAGE_CHANGED).subscribe(
            widget.handlePageChanged.bind(widget));
```

The widget's `handlePageChanged()` function should handle the `PAGE_CHANGED` event by parsing the parameters in the incoming URL to extract custom properties that identify the order. For example:

```
widget.handlePageChanged = function(pageData) {

  var widget = this;
  if (pageData.pageId === "checkout") {
    var urlParameters = pageData.parameters;
    if (urlParameters) {

      // extract the URL parameters
      var params = urlParameters.split('&');
      var result = {};
      for (var i = 0; i < params.length; i++) {
        var entries = params[i].split('=');
        result[entries[0]] = entries[1];
      }

      if (widget.order().paymentGateway()) {
        widget.order().paymentGateway().type = '';
      }

   . . .
```

In addition, the `handlePageChanged()` function should subscribe to the `DEFERRED_CART_LOAD` event. When it receives this event, it should determine whether the web checkout succeeded. If so, it should add the custom properties to the payment observables array in the `OrderViewModel`, and then load the shopping cart using the `getOrder()` function. If the checkout has failed, it should reload the cart from local storage. For example:

```
    . . .

       $.Topic(pubsub.topicNames.DEFERRED_CART_LOAD).subscribe(function() {
           if(result.param1 && result.param2) {

           // add the extracted parameters to the payment as custom properties
           var payment = {type: "generic", customProperties: result};
           var payments = [payment];
           widget.order().updatePayments(payments);
           widget.order().getOrder();
           }
           else {
             widget.cart().loadCart();
             }
           storageApi.getInstance().setItem(
                  ccConstants.SKIP_LOADING_LOCAL_CART,false);

           });

     }
   }
};
```

If the checkout succeeds, the `getOrder()` function is executed, which invokes the Store API `getInitialOrder` endpoint to retrieve the incomplete order. When the server receives the request, it invokes the Generic Payment webhook, which sends a `retrievetransaction` request to the web checkout system. For example:

```
{
    "transactionType" : "0900",
    "transactionTimestamp" : "2016-06-07T23:31:14+0000",
    "customProperties" : {
      "param1" : "test1",
      "param2" : "test2"
    },
    "gatewaySettings" : [ ],
    "channel" : "storefront"
 }
```

The webhook response from the web checkout system must include the `orderId` so Commerce can retrieve the order, and the `paymentId` to identify the correct payment group. You may want to include these values in the `customProperties` object of the webhook request to ensure they are available to send in the response.

The response can also include an `additionalProperties` object, as well as an `externalProperties` object that specifies a list of the properties in the `additionalProperties` object that should be returned to the storefront. For example:

```
{
     "orderId": "o60412",
     "paymentId: "pg60411",
     "siteId": "siteUS",
     "transactionTimestamp": "2016-06-07T23:31:14+0000",
     "hostTimestamp": "2016-06-07T23:31:14+0000",
     "transactionType": "0900",
     "additionalProperties": {
          "RetrieveAddProp2": "RetrieveProp2_value",
          "RetrieveAddProp1": "RetrieveProp1_value"
     },
     "externalProperties": ["RetrieveAddProp2"],
     "response": {
          "success": true,
          "code": "Response Code Value",
          "description": "Response description value",
          "reason": "Response reason value"
     }
}
```

A successful response triggers an `ORDER_RETRIEVED_INITIAL` event. Your widget can subscribe to this topic:

```
$.Topic(pubsub.topicNames.ORDER_RETRIEVED_INITIAL).subscribe(
    widget.handleOrderRetrieved.bind(widget));
```

The widget's `handleOrderRetrieved()` function should handle the `ORDER_RETRIEVED_INITIAL` event by adding the updated payment information to the `OrderViewModel`:

```
widget.handleOrderRetrieved = function(orderEvent) {

    var widget = this;
    widget.order().id(orderEvent.order.id);
    widget.order().isVerified(true);

    var payment = {type: "generic",
    paymentGroupId: orderEvent.order.payments[0].paymentGroupId};
    var payments = [payment];
    widget.order().updatePayments(payments);
};
```

Complete the order

When the customer clicks a button to submit the order, the Store API `createOrder` endpoint is called.

The request includes a parameter (`"op":"complete"`) to specify that a previously initiated order is being completed. The request must also include the order ID and the payment group ID.

The server invokes the Generic Payment webhook, which sends an `authorization` transaction request to the checkout system. Any custom properties in the order are included in the webhook request. The webhook response should indicate whether the authorization succeeded. If the authorization succeeds, the order is submitted and the shopper is redirected to the Order Confirmation page.

Validate the data

Before submitting the order, it is a good idea to validate the data being sent, including any custom properties. To do this, you can register a callback function with the `OrderViewModel` that is executed when the `handlePlaceOrder()` function is called. You can use the callback function to add validation and error handling.

The code below registers a callback function that validates the widget data. This function first calls the widget's `validate()` function. If that function returns true, the callback function adds a custom property to the payment data in the `OrderViewModel`. If `widget.validate()` returns false, the callback function adds a validation error to the `OrderViewModel`.

```
widget.validate = function {
  return true;
}

widget.order().addValidationCallback(function() {

if (widget.validate()) {
    var customProperties = {genericPaymentId: widget.genericPaymentId};
    widget.genericPayment.customProperties = customProperties;
    widget.updatePayment(widget.genericPayment);
    }
    else {
    widget.order().addValidationError("genericPayment",
      widget.translate('errorMessage'));
    }
});
```

45

Enable Split Payments

Oracle Commerce provides support for shoppers to use multiple payment methods to pay for an order.

This facility is also known as split payments, because it enables a shopper to split the cost of an order into two or more transactions that each have a different payment method.

Understand split payments

Depending on the needs of your store, you may want to enable shoppers to split order payments across multiple payment types.

Some online stores require a shopper to use a single payment method to pay for an order. For example, the shopper supplies a credit card at checkout, and uses it to pay for the entire order. In this situation, the order involves a single transaction, which includes information about the payment method (the credit card) and the amount charged to it (in this case, the total for the order), and so on.

Some shoppers, though, may want to split the cost of an order across two or more payment methods, or across multiple instances of a payment method (such as two different credit cards), or even a combination of both (such as a gift card and two credit cards). For example, if a shopper has a $50.00 gift card and an order's total cost is $85.00, the shopper may want to pay $40.00 using the gift card, and charge the remaining $45.00 to a credit card. Or a shopper may want to charge part of an order to one credit card, and the rest to another.

Some of the payment widgets (such as the Payment Details widget) support split payments in a limited way. They allow the use of multiple gift cards, or one or more gift cards plus one other payment method (such as a credit card, an invoice, or cash.) To support more complex scenarios, Commerce provides the Split Payment widget. This widget provides user interface controls that the shopper can use to break down the order cost into multiple parts, with each part associated with a different payment method instance. The widget also provides a single payment setting the shopper can switch to. The single payment setting supports the same options as the standard payment widgets (a single payment method, multiple gift cards, or gift cards plus one other method).

The Split Payment widget can be used with built-in payment gateway integrations (such as CyberSource) and payment gateway integrations you create (as described elsewhere in this manual), but requires customization in order to work with web checkout systems such as Amazon Payments. See Integrate with a Web Checkout System for information about processing payments using one of these systems.

Understand the Payment Options settings

The Setup tab in the Payment Processing settings of the administration interface has a Payment Options drop-down that controls certain aspects of how payments are processed in Commerce. This drop-down has two options:

- Allow Partial Payment/Early Persist – This setting should be used with storefronts built with the Open Storefront Framework, as well as Storefront Classic applications that use the Split Payment widget. With this setting, when the shopper submits the order, the

payments are processed in sequence, one payment at a time. The order is saved in the `PENDING_PAYMENT` state until the entire cost of the order has been authorized. If all authorizations succeed, the order is then sent to the order management system for processing. If any authorization fails, the successful authorizations are saved with the order. The shopper can correct the payment details, or leave the order in the `PENDING_PAYMENT` state and return later to specify payments for the remaining amount and resubmit the order.

- Full Payment Required – With this setting, when an order is submitted, all of the payments are processed at the same time. If authorization for any method fails, an error is displayed, and the successful authorizations are voided. The shopper must re-enter all payments and resubmit the order. If all of the payments are successfully authorized, the order is sent to the order management system for processing.

For additional information about these options, see Implement robust order capture.

For a registered shopper, partially paid orders are persisted and appear in the shopper's order history. The shopper can retrieve a partially paid order from the list and complete it. If the Order Payment Initiated email is enabled in the administration interface, an email is sent to the shopper when the order is created. The email includes the order ID. The shopper can use this ID to help locate and retrieve the order, or supply the ID to a customer service agent, who can then complete the order in the Agent Console.

For an anonymous shopper, the shopper must complete a partially paid order before the session times out, because the shopper has no way to access the order when returning to the site later. However, if the Order Payment Initiated email is enabled, the shopper is sent an email with the order ID when the order is created, and can supply this ID to a customer service agent for completing the order.

In addition to the drop-down, the Setup tab in the Payment Processing settings has a Price Hold Period setting that specifies the amount of time the shopper is given to provide missing payment information before an order is canceled. If a partially paid order is not completed before the price hold period expires, the order is marked for cancellation and is subsequently removed. To handle the actual removal of orders, a scheduled service is run. This is the same service used in account-based commerce to remove orders that have been approved but not paid for after the specified amount of time. See Set the frequency of canceled order clean up for information about configuring this service.

Partial payments must be enabled if any of the payment methods you support requires the shopper to take action while the payment is being processed. For example, if you have a credit card gateway integration that supports 3D-Secure, when the payment is processed, the shopper must authenticate with the card provider. Enabling partial payments ensures that payments are processed sequentially, preventing the authentication pages for multiple payments from displaying simultaneously or overwriting each other.

Use the Split Payment widget

To replace the existing payment widgets in your checkout layout with the Split Payments widget:

1. Open the Checkout Layout that you are using on your storefront. (The default is Checkout Layout with GiftCard.)

2. Switch to grid view.
3. Add the Split Payments widget to the layout.
4. Remove any other payment widgets from the layout.
   For example, by default, the Checkout Layout with `GiftCard` includes the Payment Details widget and the Gift Card widget. The Split Payments widget replaces both of these.

Enable partial payments

As discussed above, if your storefront is using the Split Payments widget, you need to enable partial payments in the administration interface:

1. Click the **Settings** icon.
2. Click **Payment Processing** and display the **Setup** tab.
3. From the **Payment Options** dropdown menu, select Allow Partial Payment.

By default, an order can be saved in a partially paid state indefinitely. If you want partially paid orders to expire automatically after a certain amount of time, the Setup tab also provides fields for specifying a price hold period. This is the amount of time an order can remain in the `PENDING_PAYMENT` state before it is marked for cancellation. Note that this is the same setting used in account-based commerce to determine when to cancel an order that has been approved but not paid for. See Set a price hold period.

Note that these settings are site-specific, so you will need to configure each site individually. Also, you must publish your changes for the configurations to take effect.

Use webhooks with split payments

When using split payments, each payment authorization for a gateway integrations is managed individually using the webhook appropriate for that gateway, such as the Generic Payment webhook or the Credit Card Payment webhook.

Payment authorizations for built-in gateways, such as CyberSource, are managed internally. A separate call is made for each payment group.

Webhooks that include payment data for entire orders, however, must handle multiple payment groups. The request bodies of the following webhooks can include multiple payment groups, one for each payment method instance used in the order:

- Order Submit
- Order Submit Without Payment Details
- Cart Idle
- External Price Validation
- Return Request Update
- Return Request Update for Without Payment Details

Customize the Split Payment widget

The Split Payment widget is very flexible, providing a broad range of checkout options.

Depending on the needs of your store, you may want to modify the widget to limit the options available or change some of the logic. For example, you could modify the logic to support using the widget with the Full Payment Required setting.

You can modify the widget by downloading it and customizing it. To download the Split Payment widget as a ZIP file, access the widget template in the **Components** tab in the administration design interface, and click the **Download Source** button. After customizing the widget, upload it and use it in your checkout layout. For more information, see Understand widgets.

46

Configure Tax Processors

Commerce includes integrations with both Avalara AvaTax and Vertex O Series to automatically calculate sales tax in the shopping cart. You can also configure an integration with any other tax processor for which you have an active account.

If you run multiple sites within a single instance of Commerce, these sites share a tax processor. However, each site can have its warehouse own ship-from address. See Configure Sites to learn about multiple sites. For information on working with tax processors when using loyalty programs, refer to the Work with Loyalty Programs.

Integrate with an external tax processor

Commerce includes built-in integrations with Avalara AvaTax and Vertex O Series to calculate sales tax in the shopping cart.

If you have an account with a different tax processor, you can configure an integration that tells Commerce to use that tax processor to perform tax calculations.

Understand the external tax processor integration

Commerce does not configure taxes, but integrates with tax processors that calculate sales tax in the shopping cart. To integrate with an external tax processor:

- Make sure you have a valid account with the tax processing service you want to use.
- Configure the `production-externalTaxCalculation` webhook (for the production storefront) and the `publishing-externalTaxCalculation` webhook (for the preview environment) with the URL where Commerce will send order information to the tax processor's web service and the username and password you use to log into the tax processor.
- Use the Commerce administration interface or the REST Admin API to configure the Commerce settings that identify and enable your tax processor and specify your warehouse ship-from address.

Commerce automatically uses the enabled tax processor to calculate taxes for every order placed on your store. The tax processor calculates sales tax as part of the pricing operation. When the shopping cart is priced with a request to include tax pricing (when the shopper begins the checkout process and when the order is submitted), Commerce sends the order information to the external tax processor in the body of a webhook request. The tax processor calculates the total tax amount and sends it in a response. The response breaks down the tax into individual components, for example, the total tax amount might include sales tax assessed by both the state and county.

Note: Commerce is not involved in the settlement process.

Commerce uses a fallback method for calculating taxes when it cannot connect to your tax processor's web service in the event of an outage. See Monitor tax processors for details about fallback tax calculation and information about configuring its settings.

If you want to display prices with tax included, for example prices that include VAT, create a price group for those tax-inclusive prices. See Configure Tax Processors for more information. If your store uses price groups with tax-inclusive prices, you may need to update certain settings on your tax processor's site.

Important: As a merchant, it is your responsibility to add the appropriate tax jurisdictions when you configure your profile or account on your tax processing service. Jurisdictions tell the tax processor where and when to calculate and report tax. All sales that occur in jurisdictions you have not configured in your tax processor result in tax calculation that returns a value of zero tax. Commerce is not responsible for configuring or validating jurisdictions. Look in your tax processor's documentation for information about nexus and tax jurisdictions.

Understand the Tax Code property

Each product and shipping method you create in Commerce has a Tax Code property. Tax calculators use the value of the Tax Code property to determine the tax category for a product or shipping method. For example, if you process taxes with vertex O Series, the value of the Tax Code property is the code you assigned to a taxability category in Vertex O Series. Consult the documentation for your tax processor to learn how it lets you categorize products and services.

Tax Code is not a required property; that is, Commerce allows you to create and publish a product or shipping method without providing a tax code. However, you should assign a tax code to every product and shipping method you create. Otherwise, the tax calculations may return unexpected results, including a higher-than-anticipated tax amount.

Configure the External Tax Calculation webhook

When the shopping cart is priced with a request to include tax pricing, the External Tax Calculator webhook sends a POST request to the URL you specified when you configured the webhook. (Typically this is the URL where your tax processor's web service listens for requests.) The body of the request contains the complete order data in JSON format.

The following example shows the body of an External Tax Calculation webhook POST request from Oracle Commerce. The request body is a JSON representation of the order.


```
{
 "shippingGroups" : [ {
   "priceInfo" : {
     "amount" : 59.96,
     "total" : 84.96,
     "shipping" : 25,
     "shippingSurchargeValue" : 0,
     "tax" : 0,
     "subTotal" : 59.96,
     "currencyCode" : "USD",
     "totalWithoutTax" : 84.96
   },
   "discountInfo" : {
     "orderDiscount" : 0,
     "shippingDiscount" : 0,
     "discountDescList" : [ ]
   },
```

```
    "shippingMethod" : {
      "shippingTax" : 0,
      "cost" : 25,
      "taxCode" : "",
      "value" : "priorityShippingMethod",
      "shippingMethodDescription" : "Priority"
    },
    "shippingGroupId" : "sg140414",
    "shippingAddress" : {
      "lastName" : "Smith",
      "country" : "US",
      "address3" : "",
      "address2" : "",
      "city" : "Syracuse",
      "prefix" : "",
      "address1" : "101 TNT Drive",
      "postalCode" : "13202",
      "companyName" : "",
      "jobTitle" : "",
      "county" : "",
      "suffix" : "",
      "firstName" : "Jean",
      "phoneNumber" : "555-555-1212",
      "faxNumber" : "",
      "alias" : "Home",
      "middleName" : "",
      "state" : "NY",
      "email" : null
    },
    "items" : [ {
      "unitPrice" : 14.99,
      "quantity" : 4,
      "productId" : "Product_1Ci",
      "commerceId" : "ci1443367",
      "rawTotalPrice" : 59.96,
      "returnedQuantity" : 0,
      "salePrice" : 0,
      "detailedItemPriceInfo" : [ {
        "discounted" : false,
        "amount" : 59.96,
        "quantity" : 4,
        "tax" : 0,
        "orderDiscountShare" : 0,
        "detailedUnitPrice" : 14.99,
        "currencyCode" : "USD"
      } ],
      "shippingSurchargeValue" : 0,
      "discountAmount" : 0,
      "catRefId" : "Sku_1Di",
      "discountInfo" : [ ],
      "price" : 59.96,
      "onSale" : false,
      "stateDetailsAsUser" : "The item has been initialized within the
shipping group",
      "listPrice" : 14.99,
```

```
    "status" : "INITIAL"
  } ]
} ],
"creationTime" : 1486073446086,
"isTaxIncluded" : true,
"orderId" : "o130414",
"profile" : {
  "firstName" : "Jean",
  "lastName" : "Smith",
  "taxExempt" : false,
  "receiveEmail" : "yes",
  "id" : "se-570031",
  "locale" : "en",
  "email" : "home@example.com",
  "daytimeTelephoneNumber" : ""
},
"orderStatus" : "Incomplete",
"creationDate" : "2017-02-02T22:10:46.086Z",
"orderProfileId" : "se-570031",
"callType" : "SalesOrder",
"priceInfo" : {
  "amount" : 59.96,
  "total" : 59.96,
  "shipping" : 25,
  "shippingSurchargeValue" : 0,
  "tax" : 0,
  "subTotal" : 59.96,
  "currencyCode" : "USD",
  "totalWithoutTax" : 59.96
},
"discountInfo" : {
  "unclaimedCouponMultiPromotions" : { },
  "orderCouponsMap" : { },
  "orderDiscount" : 0,
  "shippingDiscount" : 25,
  "orderImplicitDiscountList" : [ ],
  "unclaimedCouponsMap" : { },
  "claimedCouponMultiPromotions" : { }
},
"shipFromAddress" : {
  "country" : "US",
  "lastName" : null,
  "address3" : null,
  "address2" : null,
  "city" : "Cambridge",
  "address1" : "1 main st",
  "prefix" : null,
  "postalCode" : "02142",
  "county" : null,
  "ownerId" : null,
  "suffix" : null,
  "firstName" : null,
  "middleName" : null,
  "state" : "MA"
},
```

```
"shoppingCart" : {
  "numberOfItems" : 4,
  "items" : [ {
    "unitPrice" : 14.99,
    "quantity" : 4,
    "productId" : "Product_1Ci",
    "rawTotalPrice" : 59.96,
    "salePrice" : 0,
    "detailedItemPriceInfo" : [ {
      "discounted" : false,
      "amount" : 59.96,
      "quantity" : 4,
      "tax" : 0,
      "orderDiscountShare" : 0,
      "detailedUnitPrice" : 14.99,
      "currencyCode" : "USD"
    } ],
    "displayName" : "Liberty Heights",
    "shippingSurchargeValue" : 0,
    "giftWithPurchaseCommerceItemMarkers" : [ ],
    "discountAmount" : 0,
    "isItemValid" : true,
    "taxCode" : null,
    "catRefId" : "Sku_1Di",
    "skuProperties" : [ {
      "propertyType" : "sku-base",
      "name" : "Name",
      "id" : "displayName",
      "value" : null
    }, {
      "propertyType" : "sku-base",
      "name" : "Active",
      "id" : "active",
      "value" : true
    }, {
      "propertyType" : "sku-base",
      "name" : "Id",
      "id" : "id",
      "value" : "Sku_1Di"
    } ],
    "discountInfo" : [ ],
    "price" : 59.96,
    "variant" : [ ],
    "onSale" : false,
    "id" : "ci13000413",
    "listPrice" : 14.99
  } ]
},
"giftWithPurchaseInfo" : [ ],
"shippingAddress" : {
  "lastName" : "Smith",
  "country" : "US",
  "address3" : "",
  "address2" : "",
  "city" : "Syracuse",
```

```
   "prefix" : "",
   "address1" : "101 TNT Drive",
   "postalCode" : "13202",
   "companyName" : "",
   "jobTitle" : "",
   "county" : "",
   "suffix" : "",
   "firstName" : "Jean",
   "phoneNumber" : "",
   "faxNumber" : "",
   "alias" : "Home",
   "middleName" : "",
   "state" : "NY",
   "email" : null
 }
}
```

When you configure the webhook, you need to supply a URL for your tax processor's web service and the username and password you use to log into your tax processor account. You must also configure the external tax calculator service to read the data, calculate the tax, and send a response that includes the tax. You can configure webhooks on the Commerce administration interface or with the REST Admin API. See Use Webhooks for more information.

The following example shows a sample JSON response sent from a tax processor. Notice that the response breaks the total tax down into several individual components (state, city, and commuter) for each item in the order, including shipping.

```
"response": {
   {
   "shippingGroups": [
    {
      "taxPriceInfo": {
        "cityTax": 0,
        "amount": 22.99,
        "valueAddedTax": 0,
        "countyTax": 0,
        "isTaxIncluded": false,
        "miscTax": 0,
        "districtTax": 0,
        "stateTax": 0,
        "countryTax": 0
      },
      "priceInfo": {
        "amount": 89.94,
        "total": 114.94,
        "shipping": 25,
        "taxable": 89.94,
        "shippingSurchargeValue": 0,
        "tax": 22.988,
        "subTotal": 89.94,
        "currencyCode": "USD",
        "totalWithoutTax": 91.95
      },
      "shippingMethod": {
```

```
    "shippingTax": 0,
    "cost": 25,
    "taxable": 25,
    "taxDetails": [
      {
        "jurisType": "state",
        "rate": "0.2000",
        "tax": 2.5,
        "taxName": "state tax"
      },
      {
        "jurisType": "city",
        "tax": 2.5,
        "taxName": "city tax"
      },
      {
        "jurisType": "commuter",
        "tax": 1,
        "taxName": "commuter tax"
      }
    ],
    "rate": 0,
    "tax": 5,
    "taxCode": "",
    "value": "twoDayShippingMethod",
    "shippingMethodDescription": "Two Day"
  },
  "shippingGroupId": "sg70413",
  "items": [
    {
      "unitPrice": 14.99,
      "quantity": 6,
      "productId": "Product_1Ci",
      "commerceId" : "ci1443367",
      "rawTotalPrice": 89.94,
      "taxable": 89.94,
      "taxDetails": [
        {
          "jurisType": "state",
          "rate": "0.2000",
          "tax": 8.994,
          "taxName": "state tax"
        },
        {
          "jurisType": "city",
          "tax": 8.994,
          "taxName": "city tax
        },
        {
          "jurisType": "commuter",
          "tax": 1.9879999999999995,
          "taxName": "commuter tax"
        }
      ],
      "returnedQuantity": 0,
```

```
          "salePrice": 0,
          "shippingSurchargeValue": 0,
          "discountAmount": 0,
          "tax": 17.988,
          "catRefId": "Sku_1Di",
          "discountInfo": [],
          "rate": 0,
          "price": 89.94,
          "onSale": false,
          "stateDetailsAsUser": "The item has been initialized within
the shipping group",
          "listPrice": 14.99,
          "status": "INITIAL"
        }
      ]
    }
  ],
  "creationTime": 1475951869000,
  "isTaxIncluded": false,
  "orderId": "o60413",
  "orderStatus": "Incomplete",
  "creationDate": "2016-10-08T18:37:49.000Z",
  "orderProfileId": "se-570031",
  "taxDate": "02-02-2017",
  "callType": "SalesOrder",
  "status": "success",
  "timestamp": "2017-02-02T00:57:25.017"
}
}
```

If the tax processor cannot calculate the tax based on the information in the `POST` request, it returns an error response. The following example shows a sample JSON error response sent from a tax processor for an order that contains an invalid shipping address.

```
{
"response": {
     "status": "error",
      "errors": [
                {
                 "errorCode": 100123,
                 "description": "No taxing jurisdiction found.
                     Please check the address"}
                ]
      }
}
```

Configure the tax processing settings

To configure the external tax processor integration, you must supply your unique merchant ID. You can enable the external tax processor integration and configure its settings either on the Tax Processing settings page in the Commerce administration interface or with the Commerce Admin API.

Configure tax processing settings with the administration interface

This section describes how to configure integration settings for external tax processors in the Commerce administration interface.

To configure the settings for an external tax processor:

1. Click the **Settings** icon.
2. Select **Tax Processing** from the **Settings** list to display the **Tax Processing** page.
3. Select **External** from the **Tax Processor** list and select the **Enable Tax Processor** checkbox.
4. (Optional) Select **Show Tax Summary** to display an order's tax amount in the cart, checkout, and order summary pages, or deselect it to hide the tax summary. This checkbox is selected by default.
   If your store's prices include tax, such as VAT, you likely do not want to display the tax summary. However, if prices to not include tax, you should always display a tax summary.

   This setting has no effect on the display of the tax summary in the emails your store sends to customers. To learn how to remove the tax summary line from the order summaries in emails, see Customize tax display in templates.
5. Enter your Merchant ID.
   This page does not ask for a URL or account login information, as you specify those when you configure the webhook. See Configure the External Tax Calculation webhook.
6. Enter your Ship from Warehouse Location settings.
   If you run multiple sites within a single instance of Commerce, each site can have its own ship-from address. See Configure Ship from Warehouse Locations for more information.
7. Click **Save**.

The following table describes the Merchant ID and Ship-From Warehouse Location properties.

Configure tax processing settings with the Admin API

This section describes how to configure tax processing settings with the Admin API. See Use the REST APIs for information you need to know before you start working with the Admin API, including how to get an access token to make API requests.

If you run multiple sites within a single instance of Commerce, each site can have its own ship-from address. The ship-from address is a property of the site object, even if your Commerce instance runs only a single site. In previous releases, the ship-from address was a property of the `taxProcessor` object.

To configure the settings for an external tax processor, issue a `PUT` request to `/ccadmin/v1/taxProcessors/ext-p`.

The following table describes the properties for the request.

Property	Description
`company`	String that specifies your Vertex O Series taxpayer code.
`enabled`	Boolean that specifies whether the tax processor is enabled for tax calculations on your store.

Property	Description
`showTaxSummary`	Boolean that specifies whether a tax summary will be shown on your store's cart, checkout, and order summary. By default, the value of `showTaxSummary` is `false`. If your store's prices include tax, such as VAT, you likely do not want to display the tax summary. However, if prices to not include tax, you should always display a tax summary. This setting has no effect on the display of the tax summary in the emails your store sends to customers. To learn how to remove the tax summary line from the order summaries in emails, see Customize tax display in templates.
`type`	A string that specifies the tax processor to use. The value must be `external` to specify any tax processor that is not Avalara AvaTax or Vertex O Series.

The following example shows a `PUT` request that enables an external tax processor. Note that the request does not include a URL or account login information, as you specify those when you configure the webhook. See Configure the External Tax Calculation webhook for more information. This sample request also sets fallback tax calculation settings that Commerce uses if it cannot reach your tax processor, for example, in the event of an outage. See Monitor tax processors for more information.

```
PUT /ccadmin/v1/taxProcessors/ext-p HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
   "enabled": true,
   "showTaxSummary": true,
   "type": "external",
   "fallbackEnabled": true,
   "defaultTaxRate": 5,
   "fallbackRequestVolumeThreshold": 2,
   "fallbackTimeThreshold": 60000,
   "fallbackSleepWindow": 5000
}
```

The following is the sample response body for this request. Commerce returns properties for a warehouse ship-from address, even though you do not set them on the `taxProcessors` object. See Configure a ship-from address for more information.

```
 {
    "country": null,
    "isTaxIncluded": false,
    "fallbackRequestVolumeThreshold": 2,
    "city": null,
    "defaultTaxRate": 5,
    "postalCode": null,
    "hasPassword": false,
```

```
    "type": "external",
    "isActive": false,
    "url": null,
    "enabled": false,
    "fallbackTimeThreshold": 60000,
    "fallbackEnabled": true,
    "merchantId": 12345,
    "fallbackSleepWindow": 5000,
    "addressLine1": null,
    "region": null,
    "showTaxSummary": true,
    "username": null
}
```

Configure returns and exchanges tax calculation dates

You can use the `useOrderSubmittedDateForTax` flag in the merchant settings Admin API, to use the order's submitted date for tax calculations. This allows you to calculate tax refunds for returns using the original order date, rather than the current date. This sends the date of the original order to the external tax processors.

To modify this setting, issue a `PUT` command to the `/ccadmin/v1/merchant/useOrderSubmittedDateForTax` flag to `true`. For example:

```
{"userOrderSubmittedDateForTax":true}
```

To continue using the current date for tax calculations, set the `useOrderSubmittedDateForTax` flag to `false`.

If you are using Oracle Commerce 20.1 or earlier, the default for this flag is set to `false`. Later versions of Oracle Commerce have this flag default to `true`.

Configure a ship-from address

If you run multiple sites within a single instance of Commerce, each site can have its own address from which items are shipped. For example, suppose you run two sites, one that sells items in the United States, and another that sells items in France. For the US store, items would ship from your warehouse in Chicago; for the French store, items would ship from your warehouse in Paris. The ship-from address is specified by the `shipFromAddress` property of the `site` object, even if your Commerce instance runs only a single site. See Configure Sites for more information about working with `site` objects.

In previous releases, the ship-from properties were stored on the `taxProcessor` object. If you integrated with Avalara AvaTax, Vertex O Series, or an external tax processor in a previous release and configured a ship-from address, Commerce automatically adds the address to the default site in this release. If you do not specifically configure a ship-from address for a site, the site automatically inherits the address used for the default site.

To configure the ship-from address for a site, issue a `PUT` request to `/ccadmin/v1/sites/{id}`.

The following table describes the properties for the request.

Property	Description
addressLine1	String that specifies the first line of the address where your products ship from.
addressLine2	String that specifies the second line of the address where your products ship from.
addressLine3	String that specifies the third line of the address where your products ship from.
city	String that specifies the name of the city where your products ship from.
country	String that specifies the country where your products ship from.
postalCode	String that specifies the postal/ZIP code for the address where you products ship from. For addresses in the United States, use the nine-digit ZIP+4 code for best results.
region	String that specifies the name of the region, province, or state where your products ship from.

The following example shows a `PUT` request that configures the ship-from address for a site.

```
PUT /ccadmin/v1/sites/100002  HTTP/1.1
Authorization: Bearer <access_token>
x-ccasset-language: en

{
"properties": {
"shipFromAddress": {
"city": "cambridge",
"addressLine1": "2 Main Street",
"country": "US",
"region": "WA",
"postalCode": "12345"
}
}
}
```

Note that this call modifies the `productionURL` property only for the specified site, but also modifies `defaultLocaleId`, which is a global property, on all sites.

Configure tax exempt status

Some shoppers may have tax exempt status for certain purchases. For example, individuals working for a charitable organization may be exempt from sales tax on items used in running the charity.

Avalara AvaTax and Vertex O Series can automatically take into account a shopper's tax exempt status when they calculate sales tax. (Other tax processor services should be able to do this as well.) Configuring Commerce and your tax processor to handle tax exemptions involves the following steps:

1. The shopper obtains a tax exemption certificate from a taxing authority. (In the United States, the taxing authority is typically the state government of the state in which the shopper resides.) The certificate specifies what exemptions the shopper qualifies for.
2. The shopper provides you (the merchant) with a copy of the certificate.
3. Using the certificate management system supplied by your tax processor service, you upload the certificate to the service.
4. The tax processor returns a tax exemption code that is used to associate the shopper with the appropriate exemptions. (Some tax processors may use a different name for this value. Avalara AvaTax and Vertex O Series both refer to this as the customer code in their certificate management systems.) Or you can explicitly set the code to a specific value in the certificate management system.
5. You use the Commerce Admin API to set the `taxExemptionCode` property of the shopper's profile to the code returned by the tax processor.

Once you have performed these steps, no further intervention is required unless the shopper's tax exemption status changes. Each time the shopper places an order on your site, Commerce includes the shopper's tax exemption code with the order data it sends to the tax processor. The tax processor uses the code to look up the shopper's certificate and then applies the specified exemptions when it calculates sales tax on the order.

Set the shopper's tax exemption code

The only configuration step actually performed in Commerce setting the `taxExemptionCode` property of the shopper's profile. To do this, you use the `updateProfile` endpoint in the Admin API. For example:

```
PUT /ccadmin/v1/profiles/110000  HTTP/1.1
Authorization: Bearer <access_token>

{
    "taxExemptionCode": "187652"
}
```

Note that if the `taxExemptionCode` property is null or an empty string, Commerce instead includes the value of the `profileId` property as the tax exemption code when it sends order data to the tax processor. This means that if you explicitly set the exemption code in the certificate management system to the value of the shopper's `profileId`, you do not need to set the `taxExemptionCode` property on the profile.

Set the tax exemption code for account-based commerce

For account-based commerce, the tax exemption certificate applies to the account as a whole, and Commerce sends the exemption code to the tax processor for every purchase made by that account, regardless of the shopper. The account has a `taxExemptionCode` property that you set to this value using the `updateOrganization` endpoint in the Admin API. For example:

```
PUT /ccadmin/v1/organizations/100002  HTTP/1.1
Authorization: Bearer <access_token>

{
    "taxExemptionCode": "339657"
}
```

Set the tax exemption code for account-based commerce

For account-based commerce, the tax exemption certificate applies to the account as a whole, and Commerce sends the exemption code to the tax processor for every purchase made by that account, regardless of the shopper. The account has a `taxExemptionCode` property that you set to this value using the `updateOrganization` endpoint in the Admin API. For example:

```
PUT /ccadmin/v1/organizations/100002  HTTP/1.1
Authorization: Bearer <access_token>

{
    "taxExemptionCode": "339657"
}
```

Monitor tax processors

Commerce uses a fallback method for calculating taxes when it cannot connect to your tax processor's web service in the event of an outage.

The fallback tax logic automatically applies a default tax rate (that you set with the Admin API) to all orders if a specified number of calls to the tax processor fail within a specified time span. This prevents errors at the tax calculation step of order processing and allows orders to progress to the payment processing step.

Note: During the initial configuration of your environment, Oracle sets certain settings that trigger the fallback tax calculation, including the number of consecutive failed tax calls, the time span over which to count failed tax calls, and the time period after which Commerce should try calling the tax processor's service again.

Commerce uses the fallback logic to calculate taxes only when calls to the tax processor fail. That is, when the tax processor's web service responds to a call with a 500-level status code. Fallback tax calculation is not used when any other type of response is received. For example, if your Commerce tax settings contain the wrong postal code for your tax nexus, the tax processor will reply with an error, but that error will not trigger fallback tax calculation logic.

Set the default tax rate

You use the Commerce Admin API to set the `defaultTaxRate` property, an integer that specifies the tax rate to apply when fallback tax processing is used. The default value for `defaultTaxRate` is 0, which means that no tax is charged. However, you could set it to a rate that is higher than any jurisdiction where your store charges tax. The default tax rate is applied to all prices in an order, including shipping costs.

Note: During the initial configuration of your environment, Oracle sets certain settings that trigger the fallback tax calculation, including the number of consecutive failed tax calls, the time span over which to count failed tax calls, and the time period after which Commerce should try calling the tax processor's service again. You cannot change these settings. If you find that the settings are not appropriate for your store, you can file a support ticket and to change the settings. For example, if orders are failing at the tax processing step, it might be appropriate to allow fewer failed calls over a shorter time span.

To set the `defaultTaxRate` property, issue a PUT request to one of the following endpoints:

- `/ccadmin/v1/taxProcessors/vertex-p` if your tax processor is Vertex O Series
- `/ccadmin/v1/taxProcessors/ava-p` if your tax processor is Avalara AvaTax

The following example shows a PUT request that sets the default tax rate for a store that uses Avalara AvaTax to process taxes. When the fallback logic indicates that the AvaTax web service is down, Commerce stops making calls to it and automatically applies a tax rate of 10% to all orders.

```
PUT /ccadmin/v1/taxProcessors/ava-p HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
   "type":"avalara",
   "url": "<dedicated_host_instance_url>",
   "defaultTaxRate": 10,
}
```

Disable fallback tax calculation

The fallback tax calculation settings are enabled by default, but you can disable them. If you disable fallback tax calculation and Commerce cannot reach your tax processor's web service, for example in the case of an outage, shoppers will see an error during checkout and will not be able to complete their orders.

You use the Commerce Admin API to set the `fallbackEnabled` property, a Boolean that specifies whether fallback tax processing is used. The default value for `fallbackEnabled` is true.

To set this property, issue a `PUT` request to one of the following endpoints:

- `/ccadmin/v1/taxProcessors/vertex-p` if your tax processor is Vertex O Series
- `/ccadmin/v1/taxProcessors/ava-p` if your tax processor is Avalara AvaTax

The following example shows a PUT request that disables fallback tax processing for a store that uses Avalara AvaTax to process taxes.

```
PUT /ccadmin/v1/taxProcessors/ava-p HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access_token>

{
   "type":"avalara",
   "url": "<dedicated_host_instance_url>",
   "fallbackEnabled": false,
}
```

Track fallback tax calculation on orders

A Boolean property on Commerce order objects, `TaxCalculated`, indicates how tax was calculated for the order:

- If the value of `TaxCalculated` is true, tax was calculated normally by your specified tax processor.

- If the value of `TaxCalculated` is false, the default tax rate you set was applied to the order.

This property is also included in the data Commerce sends in the Order Submit webhook. You can use this information to identify orders to which the default tax rate was applied so you can decide how to handle them in your order management system.

47

Configure Search Features

This chapter describes how to use the Search and Navigation REST API to configure the features that enable shoppers to search for product data in the storefront.

The REST API enables you to export the search features from your application so that you can configure them, and then re-import them into your application. Configuration can be exported and imported in either ZIP format or JSON format, as described in this chapter. The two formats use the same set of URLs.

For an online summary of the Search and Navigation REST API, refer to the REST API for Oracle Commerce.

Important: Oracle recommends that you not modify the configuration of search features unless you are an advanced user and have requirements that you cannot meet in any other way. Oracle also recommends that you not attempt to modify any search features other than those described in this chapter. Note that some search features can be configured by the merchandiser, as described in Manage Search Settings. Consult this section before deciding to configure search features as described in this chapter.

Understand which search features can be configured

You can configure the search features that enable shoppers to search for product data in the storefront.

The following table summarizes the search features that you can configure to determine which records are returned by searches, and the order in which they are returned.

Configurable feature	How is this feature configured?	For more information, see:
How shoppers' search strings are matched with index fields (property values or dimension values)	By selecting a match mode, as part of configuring a search interface.	"Understand how search strings are matched to index fields" in the topic Understand what a search interface does
Which index fields are examined as possible matches with shoppers' search strings.	By adding index fields to the fields array in a search interface.	Specify which index fields are included in searches
Equivalent phrases for the shoppers' search terms.	By configuring a thesaurus of equivalent phrases for specific search terms.	Configure a thesaurus
URLs to which shoppers are directed when they enter specific search strings.	By configuring keyword redirects that specify the URLs to which shoppers are directed.	Configure keyword redirects
URLs that are optimized for search engines and shoppers	By configuring URLs to have directory-style structures that better enable search engines to index the URLs and that better describe the contents of pages for shoppers.	Optimize URLs for search engines

Configurable feature	How is this feature configured?	For more information, see:
How search results are ranked and sorted.	By defining a relevance ranking strategy that includes one or more relevance ranking modules. Relevance ranking strategies determine the order of records in search results.	Configure the ranking of records in search results

Understand how to execute endpoints

Each request to execute an endpoint consists of the endpoint itself - that is, an HTTP method and a URL - as well as other information required by the endpoint being executed.

For example, the endpoint that imports configuration of keyword redirects is as follows:

```
POST /gsadmin/v1/cloud/redirects
```

You must provide the following information:

- A message header specifying a valid OAuth access token
- In JSON format only, a message header specifying that imported content is to be parsed as JSON. The value of this header must be: `"Content-Type:application/json"`. This header is not used with endpoints that export or delete configuration.
- A parameter that points to the file containing the content to be imported. This parameter is not used with endpoints that export or delete configuration.

A variety of tools and utilities exist to enable you to execute endpoints. For samples of how to execute the endpoints using the cURL command line utility, see Sample Search and Navigation REST API endpoints using cURL.

Understand ZIP format and JSON format

Any request can be made in either of two formats: ZIP format and JSON format.

These formats differ only in how they import and export configuration. All configuration, however exported or imported, is written in JSON code.

The following table summarizes the differences between JSON format and ZIP format.

Format	Input and output	HTTP methods	Best uses
JSON	JSON content specified in command line or utility, or In JSON files.	GET POST PUT PATCH	Exporting configuration for viewing, or Configuring resources.

Format	Input and output	HTTP methods	Best uses
ZIP	In ZIP files that contain JSON files.	GET POST	Exporting configuration for backup and migration, or Configuring resources.

HTTP methods for configuring search features

The following table summarizes the HTTP methods supported for use with JSON format and ZIP format.

Format	Operation	Method	Examples
JSON	Export	GET	`GET/gsadmin/v1/cloud/thesaurus.json` or `GET /gsadmin/v1/cloud/thesaurus`
JSON	Create	POST	`POST/gsadmin/v1/cloud/thesaurus`

JSON	Overwrite	PUT	`PUT/gsadmin/v1/cloud/thesaurus`
			`PUT /gsadmin/v1/cloud/thesaurus/thesaurus-entry-name`
			The input to the PUT request can include configuration of the child objects (thesaurus-entry) of thesaurus. The imported configuration of the child object replaces its existing configuration.

Format	Operation	Method	Examples

JSON	Modify	PATCH	`PATCH / gsadmin/v1/ cloud/thesaurus`
			PATCH modifies an existing object by modifying the values of existing attributes in the object or adding new attributes to the object.
			Note: PATCH cannot modify child objects of the object specified in the URL.
			For example, to modify a `thesaurus-entry` object with PATCH, you must specify the URL:
			`PATCH / gsadmin/v1/ cloud/thesaurus/ thesaurus-entry-name.`

ZIP	Export	GET	`GET/gsadmin/v1/ cloud/ thesaurus.zip`
ZIP	Create	POST	`POST/gsadmin/v1/ cloud/thesaurus`
ZIP	Overwrite	POST	`POST/gsadmin/v1/ cloud/thesaurus`
n/a	Delete	DELETE	For information, see Delete resources.

Note the following differences between the HTTP methods that are supported for ZIP format and for JSON format:

- JSON format supports two methods that are not supported for ZIP format: PUT and PATCH. These methods cannot be used with ZIP format.
- With ZIP format, the POST method creates configuration, overwriting any existing configuration. With JSON format, POST can create but not overwrite; to overwrite configuration in JSON format, use the PUT method.

Delete resources

Important: Oracle recommends that you not delete high level resources such as `/thesaurus` or `/redirects`.

Oracle also recommends that you back up your configuration before deleting any object. For information, see Back up and restore all application configuration.

To delete a resource, issue the following:

```
DELETE /gsadmin/v1/cloud/resourcePath
```

where *resourcePath* indicates the object to be deleted. Both the object indicated by *resourcePath* and its child objects (if any) are deleted.

For example, the following endpoint deletes the redirects object and all its child objects (`redirect-group` objects and `redirect-entry` objects):

```
DELETE /gsadmin/v1/cloud/redirects
```

Note: Deleting the application is not supported. For example, the following endpoint fails with an error:

```
DELETE /gsadmin/v1/cloud
```

Understand system-generated object attributes

The following table lists attributes that the system adds to the configuration of objects imported through the POST, PUT, or PATCH methods.

These attributes are read only. You do not need to include these attributes in any configuration that you import.

Attribute	Name	Value
`ecr:lastModifiedBy`	user ID	The user who last modified the search interface. Example: `"ecr:lastModifiedBy": "admin"`
`ecr:lastModified`	time stamp	The time when the search interface was last modified. Example: `"ecr:lastModified": "2016-03-27T13:39:15.486Z"`
`ecr:createDate`	time stamp	The time when the search interface was created. Example: `"ecr:createDate": "2016-03-27T13:39:15.486Z"`

Export and import all search configuration

Exporting and importing all search configuration in ZIP format is a convenient technique for performing the following tasks:

- Backing up and restoring all search configuration. For information, see Back up and restore all application configuration.

- Migrating all search configuration from one environment to another. For information, see Migrate configuration of all search features.

You can also use ZIP format to configure individual resources. For information, see Configure individual resources using ZIP format.

For examples of how to specify input and output for endpoints, see Sample Search and Navigation REST API endpoints using cURL.

Export all configuration in ZIP format

Use the following endpoint to export search configuration in ZIP format:

```
GET /gsadmin/v1/cloud/resourcePath.zip
```

where:

resourcePath specifies the location of the particular resource to the exported.

`.zip` indicates that ZIP format is used. The `.zip` extension is used only with the `GET` command.

The following are examples of endpoints in ZIP format:

```
GET /gsadmin/v1/cloud.zip
```

(Exports the entire search configuration for the cloud application.)

```
GET /gsadmin/v1/cloud/thesaurus.zip
```

(Exports the entire thesaurus for the cloud application.)

```
GET /gsadmin/v1/cloud/searchInterfaces/All.zip
```

(Exports configuration for the search interface named `All`.)

When the ZIP file is downloaded through a browser client, it is saved to the default downloads location of the browser, or to a location that you specify. If the ZIP file is downloaded programmatically, your client application must determine how to handle the ZIP file.

Import all configuration in ZIP format

The endpoint to import configuration of a resource using the ZIP format is of the following form:

```
POST /gsadmin/v1/cloud/resourcePath
```

For example, the following endpoint imports configuration of the search interface `ALL`:

```
POST /gsadmin/v1/cloud/searchInterfaces/All
```

The ZIP file provided with the POST endpoint is first scanned for viruses and unsupported file types. The JSON content to be imported is then validated for

syntactical correctness. If the scanning or validation fails, an error is returned. If the scanning and validation are successful, the current configuration of the resource at `resourcePath` is fully replaced by configuration in the specified zip file.

Important: The ZIP file content that you import entirely replaces any existing configuration at *resourcePath*. Thus, if you import only the configuration for a thesaurus into `/cloud` (rather than into `/cloud/thesaurus`), all existing search configuration at `/cloud` will be replaced by the configuration of the thesaurus alone.

Configure individual resources using ZIP format

To configure an individual search resource in ZIP format, follow these steps:

1. Export the configuration of the search resource that you want to edit. For example, the following endpoint exports the configuration of the search interface `All`:

   ```
   GET /gsadmin/v1/cloud/searchInterfaces/All.zip
   ```

2. Open the downloaded ZIP file containing the configuration of the search interface `ALL` and edit the configuration.
3. Zip up the edited JSON configuration.
4. Import the ZIP file containing the edited configuration of `All` using the following endpoint:

   ```
   POST /gsadmin/v1/cloud/searchInterfaces/All
   ```

   With the POST endpoint you must specify the name and location of the ZIP file containing the configuration to be imported. For an example of how to specify the ZIP file, see Understand how to execute endpoints.

Important: Do not delete files from or add files to the exported ZIP file, or change the arrangement of folders and files in it.

Back up and restore all application configuration

To back up and restore all application configuration, use the procedure for exporting and importing configuration in ZIP files as follows:

1. Export a search configuration using the following endpoint:

   ```
   GET /gsadmin/v1/cloud.zip
   ```

   This endpoint exports all search configuration in a file whose name and location must be specified with the endpoint.
2. Store the downloaded ZIP file in a secure location.
3. To restore your search configuration, import the ZIP file containing the configuration using the following endpoint:

   ```
   POST /gsadmin/v1/cloud
   ```

With the POST endpoint you must specify the name and location of the ZIP file containing the configuration to be imported. For an example of how to specify the ZIP file, see Understand how to execute endpoints.

Migrate configuration of all search features

To migrate the configuration of all search features from one system to another, follow these steps:

1. Export all search configuration using the following endpoint:

   ```
   GET /gsadmin/v1/cloud.zip
   ```

   This endpoint exports all search configuration in a file whose name and location are specified with the endpoint.
2. Copy the downloaded ZIP file to the system to which you want to migrate the search configuration.
3. Import the search configuration in the downloaded ZIP file onto the system to which you are migrating, using the following endpoint:

   ```
   POST /gsadmin/v1/cloud
   ```

With the POST endpoint you must specify the name and location of the ZIP file containing the configuration to be imported. For an example of how to specify the ZIP file, see Understand how to execute endpoints.

Apply configuration changes to your live storefront

Changes to search configuration do not take effect in your live storefront until you publish the catalog.

Some changes, however, take effect in the preview storefront immediately, as described in the following section.

Changes that take effect in the preview storefront immediately

The following changes to search configuration take effect in the preview storefront immediately, without publishing:

- Changes in the Catalog tab.
- Changes in Search tab.
- Changes made to the configuration of resources through the Search and Navigation REST API. These resources currently include: Thesaurus, Search Interfaces, Relevance Ranking Strategy, and Keyword Redirects.

Configure a thesaurus

To broaden the search for product information that is performed when a shopper searches on a given term, you can specify one or more thesaurus entries for that term.

The search results include matches on the thesaurus entries as well as on the search term.

To create thesaurus entries, you must first create a thesaurus, which will serve as a container for all your thesaurus entries. You then add an entry to the thesaurus for each thesaurus entry that you want to define.

If your instance of Oracle Commerce is running multiple sites, all sites must share the same thesaurus configuration. You cannot configure the thesaurus differently for the different sites.

This section includes the following topics:

- Understand thesaurus entries
- Export thesaurus entries
- Create thesaurus entries
- Modify thesaurus entries
- Replace the thesaurus

Understand thesaurus entries

You can add two kinds of the following entries to your thesaurus:

- A one-way thesaurus entry, which establishes a mapping between a search term and its thesaurus entry that applies in a single direction only.
  For example, you can define a one-way mapping so that all queries on “tools” (the shopper’s search term) return matches containing “hammers” (a synonym for “tools” specified in the thesaurus) as well as matches on “tools”. Note, however, that this mapping works only one way: searching for the thesaurus entry “hammers” does not return matches containing the word “tools”.
- A multi-way thesaurus entry, which specifies two-way mappings among two or more words or phrases that are treated as equivalents of each other. **Note**: In the Oracle Commerce interface, the term *equivalent* is used in place of multi-way. Use multi-way in the REST API.
  For example, a multi-way entry might specify that the terms “adapter”, “converter”, and “adapter converter” are equivalents of each other. A search on any of these terms can return matches on any of the three.

The following table describes the JSON attributes that configure a thesaurus and thesaurus entries.

Attribute	Value
`ecr:type`	The `ecr-type` of the node. The value can be `thesaurus` or `thesaurus-entry`. Required.
`id`	Can be generated by the system or specified by the user. Required for operations on a thesaurus entry.

Attribute	Value
thesaurus-entry	Each entry has the following attributes: Type: (string, required). The supported values are: one-way: Specifies a single thesaurus entry for the searchTerms value. If entered as a search term, a searchTerms value matches the synonyms value; but a synonyms value, if entered as a search term, does not match a searchTerms value. equivalent: Specifies a list of synonyms, any one of which, if entered as a search term, matches any of the other synonyms. **Note**: In the REST API, the term multi-way is used in place of equivalent. searchTerms: (string) Required if the Type value is one-way. Not used if the entry type is multi-way. synonyms: (string or string[], required). The synonyms values are treated in the following ways: If type is one-way, the synonyms value is a single word or phrase that is considered a match for the searchTerm value. If type is multi-way, the synonyms value is a set of two or more words or phrases, any one of which is considered a match for any of the others when entered by the user as a search term.

Example: thesaurus with two thesaurus-entry objects

The following JSON illustrates the configuration of a thesaurus object containing two thesaurus-entry objects: a one-way entry that configures “shirt” as a searchTerms value and “blouse” as a synonyms value; and a multi-way entry that configures “adapter”, “converter”, and “adapter-converter” as synonyms of each other. Note that the ID of one thesaurus-entry object was generated by the system and the other was specified by the user; for information about how to specify the IDs of thesaurus-entry objects, see Create thesaurus entries.

```
{
  "ecr:type" : "thesaurus",
  "auto_generated_id":
   {
      "ecr:type": "thesaurus-entry",
      "type": "one-way",
      "searchTerms": "shirt",
      "synonyms" : ["blouse"]
    },
    "user_specified_id":
   {
      "ecr:type": "thesaurus-entry",
      "type": "multi-way",
      "synonyms": [
```

```
        "converter","adapter","adapter-converter"
      ]
    }
}
```

Thus:

- If a shopper enters “shirt” as a search term, records that include “blouse” appear in the search results; but if a shopper enters “blouse” as a search term, records that include “shirt” do not appear in the search results.
- If a shopper enters any of the words “ adapter”, “converter”, or “ adapter-converter” as a search term, records that contain any of the three words appear in the search results.

Export thesaurus entries

You can use the following endpoint to export thesaurus configuration in JSON or in a ZIP file:

```
GET /gsadmin/v1/cloud/thesaurus  (JSON format)
GET /gsadmin/v1/cloud/thesaurus.zip  (ZIP format)
```

Create thesaurus entries

Execute a POST endpoint with input such as the following to add a thesaurus entry with a user-specified ID to the thesaurus:

```
POST /gsadmin/v1/cloud/thesaurus/user_specified_id
{
  "ecr:type" : "thesaurus-entry",
  "type": "one-way",
  "synonyms": [
    "dig"
  ]
  "searchTerms": "digit"
}
```

Execute a POST endpoint with input such as the following to add a thesaurus entry with a system-generated ID to the thesaurus:

```
POST /gsadmin/v1/cloud/thesaurus
{
  "ecr:type" : "thesaurus-entry",
  "type": "one-way",
  "synonyms": [
    "dig"
  ]
  "searchTerms": "digit"
}
```

Execute the following POST endpoint to import thesaurus configuration in a zip file:

```
POST /gsadmin/v1/cloud/thesaurus
```

Modify thesaurus entries

Execute a PATCH endpoint to modify the values of existing thesaurus entries. For example, to change the synonym from `dig` to `digi` for `sample_entry_1`, you can execute the PATCH method with the following input:

```
PATCH /gsadmin/v1/cloud/thesaurus/sample_entry_1
{
  "synonyms": ["digi"],
  "ecr:type": "thesaurus-entry"
}
```

Note: PATCH is supported only in JSON format.

Replace the thesaurus

Execute a PUT endpoint to replace the thesaurus in its entirety. For example, the following endpoint replaces configuration of the thesaurus:

```
PUT /gsadmin/v1/cloud/thesaurus
```

The new configuration of the thesaurus must be included in the JSON body of the `endpoint`.

Note: PUT is supported only in JSON format.

Configure keyword redirects

This section describes keyword redirects and how to export, create, replace, and modify keyword redirect configuration using the Search Admin and Configuration REST API.

It includes the following topics:

- Understand keyword redirects
- Configure keyword redirects for multiple sites
- Keyword redirect objects
- Configure the redirects object
- Configure a redirect-group object
- Configure a redirect-entry object

Understand keyword redirects

You can configure your application to send a storefront shopper to a particular URL when the shopper enters a particular search term. The URL can represent a static page in your application (such as an “About Us” page) or a specific search or navigation state. This type of configuration is known as a keyword redirect.

You can configure any number of search terms to redirect the shopper to a particular URL. For example, you can configure the search terms “delivery” and “shipping” to redirect shoppers to a URL such as `http://shipping.example.com`.

Configure keyword redirects for multiple sites

If your instance of Oracle Commerce is running multiple sites, you can configure keyword redirects differently for the different sites by using the `siteIds` attribute. A redirect entry created without the `siteIds` attribute applies to all sites.

Keyword redirect objects

Keyword redirects are configured by the following three `ecr:type` objects:

- a `redirects` object, which contains a single:
- `redirect-group` object, which contains one or more of the following objects:
  - `redirect-entry` objects, each of which specifies the URL to which shoppers are redirected when they enter a particular search term.
  - Multiple `redirect-entry` objects, each specifying a different search term, can redirect shoppers to the same URL.

Note: It is possible to configure more than one `redirect-group` object, but for all ordinary purposes a single `redirect-group` object is sufficient.

Sample redirects configuration

The following example illustrates the configuration of a `redirects` object containing a `redirect-group` object named `Products`, which contains `redirect-entry` objects named `id1` and `id2`:

```
{
    "ecr:type": "redirects",
    "Products" : {
        "ecr:type": "redirect-group",
        "id1" : {
          "ecr:type": "redirect-entry",
          "searchTerms": "canon",
          "matchmode": "MATCHEXACT",
          "url": "/browse/Canon/_/N-1z141ya"
        },
        "id2" : {
          "ecr:type": "redirect-entry",
          "searchTerms": "nikon",
          "matchmode": "MATCHEXACT",
          "url": "/browse/Nikon/_/N-1z141ya"
      }
    }
}
```

Configure the redirects object

This section describes the operations that can be performed to export, create, replace, and modify configuration of the redirects object.

Redirects object attribute

The following table summarizes the attribute of redirects objects.

Attribute	Required?	Type	Values
"ecr:type"	yes	String	Redirects (required value) There can be only one redirects object. The redirects object can contain one or more redirect-group objects. See Sample redirects configuration.

Note: You cannot add attributes to or modify the attribute value of the redirects object. You can only create it, if for any reason it is missing.

Export redirects object in JSON format

Use the following endpoint to export configuration of the redirects object, in JSON format:

```
GET /gsadmin/v1/cloud/redirects.json
```

In JSON format, the GET endpoint returns full configuration of the redirects object, and lists the child object, Default, of the redirects object; for example:

```
{
  "ecr:lastModifiedBy": "occ_admin",
  "ecr:lastModified": "2016-10-26T09:32:32.602-07:00",
  "ecr:createDate": "2016-10-26T09:32:32.602-07:00",
  "ecr:type": "redirects",
  "Default": {
    "ecr:lastModifiedBy": "occ_admin",
    "ecr:lastModified": "2016-10-26T09:32:32.661-07:00",
    "ecr:createDate": "2016-10-26T09:32:32.661-07:00",
    "ecr:type": "redirect-group"
  }
}
```

To export the full configuration of the child object of the redirects object, use the GET method in ZIP format, as described in the following section.

Export redirects object in ZIP format

Use the following endpoint to export configuration of the redirects object in ZIP format:

```
GET /gsadmin/v1/cloud/redirects.zip
```

In ZIP format, the GET endpoint returns not only the redirects object, but also the immediate child object (redirect-group) of redirects.

The configuration of all objects is returned in a ZIP file. The ZIP file includes a file named `_.json` containing the configuration of the redirects object, and an additional `_.json` file containing partial configuration of the `redirect-group` object.

For example, suppose that the `redirects` object contains a `redirect-group` named `Default`. The `_.json` file that configures the redirects object contains the following:

```
{
  "ecr:lastModifiedBy": "admin",
  "ecr:lastModified": "2016-10-17T05:25:35.760-07:00",
  "ecr:createDate": "2016-10-17T05:25:35.760-07:00",
  "ecr:type": "redirects"
}
```

The `_.json` file that configures the `redirect-group` object named `Default` is contained in a directory named Default within the ZIP file. This `_.json` file contains configuration such as the following:

```
{
  "ecr:lastModifiedBy": "admin",
  "ecr:lastModified": "2016-10-24T07:57:48.924-07:00",
  "ecr:createDate": "2016-10-24T07:57:48.924-07:00",
  "ecr:type": "redirect-group"
}
```

To get full configuration of a `redirect-group` object, specify the following endpoint:

```
GET /gsadmin/v1/cloud/redirects/redirect-group-name.zip pathname/filename.zip
```

Create a redirects object

Use the following endpoint to configure the `redirects` object, in JSON format or ZIP format:

```
POST /gsadmin/v1/cloud/redirects
```

For most purposes, it is convenient to configure all child objects of `redirects` through the same request that configures `redirects`.

For example, the POST endpoint above can import the following JSON content to configure not only the `redirects` object, but also a `redirect-group` object named `Default` and a `redirect-entry` object named `id1` within the `redirect-group` object:

```
{
  "ecr:type": "redirects",
  "Default": {
    "ecr:type": "redirect-group",
    "id1": {
      "ecr:type": "redirect-entry",
      "searchTerms": "fujifilm",
      "siteIds":  "siteA",
      "matchmode": "MATCHEXACT",
      "url": "http://www.example.com/about-us"
```

```
      }
    }
}
```

Replace the redirects object and its children

In JSON format, use a PUT endpoint to replace the `redirects` object and all its child objects with the content provided as input to the endpoint:

```
PUT /gsadmin/v1/cloud/redirects
```

Note: The PUT method cannot be used with ZIP format.

In ZIP format, use a POST endpoint to replace the `redirects` object and all its child objects with the content provided as input to the endpoint:

```
POST /gsadmin/v1/cloud/redirects
```

For example, the PUT and POST endpoints above can import the following JSON content to replace all existing configuration of the `redirects` object and its child objects:

```
{
  "ecr:type": "redirects",
  "Default": {
    "ecr:type": "redirect-group",
    "id1": {
      "ecr:type": "redirect-entry",
      "searchTerms": "kodak",
      "matchmode": "MATCHEXACT",
      "url": "http://www.example.com/about-us"
    }
  }
}
```

Configure a redirect-group object

This section describes how to get, create, modify, and replace the configuration of a `redirect-group` object.

Redirect-group object attributes

The following table lists the attributes of a `redirect-group` object:

Attribute	Required?	Type	Values
"ecr:type"	yes	String	redirect-group (required value) **Note**: For almost all purposes, only one redirect-group object needs to be configured. Each redirect-group object, however, can contain one or more redirect-entries child objects. See Sample redirects configuration.

Export the redirect-group object in ZIP format

Use an endpoint of the following form to export configuration of a redirect-group object in ZIP format:

```
GET /gsadmin/v1/cloud/redirects/redirect-group-name.ZIP
```

The ZIP file in which the configuration is downloaded contains the following _.json files:

- A file containing the configuration of the redirects object; for example:

```
{
  "ecr:lastModifiedBy": "occ_admin",
  "ecr:lastModified": "2016-10-26T17:11:47.308Z",
  "ecr:createDate": "2016-10-26T09:32:32.602-07:00",
  "ecr:type": "redirects"
}
```

- An individual _.json file for the redirect-group object in redirects; for example, the configuration of the Default redirect-group object can be as follows:

```
{
  "ecr:lastModifiedBy": "occ_admin",
  "ecr:lastModified": "2016-10-26T10:11:47.369-07:00",
  "ecr:createDate": "2016-10-26T10:11:47.369-07:00",
  "ecr:type": "redirect-group",
  "id1": {
    "ecr:type": "redirect-entry",
    "searchTerms": "canon",
    "matchmode": "MATCHEXACT",
    "url": "http://www.example.com/about-us",
    "searchTermExpansions": {
      "0":{"canon": ["canon"]}
    }
  },
  "id2": {
```

```
    "ecr:type": "redirect-entry",
    "searchTerms": "contacts",
    "matchmode": "MATCHEXACT",
    "url": "/contact-us",
    "searchTermExpansions": {
      "0": {"contacts": ["contact"]}
    }
  }
}
```

Note: `searchTermExpansions` is a system-generated attribute. Do not delete or modify it except when the `searchTerms` attribute is updated, in which case delete the `searchTermExpansions` attribute from the JSON, use the PUT endpoint to update the redirect-group object and generate a new `searchTermExpansions` attribute.

Export the redirect-group object in JSON format

Use an endpoint of the following form to export configuration of a redirect-group object in JSON format:

```
GET /gsadmin/v1/cloud/redirects/redirect-group-name.json
```

or

```
GET /gsadmin/v1/cloud/redirects/redirect-group-name
```

For example, the following endpoint exports configuration of a `redirect-group` named Default:

```
GET /gsadmin/v1/cloud/redirects/Default
```

The following sample illustrates content of the file to which the configuration of the Default redirect-group object is exported; note that the configuration of the two redirect-entry child objects under Default is contained in this same file:

```
{
  "ecr:type": "redirect-group",
  "id1": {
    "ecr:type": "redirect-entry",
    "searchTerms": "canon",
    "matchmode": "MATCHEXACT",
    "url": "http://www.example.com/about-us",
    "searchTermExpansions": {
      "0": {"canon": ["canon"]}
    }
  },
  "id2": {
    "ecr:type": "redirect-entry",
    "searchTerms": "contacts",
    "matchmode": "MATCHEXACT",
    "url": "/contact-us",
    "searchTermExpansions": {
      "0": {"contacts": ["contact"]}
```

```
    }
  }
}
```

Note: `searchTermExpansions` is a system-generated attribute. Do not delete or modify it except when the `searchTerms` attribute is updated, in which case delete the `searchTermExpansions` attribute from the JSON, use the PUT endpoint to update the redirect-group object and generate a new `searchTermExpansions` attribute.

Create the redirect-group object

If no `redirect-group` object currently exists, you can create one using the POST method. If a `redirect-group` object currently exists, however, you do not need to create another. One `redirect-group` object is sufficient for all ordinary purposes.

Use a POST endpoint of the following form to create a `redirect-group` object, in JSON format or ZIP format:

```
POST /gsadmin/v1/cloud/redirects/redirect-group-name
```

The JSON content that configures the redirect-group object can include the configuration of the `redirect-entry` objects that the `redirect-group` is to contain; for example, the following content supplied to the POST endpoint above creates a `redirect-group` object containing two `redirect-entry` objects, `id1` and `id2`:

```
{
  "ecr:type": "redirect-group",
  "id1": {
    "ecr:type": "redirect-entry",
    "searchTerms": "canon",
    "matchmode": "MATCHEXACT",
    "url": "http://www.example.com/about-us",
    "searchTermExpansions": {
      "0": {"canon": ["canon"]}
    }
  },
  "id2": {
    "ecr:type": "redirect-entry",
    "searchTerms": "contacts",
    "matchmode": "MATCHEXACT",
    "url": "/contact-us",
    "searchTermExpansions": {
      "0": {"contacts": ["contact"]}
    }
  }
}
```

Configure a redirect-entry object

You can get, create, modify, and replace `redirect-entry` object configuration using the Search Admin and Configuration REST API.

Export redirect-entry object configuration in ZIP format

Use a GET endpoint of the following form to export configuration of a `redirect-entry` object in ZIP format:

```
GET /gsadmin/v1/cloud/redirects/redirect-group-name/redirect-entry-name.ZIP
```

The ZIP file in which the configuration is downloaded contains a file named `_.json` that contains the configuration of the `redirect-entry` object.

Export redirect-entry object configuration in JSON format

Use a GET endpoint of the following form to export configuration of a `redirect-entry` object in JSON format:

```
GET /gsadmin/v1/cloud/redirects/redirect-group-name/redirect-entry-name
```

For example, the following GET endpoint exports the configuration of a `redirect-entry` object named `ID3`:

```
GET /gsadmin/v1/cloud/redirects/Default/ID3
```

The following JSON content illustrates configuration of a `redirect-entry` object that can be exported by the endpoint above:

```
{
  "ecr:type": "redirect-entry",
  "searchTerms": "fujifilm",
  "matchmode": "MATCHEXACT",
  "url": "http://www.example.com/about-us",
  "searchTermExpansions": {"0": {"fujifilm": ["fujifilm"]}}
}
```

Create a redirect-entry object

In JSON or ZIP format, use the following POST endpoint to configure a specified `redirect-entry` object in a specified `redirect-group` object:

```
POST /gsadmin/v1/cloud/redirect/redirect-group-name/redirect-entry-name
```

For example, the following endpoint configures a `redirect-entry` object named `id3` in a `redirect-group` named Default:

```
POST /gsadmin/v1/cloud/redirect/Default/id3
```

The following code illustrates the configuration of a `redirect-entry` object that can be input to an endpoint in either ZIP or JSON format:

```
{
  "ecr:type": "redirect-entry",
  "searchTerms": "fujifilm",
  "matchmode": "MATCHEXACT",
```

```
  "url": "http://www.example.com/about-us"
}
```

Modify a redirect-entry object

In JSON format, you can use the following endpoint to modify the configuration of a specified `redirect-entry` object in a specified `redirect-group` object:

```
PATCH /gsadmin/v1/cloud/redirects/redirect-group-name/redirect-entry-name
```

Note: The PATCH method cannot be used in ZIP format.

For example, the following endpoint modifies a `redirect-entry` object named `id3` in a `redirect-group` object named Products:

```
PATCH /gsadmin/v1/cloud/redirects/Products/id3
```

The endpoint above can use JSON content such as the following to modify the `url` attribute of `redirect-entry` object named `id3`:

```
{
  "ecr:type": "redirect-entry",
  "url": "http://www.example.com/about-us"
}
```

Redirect-entry object attributes

The following table lists the attributes of `redirect-entry` objects.

Attribute	Required?	Type	Description
`"ecr:type"`	yes	String	`redirect-entry` **Note**: One or more `redirect-entry` objects can be configured as child objects of a single `redirect-group` object. See Sample redirects configuration.
`searchTerms`	yes	String	One or more phrases to be compared with search terms entered by shoppers. When matches occur between a `searchTerms` value and a search term entered by a shopper, the keyword redirect is triggered.

Attribute	Required?	Type	Description
`matchmode`	yes	String	The type of match between the specified `searchTerms` value and the search term that the user enters. Must be one of: `MATCHEXACT` – The keyword redirect is triggered only if a shopper's search terms exactly match the specified `searchTerms` values, in the same order, with no additional terms. `MATCHPHRASE` – Default. The keyword redirect is triggered if the shopper's search terms match the specified `searchTerms` values, in the same order. The shopper's search terms may include additional words before or after the `searchTerms` values. `MATCHALL` – The keyword redirect is triggered if a shopper's search terms exactly match the specified `searchTerms` values, with no additional terms, but not necessarily in the same order
`url`	yes	String	The URL to which users are redirected when a search term entered by a shopper matches a `searchTerms` value.
`siteIds`	no	Array of Strings	Specifies one or more site IDs that this redirect entry belongs to. A redirect entry created without the `siteIds` attribute applies to all sites.
`enableStemming`	no	Boolean	Determines whether the system considers the stems of search terms (for example, "box" the stem of "boxes") when constructing keyword redirects.

Attribute	Required?	Type	Description
`searchTermExpansions`	no	object	A system-generated attribute

Optimize URLs for search engines

This section describes how custom widgets can invoke an `/assembler` endpoint to generate URLs that are optimized for search engines.

The optimization not only increases the rankings of the URLs in search engine results. It also makes the URLs more descriptive of the contents of the pages and thus more intelligible to shoppers.

When URLs are optimized, the display labels of the refinements to which a shopper navigates are converted into keywords. The keywords are organized into a directory-style path that becomes part of the URL. Any search terms entered by the shopper are appended to the URL.

For example, the following optimized URL points to a page that a shopper reaches by performing a search on the term “xbox” and then selecting the refinement “Consoles”:

```
https://www.example.com/searchresults/Consoles/_/N-1817514100?Ntt=xbox
```

where:

`Consoles` is the display label for the dimension value with the ID 1817514100.

`Ntt=xbox` specifies the record search term entered by the shopper (“xbox”).

By default, only the Brand dimension and the Category dimension and its hierarchy are included in optimized URLs. For information about how to include additional dimensions in optimized URLs, see Specify which dimensions are included in optimized URLs.

To turn on SEO URL optimization for search URLs, you must first use the Admin API to issue a PUT request to `/ccadmin/v1/merchant/clientConfiguration` that sets the `useEnhancedSearch` property to `true`. The `useEnhancedSearch` property is set to `false` by default.

Optimize URLs through custom widgets

You create optimized URLs from within a custom widget by calling the URL optimization API. For detailed information about how to create custom widgets, see Understand widgets. Note that the standard widget types do not optimize URLs.

Pages pointed to by the optimized URLs can include links to refinements of the current navigation state. When a shopper clicks a refinement, the custom widget generates an optimized URL and renders that page, which can contain a link to a further refinement, and so on. Thus, a shopper can navigate from page to page following a sequence of optimized URLs.

For example, if a shopper searches on the keyword “xbox”, the custom widget executes the following endpoint and displays the corresponding page:

```
/ccstoreui/v1/assembler/pages/Default/services/guidedsearch?Ntt=xbox
```

This endpoint returns JSON containing a link to a refinement of “xbox”; for example, “Consoles”:

```
{
        "@type": "RefinementMenu",
        "displayName": "Category",
        "name": "product.category",
        "ancestors": [],
        "dimensionName": "product.category",
        "refinements": [
          {
            "link":
              "/Consoles/_/N-1817514100?Ntt=xbox",
              "label": "Consoles",
          },
          ...
        ]
      }
```

When this JSON is rendered, the shopper can click the refinement “Consoles” to advance to the next page.

The following steps illustrate in detail how custom widgets can generate a sequence of optimized URLs that reflect a shopper’s selection of refinements:

1. When a shopper navigates to a refinement or enters a keyword search term, the custom widget executes a GET operation on the following endpoint:

   ```
   /ccstoreui/v1/assembler/pages/Default/{pagePath}/{miscPath}/
   {pathSeparatorToken}/{pathParameters}?{queryString}
   ```

   where:

   `pagePath` = Indicates the page to which the URL points.

   `miscPath` = Keywords representing dimension value IDs, arranged into a directory-style structure.

   `pathSeparatorToken` = An arbitrary token (an underscore by common convention) that separates the optimized portion of the URL from the query parameters that the optimized portion represents.

   `pathParameters` = The dimension IDs in the current navigation state, prefixed by the query parameter `N-`.

   `queryString` = The search terms that the shopper enters, prefixed by the query parameter `Ntt-`.

   Note: `Default` is the required site ID.

2. The widget renders the JSON as a refinement menu for the `dimension` specified by `dimensionName`. The refinement’s display name is specified by the `label` property.

3. When the shopper selects the refinement, the custom widget constructs a search service URL consisting of the “link" value appended to the following prefix: "`/ccstoreui/v1/assembler/pages/Default/guidedsearch`".

Thus, when a shopper searches on the keyword “xbox” and then clicks the refinement Consoles, the widget constructs an optimized URL such as the following:

```
/ccstoreui/v1/assembler/pages/Default/guidedsearch” + /Consoles/_/
N-1817514100?Ntt=xbox
```

4. The custom widget renders the page pointed to by the above URL. If the shopper chooses any further refinements, the widget again executes the URL optimization API and constructs and renders a URL for the refinement page.

Note: Shoppers see optimized URLs such as the following in their browsers:

```
http://www.example.com/products/consoles/_/N-1817514100?Ntl=en&Ntt=xbox
```

They do not see the search URLs such as the following, which the custom widgets construct behind the scenes and invoke through a GET operation:

```
http://www.example.com/ccstoreui/v1/assembler/pages/Default/services/
guidedsearch/consoles/_/N-1817514100+927940346?Ntl=en&Ntt=xbox
```

Specify which dimensions are included in optimized URLs

By default, only the Brand dimension, the Category dimension, and dimensions in the Category hierarchy are included in optimized URLs.

You can specify additional dimensions to appear in the URLs by setting the `includeInNavLinks` property of the dimension’s `seoConfig` attribute. If this property is set to `true`, the dimension is included in navigation URLs. (Navigation URLs point to pages that a shopper arrives at by guided navigation.)

Recommended practice is to include dimensions that describe the qualities of products that shoppers are likely to search for (such as category, brand, fabric, or features) and to exclude dimensions with numerical values (such as rating, price or weight).

Note: For example, the `seoConfig` attribute of the color dimension can be configured as follows:

```
"product.color": {
     "ecr:type": "dimension",
     "mergeAction" : "update"
     ...
     "seoConfig": {
        "includeInNavLinks": true
     },
    ...
}
```

In the example above, `"includeInNavLinks": true` causes the dimension color to be included in navigation URLs.

Note: Include the `"mergeAction" : "update"` property if the dimension attribute already exists in Oracle Commerce. For a dimension that you are creating, omit the `"mergeAction" : "update"` property. (add is the default `mergeAction` value.)

To apply your configuration of optimized URLs, execute the following endpoint:

```
POST | PUT /gsadmin/v1/${appName}/attributes/system
```

View your changes

You can immediately view the optimized URLs in your preview store.

You must publish your changes, however, before they are visible in your production store. For information about how to preview and publish you store, refer to Publish Changes.

Specify which index fields are included in searches

This section describes search interfaces, the underlying configurable mechanism through which a storefront shopper's search for product data records is executed.

A search interface performs the following actions:

- Specifies which index fields (property values or dimension values) are examined as possible matches with shoppers' search terms.
- Specifies how search terms are matched to index fields.
- Together with a relevance ranking strategy, it determines the order in which records appear in the search results. For information about relevance ranking strategies, see Configure the ranking of records in search results.

Search interface configuration can be exported and imported in either ZIP format or JSON format. For information about these formats, see Understand ZIP format and JSON format.

Note: A search interface can be partially configured in Oracle Commerce administration interface, where it is known as a searchable field ranking. For information, see Search Results.

This section includes the following topics:

- Configure a single search interface for multiple sites
- Understand what a search interface does
- Know how the Search and Navigation REST API can configure search interfaces
- Understand search interface elements
- Know how to make fields searchable
- Add a field to a search interface
- Know when changes that affect the search interface take effect
- Export a search interface
- Create a folder object for search interfaces
- Create a search interface
- Replace configuration of a search interface
- Modify configuration of a search interface

Configure a single search interface for multiple sites

If your instance of Oracle Commerce is running multiple sites, all sites must share the same search interface configuration. You cannot configure the search interface differently for the different sites.

Understand what a search interface does

A search interface does not define or control the user interface through which shoppers search for product data in the storefront.

Understand the All and TypeAhead search interfaces

The Oracle Commerce provides two search interfaces, which perform different functions as follows:

- The search interface `All` determines which index fields are examined for possible matches with the shopper's search terms, how search terms are matched to index fields, and, together with the relevance ranking strategy, how records are sorted in results lists. Only the index fields that are included in `All` are examined for possible matches with shoppers' search terms.
  Modify the default version of `All` as needed to enable shoppers to find product information easily and efficiently.
- The search interface `TypeAhead` specifies the index fields for which `typeahead` functionality is enabled. Removing an index field from `TypeAhead` disables `typeahead` functionality for searches on that index field. Note that because wildcarding is required by the `typeahead` feature, wildcarding is enabled by default for all members of the `TypeAhead` search interface.

You can specify which index fields are included as members in `All` and `TypeAhead` using the Search tab of Oracle Commerce, or the Search and Navigation REST API endpoints to modify the configuration of the search interface.

Important: Use of search interfaces other than `All` and `Typeahead` is not supported.

Understand how searches are narrowed

A search interface narrows the scope of shoppers' searches to specified index fields in your published product data. A search interface does this by specifying the following:

- The dimensions whose dimension values will be examined as possible matches with the shopper's search term.
- The properties whose values will be examined as possible matches with the shopper's search term.
  Note: The dimensions and properties contained by a search interface are known as its members. Searches ignore dimensions and properties that are not members of a search interface.

The dimensions and properties correspond to catalog fields. When catalog content is published the following occurs:

- Product fields that the merchandiser marked as Facet become dimensions.
- Product fields that the merchandiser did not mark as Facet become properties.
- Product category fields such as "Shirts", "Dresses", "Skirts" become dimension values.

- Product records (SKUs) grouped under a product category are tagged to the corresponding dimension value.

When a storefront shopper searches on a particular term, the search results include the following product data records:

- Contains a property whose value matches the shopper's search term, and/or
- Are tagged to dimension values that match the shopper's search term. The dimension values must be in dimensions that are members of the search interface.

Example: the search interface All

Suppose that the search interface `All` includes:

- A property named `product.short_desc`
- A dimension named `product.category`

If a storefront shopper performs a search on the term "backpacks", the search results contain the following:

- Records with a `product.short_desc` property whose value matches the search term ("backpacks"); for example, a record whose `product.short_desc` property is set to "`SLR Camera/Laptop Backpack`".
  and/or
- Records tagged to dimension values in the dimension `product.category` that match the search term ("backpacks"); for example, a record tagged to the dimension value "camera backpacks and cases" in the dimension `product.category`.

Note: See the following section for information about how search terms are matched to properties and dimensions.

Understand how search strings are matched to index fields

You can configure how search strings are matched to index fields by specifying the following:

- Whether parts of a search string can be individually matched with different members.
- Whether partial query matches are supported.

For detailed information, see Search interface attributes below.

Understand how the fields array affects the sorting of search results

The order of records in search results is determined by your relevance ranking strategy. A relevance ranking strategy is composed of different modules, each of which sorts records according to its own criteria.

Some modules take into account the order of the members in the fields array of your search interface. For information about how they do this, see Understand how relevance ranking modules sort results.

The position of the member `ecr:crossField` in the `fields` array determines the position in the search results of records that are added to the list because of cross-field matches. The member `ecr:crossField` is taken into account only when the `crossFieldMatch` attribute is set to `always`.

Know how the Search and Navigation REST API can configure search interfaces

The following tasks can be performed only through the Search and Navigation REST API:

- Create and delete search interfaces
- Enable and disable cross-field matching
- Enable different types of partial matching
- Specify snippet sizes

In addition, the Search and Navigation REST API can perform a number of tasks that can also be performed through the Search tab of Oracle Commerce:

- Add fields to and delete them from a search interface
- Specify the position in search results of records that are selected because of cross-field matches

Dimensions and properties can be marked as searchable in the Catalog tab of Oracle Commerce.

Note: The Search and Navigation REST API cannot create or modify dimensions or properties.

Understand search interface elements

A search interface is configured by a JSON document that includes the following elements:

- attributes: which configure the behavior of the search interface as a whole. See the following section for information about the attributes of a search interface.
- fields: an attribute array containing the members of the search interface. Each member is a dimension or property that is examined for matches with the shopper's search term. The order of members in the fields array can affect the order of records in search results; see Understand how the fields array affects the sorting of search results.

Search interface attributes

A search interface is configured by JSON attributes. The following table summarizes these attributes. The use of these attributes is discussed in the sections following this table.

Attribute	Supported values	Description
`ecr:type`	(required) `search-interface`	Specifies that this JSON document configures a search interface.
`isAutoWildcardEnabled`	`true`, `false`	`True` enables wildcarding for all members of this search interface.

Attribute	Supported values	Description
`crossFieldMatch`	`always`, `never`, or `onFailure` (See below.)	Specifies whether parts of a search string can be individually matched with different members. For example, given the search string “red shoes”, can matches be made between “red” and one member, and between “shoes” and another member? See Examples: Cross-field matching and partial matching. Cross field matches can be made between dimensions, between properties, or between dimensions and properties. The possible values of this attribute have the following meanings:
`crossFieldMatch`	`always`	Causes the search always to look for matches between members of the search interface and the parts of a search string. `always` is the recommended setting for most purposes. See below for a description of the effect of setting `crossFieldMatch` to `always`. **Note**: If `crossFieldMatch` is not specified, always is used as a default.
`crossFieldMatch`	`never`	Requires that the entire search string be found within the same member in order to make a match. You can set `crossFieldMatch` to never only when the `ecr:crossField` attribute is not a member of the fields array.
`crossFieldMatch`	`onFailure`	Causes the search to look for matches between members of the search interface and parts of a search string only if it cannot match the entire search string to any single member of the search interface. You can set `crossFieldMatch` to `onFailure` only when the `ecr:crossField` attribute is not a member of the fields array.

Attribute	Supported values	Description
`partialMatch`	`maxWordsOmitted` or `minWordsIncluded` (See below.)	Specifies whether partial query matches should be supported for the search interface that contains this element. See Examples: Cross-field matching and partial matching. The possible values of this attribute have the following meanings:
`partialMatch`	`maxWordsOmitted`	A positive integer or zero. If the original keyword search string includes N words, then all results will match at least `N - maxWordsOmitted` words. The default is 2.
`partialMatch`	`minWordsIncluded`	A positive integer. All results will match at least this many words from the search query. The default is 2.
`fields`	(required) array of attributes	Each attribute in the `fields` array specifies a member of the search interface. An attribute can represent a property or a dimension. The position of the attribute `ecr:crossFields` in the `fields` array specifies the position in the search results of any records that are added to the search results because of cross field matches. **Note:** Include `ecr:crossfield` only if the value of `crossFieldMatch` is `Always`. Note also that `ecr:crossfield` is used only if the `Fields` module is included in your relevance ranking strategy. For information about relevance ranking strategies, see Configure the ranking of records in search results.
`snippetSize`	positive integer	Specifies the maximum number of words that this member can contain. Omitting this attribute or setting its value to zero disables the ability to create snippets. **Note**: Snippets are useful for document searches, but are not ordinarily used by eCommerce Web sites.

The following sample JSON illustrates the configuration of a search interface that includes four members, `product.displayName`, `product.brand`, `ecr:crossField`, and `product.category`:

```
{
  "ecr:type": "search-interface",
  "crossFieldMatch": "always",
  "fields": [
   {
     "attribute": "product.displayName "
   },
   {
     "attribute": "product.brand"
   },
   {
     "attribute": "ecr:crossField"
   },
   {
     "attribute": "product.category"
   }
  ]
}
```

Examples: cross-field matching and partial matching

The following examples illustrate the effects on the search results of the `crossfieldMatch` and `partialMatch` attributes.

Cross-field matching example

Suppose that a shopper searches for matches on the string “men’s shoes” through a search interface that is configured as follows:

- The `crossFieldMatch` attribute of the search interface is set to `Always`, as follows: `"crossFieldMatch": "always"`
- The members of the search interface include a dimension named `product.category` and a property named `product.description`, as follows:

```
"fields": [
    {
      "attribute": "product.category "
    },
    {
      "attribute": "product.description"
    }
]
```

The search results for a search on the string “men’s shoes” will include the following records:

- Records tagged to dimension values in the dimension `product.category` that match both “men’s” and “shoes”.
- Records with a property named `product.description` that matches both “men’s” and “shoes”.

- Records tagged to one or more dimension values in the dimension `product.category` that match “men’s” and that have a property named `product.description` whose value matches “shoes”.
- Records tagged to one or more dimension values in the dimension `product.category` that match “shoes” and that have a property named `product.description` whose value matches “men’s”.

Partial match example

Suppose that a shopper searches for matches on the string “men’s large blue suede shoes” through a search interface that is configured with the following `partialMatch` values:

```
"partialMatch":
   {"maxWordsOmitted": 1,
    "minWordsIncluded": 2
   }
```

The value of `maxWordsOmitted` specifies that a property or dimension value cannot match the shopper’s search string if it omits more than one word of the search string. Thus, only a property or dimension value that includes at least four of the five words in the search string “men’s large blue suede shoes” will match it.

The value of `minWordsIncluded` specifies that a property or dimension value cannot match any search string if it includes fewer than two words of the search string.

Note: Setting `minWordsIncluded` to a low value can increase the possibility of getting irrelevant results. Suppose, for example, that `minWordsIncluded` is set to 1.

If the shopper enters the search string “purple jeans”, the search results include all records that match “purple jeans”. If it finds no records that match “purple jeans”, it returns any records that match “purple” or “jeans” – neither of which results is relevant to the shopper’s search.

To eliminate from the search results any records that match only the single words “purple “ or “jeans”, set `minWordsIncluded` to 2. The search results now include only records that match “purple jeans”; if no records match “purple jeans” there will be no records in the search results.

Add a field to a search interface

You can add searchable fields to a search interface in any of the following ways:

- Using the Search tab of the Oracle Commerce administration interface.
- Using the Search and Navigation REST API to modify the configuration of the search interface.

Know how to make fields searchable

Index fields are examined for possible matches with the shopper’s search string only if the following conditions are met:

- They have been marked as searchable. Fields can be marked searchable by the merchandiser through the catalog view for editing product attributes, in the Oracle Commerce administration interface, and

- They have been added to the search interface as members, on the Search tab of the Oracle Commerce administration interface or through the Search and Navigation REST API, and
- They are members of the default search interface `All`.

Note: You can make a field not searchable by removing it from the search interface `All`.

Know when changes that affect the search interface take effect

Changes that you make to a search interface through the Search and Navigation REST API take effect in the preview storefront immediately, without being published. Similarly, changes that you make to an existing search interface through the Search tab take effect in the preview storefront immediately.

However, none of these changes take effect in the live storefront until you have published changes in the catalog.

Toggling fields in the catalog to determine whether they are searchable, are facets, or are multi-select facets, does not take effect until the merchandiser publishes these changes.

For more information about when changes to search interface configuration take effect in your preview and live storefronts, see Apply configuration changes to your live storefront.

Export a search interface

You can export the search interfaces `All` and `TypeAhead` in ZIP format or JSON format in order to view them, edit them, or back them up.

After you export these search interfaces, you can modify them to meet your requirements.

To export search interface configuration in ZIP format, use the following endpoint:

```
GET /gsadmin/v1/cloud/searchInterfaces/searchInterfaceName.zip
```

To export search interface configuration in JSON format, use the following endpoint:

```
GET /gsadmin/v1/cloud/searchInterfaces/searchInterfaceName.json
```

or

```
GET /gsadmin/v1/cloud/searchInterfaces/searchInterfaceName
```

Note: The search interface's name is not included in the JSON that configures the search interface.

The following example illustrates the JSON representation of a search interface named `All` that can be exported by the endpoints above:

```
{
    "ecr:lastModifiedBy": "admin",
    "ecr:lastModified": "2016-03-27T13:39:15.486Z",
```

```
    "ecr:createDate": "2016-03-27T13:39:15.486Z",
    "ecr:type": "search-interface",
    "crossFieldMatch": "always",
    "fields": [
        {
            "attribute": "product.id"
        },
        {
            "attribute": "product.sku"
        },
        {
            "attribute": "product.code"
        },
        {
            "attribute": "product.brand.name"
        },
        {
            "attribute": "product.category"
        },
        {
            "attribute": "product.name"
        },
        {
            "attribute": "ecr:crossField"
        },
        {
            "attribute": "product.long_desc"
        }
    ]
}
```

Note: The endpoints `GET/gsadmin/v1/cloud/searchInterfaces.zip` and `GET/gsadmin/v1/cloud/searchInterfaces.json` return a list of search interfaces that does not include the full configurations of the individual search interfaces.

Create a folder object for search interfaces

If a folder object for search interfaces does not exist, you must re-create it. Do not attempt to create more than one `searchInterfaces` folder.

The `searchInterfaces` folder must contain a search interface named All; if All is missing, shoppers cannot search the catalog. It must also contain a search interface named TypeAhead if typeahead functionality is to be enabled for shoppers' search strings.

Use the following endpoint to create the `searchInterfaces` folder:

```
POST /gsadmin/v1/cloud/searchInterfaces
```

The POST endpoint must import the following JSON configuration:

```
{
  "ecr:type": "search-interface-folder"
}
```

Note: For most purposes, it is convenient to create the search interface `All` when you create the `searchInterfaces` folder. In this case, the JSON configuration that you provide as input to the POST request includes the configuration of `All` as well as the configuration of the `searchInterfaces` folder.

Create a search interface

Note: Only two interfaces are supported: `All` and `TypeAhead`. Thus, you will need to create a search interface only if, for whatever reason, one of these search interfaces is missing. Do not attempt to create or use search interfaces other than `All` and `TypeAhead`.

In JSON format or ZIP format, use a POST endpoint to create the configuration of a search interface.

For example, the following endpoint imports configuration of a search interface named `All`:

```
POST /gsadmin/v1/cloud/searchInterfaces/All
```

The endpoint must include JSON configuration of the search interface `All` as its content; for example:

```
{
  "ecr:type": "search-interface",
  "crossFieldMatch": "always",
  "fields": [
    {
      "attribute": "product.id"
    },
    {
      "attribute": "product.sku"
    },
    {
      "attribute": "product.code"
    },
    {
      "attribute": "product.brand.name"
    },
    {
      "attribute": "product.category"
    },
    {
      "attribute": "product.name"
    },
    {
    {
      "attribute": "ecr:crossField"
    },
      "attribute": "product.long_desc"
    }
  ]
}
```

Note: The JSON object that configures a search interface does not specify the name of the search interface. Instead, the name of the search interface is assumed to be the same as the name of the subfolder (for example, `All`) where the search interface is created.

Replace configuration of a search interface

In ZIP format, you can use the following POST endpoint to replace the current configuration of a search interface in its entirety. For example, the following endpoint replaces configuration of the search interface `ALL`:

```
POST /gsadmin/v1/cloud/searchInterfaces/All
```

In JSON format, you can use the following PUT endpoint to replace the current configuration of a search interface in its entirety. For example, the following endpoint replaces configuration of the search interface `ALL`:

```
PUT /gsadmin/v1/cloud/searchInterfaces/All
```

The PUT method can only replace a search interface in its entirety; it cannot replace parts of a search interface. An error results if you attempt to PUT a search interface with a given name when no search interface with that name currently exists.

Modify configuration of a search interface

In JSON format, you can use a PATCH endpoint to modify the configuration of a search interface by adding attributes to it or changing the values of existing attributes. For example, the following endpoint modifies the configuration of the search interface `All`:

```
PATCH /gsadmin/v1/cloud/searchInterfaces/All
```

For example, suppose that the search interface `All` is currently configured as follows:

```
{
  "ecr:lastModifiedBy": "admin",
  "ecr:lastModified": "2016-07-07T17:17:07.474-04:00",
  "ecr:createDate": "2016-05-06T17:15:06.414-04:00",
  "ecr:type": "search-interface",
  "crossFieldMatch": "always",
  "fields": [
    { "attribute": "product.displayName" },
    { "attribute": "product.brand" },
    { "attribute": "ecr:crossField" },
    { "attribute": "product.category" }
  ]
}
```

You can add a snippet size attribute to this configuration of `All` by executing the PATCH endpoint above with the following input:

```
{
  "ecr:type": "search-interface",
  "fields": [
    {
```

```
      "snippetSize": 20,
      "attribute": "product.short_desc"
    }
  ]
}
```

The search interface `ALL` is now configured as follows:

```
{
  "ecr:lastModifiedBy": "admin",
  "ecr:lastModified": "2016-07-07T17:17:07.474-04:00",
  "ecr:createDate": "2016-05-06T17:15:06.414-04:00",
  "ecr:type": "search-interface",
  "crossFieldMatch": "always",
  "fields": [
    { "attribute": "product.displayName" },
    { "attribute": "product.brand" },
    { "attribute": "ecr:crossField" },
    { "attribute": "product.category" },
    {
      "snippetSize": 20,
      "attribute": "product.short_desc"
    }
  ]
}
```

You can also use PATCH to modify the values of existing attributes. For example, to change the snippet size from 20 to 25, you can execute the PATCH method with the following input:

```
{
  "ecr:type": "search-interface",
  "fields": [
    {
      "snippetSize": 25,
      "attribute": "product.short_desc"
    }
  ]
}
```

Note: You cannot re-order the field members in a search interface using PATCH.

Index and Query Popular Searches

Oracle Commerce offers a ready-to-use typeahead function, which searches against and returns product records. This can be used to display products, their prices, images, and so on

Typeahead enables shoppers to also search against other popular search terms. It comes with pre-configured APIs and schema. You need to provide your own search terms and handle rendering the results from this endpoint.

To provide an alternative type-ahead experience for shoppers, you can configure Commerce to search against terms that it has automatically identified as popular with other shoppers. Additionally, Commerce can also search against your own custom or curated search terms.

You must implement this feature using the REST APIs. Commerce widgets do not support this feature by default. To use this feature, you must ensure that you are using the correct configuration, and handle rendering the results from this endpoint. Perform the following steps to use this feature:

- Index popular searches records
- Query the correct endpoint

Note: Popular searches are automatically generated only in production and live environments.

Index popular searches records

Each term that can appear in the results for a query against popular searches is a single record. If you have a list of 15,000 popular search terms, each term is a separate record and indexed separately.

For each record, there are four required fields:

- `record.id`
- `keyword.terms`
- `keyword.searchable`
- `keyword.score`

Shopper searches are normally matched against the `keyword.searchable` field, which may contain synonym forms, common misspellings, or any other “alternative” forms of the term corresponding to a given record. The `keyword.terms` field contains the canonical form of the term and is the one normally displayed in the results and searched for when the result is selected.

Ensure that all the fields above are properly configured by submitting the following `POST` request:

```
POST /gsadmin/v1/cloud/attributes/keywords
{
  "ecr:type": "attributes-owner-folder",
  "keyword.terms": {
    "ecr:type": "property",
    "propertyDataType": "ALPHA",
    "isRecordSearchEnabled": false,
    "isWildcardEnabledInRecordSearch": false,
    "context": [ "locale" ]
  },
  "keyword.searchable": {
    "ecr:type": "property",
    "propertyDataType": "ALPHA",
    "isRecordSearchEnabled": true,
    "isWildcardEnabledInRecordSearch": true,
    "context": [ "locale" ]
  },
  "keyword.score": {
```

```
    "ecr:type": "property",
    "propertyDataType": "DOUBLE"
  },
  "keyword.searchCount": {
    "ecr:type": "property",
    "propertyDataType": "INTEGER"
  }
}
```

Suppress product names as suggestions

In all environments, product names are included in the search suggestions to provide a "cold start" for newly launched sites. You may prefer to switch this off. To do so, call the API endpoint `/gsadmin/v1/cloud/configuration/services/internal-keywords` and change the property `gsdataCatalogLimit` to `1`.

Automatic popular searches records

Commerce automatically identifies popular searches based on shoppers' past behavior, and periodically uploads the corresponding records to the internal-keywords record collection. From there, they are processed for indexing. The contents of the internal-keywords record collection at any given time can be inspected by querying the following endpoint:

```
GET /gsdata/v1/cloud/data/internal-keywords
```

Important: Do not attempt to modify the internal-keywords record collection. Your changes will be overwritten.

Preview popular searches in non-live environments

Should you wish to preview the search suggestions from a Live environment in a lower environment, you can temporarily migrate the search suggestions to a lower environment by performing the following steps:

1. Issue a `DELETE` command to `/gsdata/v1/cloud/data/internal-keywords` on the DEV / TEST environment to remove any existing data (if necessary).
2. Issue a `GET` command to `/gsdata/v1/cloud/data/internal-keywords` on the LIVE environment to retrieve the production data.
3. Issue a `POST` command to `/gsdata/v1/cloud/data/internal-keywords` on the DEV / TEST environment with the retrieved production data.

Doing this will copy the production suggestions to the targeted environment. Note that this is a temporary operation. Should you wish to see the latest suggestions, you will need to perform the above steps again.

Add your own custom keyword records

To add custom keyword records to the system so they can be processed for indexing, you must upload them to the keywords record collection with the following endpoint:

```
POST /gsdata/v1/cloud/data/keywords
```

Multiple records can be submitted in each request. Multiple requests can be made to add data. That is, subsequent `POST` calls to `/gsdata` will not replace previously

submitted records. If you need to submit a large number of records, it is recommended to submit then in a few sets, rather than one at a time.

The following is a sample `POST` command that submits three records using the above schema:

```
POST /gsdata/v1/cloud/data/keywords
{
  "items":[
   {
    "record.id": "kw-en:digital+cameras",
    "keyword.terms@locale:en": "digital cameras",
    "keyword.searchable@locale:en": [
      "digital cameras", "digicams"
    ],
    "keyword.score": "12234"
   },
   {
    "record.id": "kw-en:film+cameras",
    "keyword.terms@locale:en": "film cameras",
    "keyword.searchable@locale:en": "film cameras",
    "keyword.score": "3234"
   },
   {
    "record.id":"kw-en:dslr+cameras",
    "keyword.terms@locale:en": "dslr cameras",
    "keyword.searchable@locale:en": [
      "dslr cameras", "digital slr cameras", "slr digicams"
    ],
    "keyword.score": "23421"
   }
  ]
}
```

Localize the custom keyword records

The fields `keyword.terms` and `keyword.searchable` in the previous section are localized. When uploading records, you must specify the correct locale (for example, `@locale:en` for English) that shoppers will search under.

Each keyword record can optionally contain translations for multiple locales. It is perfectly acceptable to create multiple, single-locale keyword records for multiple locales. For example:

```
POST /gsdata/v1/cloud/data/keywords
{
  "items":[
   {
    "record.id": "kw-en:book",
    "keyword.terms@locale:en": "book",
    "keyword.searchable@locale:en": "book",
    "keyword.score": "1"
   },
   {
```

```
    "record.id": "kw-fr:livre",
    "keyword.terms@locale:fr": "livre",
    "keyword.searchable@locale:fr": "livre",
    "keyword.score": "1"
   },
   {
    "record.id": "kw:notebook",
    "keyword.terms@locale:en": "notebook",
    "keyword.searchable@locale:en": "notebook",
    "keyword.terms@locale:fr": "cahier",
    "keyword.searchable@locale:fr": "cahier",
    "keyword.score": "1"
   }
]
}
```

Index additional fields for custom keyword records

You may want to return additional information with your custom keyword records. In this case, you can add custom fields. There is no fixed schema for those additional fields, but it is recommended that you follow the naming convention of `keyword.x`. For example, `keyword.related_terms`, or `keyword.related_product_id`. For more information, see Modify the configuration to return additional fields.

Perform indexing

If you added custom keyword records to the system and defined schema, you need to run the search indexing process. To do so, use the following API call. Keep in mind that indexing times will vary depending on the size of your catalog.

```
POST /ccadmin/v1/search/index {"op":"baseline" }
```

Query the keywords endpoint

At this point, your popular search records are available in the index.

Ensure that the search service definition is properly configured by using the following endpoints:

```
POST /gsadmin/v1/cloud/searchInterfaces/keywords
{
  "isAutoWildcardEnabled": true,
  "ecr:type": "search-interface",
  "crossFieldMatch": "never",
  "fields": [
    {
      "attribute": "keyword.searchable"
    }
  ]
}

POST /gsadmin/v1/cloud/pages/Default/keywords/typeahead
{
  "contentType": "Page",
  "ecr:type": "page",
```

```
  "contentItem": {
    "@name": "Keyword Search Service",
    "@type": "KeywordSearchService",
    "@appFilterState": {
      "@type": "FilterState",
      "typeAhead": true,
      "recordFilters": [
        "OR(record.collection:keywords,record.collection:internal-keywords)"
      ]
    },
    "resultsList": {
      "@type": "ResultsList",
      "relRankStrategy": "static(keyword.score,descending)",
      "fieldNames": [
        "record.id",
        "keyword.terms",
        "keyword.score"
      ]
    }
  }
}
```

Run the following endpoint:

```
GET /ccstore/v1/assembler/pages/Default/keywords/typeahead
```

and specify the following two parameters to perform a search:

- `Ntt=` This is the user's search terms. For instance, if the customer has typed "cam" as part of searching for "camera", you would set it to `&Ntt=cam`
- `Ntk=` This specifies the search key. A search interface named "keywords" has been defined

For example:

```
GET /ccstore/v1/assembler/pages/Default/keywords/typeahead?
Ntt=cam&Ntk=keywords
```

Modify the configuration to return additional fields

In order to provide a responsive service that is also quite small in size, the default /keywords/typeahead service is configured only to return a minimal set of fields in the `resultsList`.

If you want additional fields returned, you must modify the `/keywords/typeahead` service:

1. Run `GET /gsadmin/v1/cloud/pages/Default/keywords/typeahead`.
2. Add the custom fields to the list of `fieldNames`.
3. `PUT` the modifications back to `/gsadmin/v1/cloud/pages/Default/keywords/typeahead`.

Configure typeahead query prefixes

By default, Commerce ignores typeahead and wildcard search terms of less than 2 characters.

If you want to change the default behavior, you can use the Search Interfaces endpoints in the Search Admin and Configuration API to change the value of the `minWildcardWordLength` search configuration property. For example, if you set the property value to zero (0), the search begins as soon as the shopper types a character.

Keep in mind that setting the value of `minWildcardWordLength` to less than the default value (2) might impact performance.

Note that the `minWildcardWordLength` search configuration property does not apply to dimension search or compound dimension search, or to the Chinese, Japanese, or Korean languages.

The following sample shows how you can set the value of `minWildcardWordLength` to 0 for typeahead and wildcard searches.

```
PUT /gsadmin/v1/cloud/searchInterfaces/TypeAhead
{
  "ecr:type": "search-interface",
  "crossFieldMatch": "always",
  "isAutoWildcardEnabled": true,
  "minWildcardWordLength": 0,
  "fields": [
    {
      "attribute": "product.displayName"
    },
    {
      "attribute": "ecr:crossField"
    },
    {
      "attribute": "product.category"
    }
  ]
}
```

Modify data structures to enhance searches and navigation

In some cases, you can enhance the ease and accuracy of navigation or of searches on search terms by performing operations known as transformations on your data structures.

Oracle Commerce supports three types of transformations:

- `split-regex` - Splits the value of a dimension or property and assigns each part of the split value to a new dimension or property.
- `split-jsonpath` - Replaces the JSON value of a property with values that it extracts from the JSON.
- `concatenate` - Concatenates the values of dimensions or properties into a single new dimension or property.

One or more transformations can be performed on a single attribute (property or dimension). Transformations are in all cases optional.

The following sections describe each of these transformations.

Configure a split-regex transformation

A `split-regex` transformation divides the value of a dimension or property into separate values and assigns each separate value to a new dimension or property.

This transformation is useful when the Catalog contains an attribute whose value is a delimited list of values. Such a list is not usable for user navigation. However, the `split-regex` transformation can split the list into separate values that are useful for navigation.

The transformation uses a regex pattern to determine where to split the dimension or property value. The pattern can be either a single character or a more complex regex pattern; for example, the following pattern specifies that a value is split at any occurrence either of two dashes or of a comma and a semicolon:

```
"(\-\-)|[,;]"
```

Suppose that your product data provides the following different values for a dimension named country:

```
"country": "UK, US, DE"
```

Although such an assignment is acceptable in the Catalog, is it not useful for user navigation by dimension. The `split-regex` transformation, however, can convert this assignment into ones that are useful for navigation, as follows:

```
"country": "UK",
"country": "US",
"country": "DE"
```

The conversion above is performed by the following `split-regex` transformation:

```
{
  "ecr:type": "attributes-owner-folder",
  "country": {
    "ecr:type": "dimension",
    "mergeAction": "update",
    "indexingTransforms": [
      {
        "transform": "split-regex",
        "pattern": ","
      }
    ],
    "ignoreDuplicatePropertyValues": "true"
  }
}
```

The following table lists the properties of the `split-regex` transformation:

Property	Data type	Default value	Comments
`"transform"`	String	n/a	Required. Must be set to `"split-regex"` Example: `"transform": "split-regex"`
`"splitPattern"`	String	none	Required. The regex pattern must comply with `java.util.regex.Pattern`. Example: `"splitPattern": "[,;\t]"`

`"sourcePropertyNames"`	JSON Array	The name of the attribute to which this transform applies.	Optional.
			Example:
			`"sourcePropertyNames": [`
			`"shirt.color",`
			`"dress.color",`
			`"pants.color"`
			`]`

`"ignoreDuplicatePropertyValues"`	Boolean	True	Optional. Example: `"ignoreDuplicatePropertyValues":false`
`"trimWhitespace"`	Boolean	True	Optional. Example: `"trimWhitespace": false`
`"removeSourcePropertyValues"`	Boolean	True when the source property is the same as the attribute name. False otherwise.	Optional.

Configure a split-jsonpath transformation

The `split-jsonpath` transformation extracts values from a specified location in the JSON value of a specified property. The transformation replaces the JSON value of the property with the extracted values. No other properties are created or modified by this transformation.

For example, suppose that the following JSON has been assigned as the value of a property named `product.shoeSize`:

```
{
  "product.name": "Suede Shoes",
```

```
  "product.childSKUs": [
    {
      "sku.id": "1000-R-M8",
      "shoe.color": "Red",
      "shoe.size": "8"
    },
    {
      "sku.id": "1000-B-M8",
      "shoe.color": "Blue",
      "shoe.size": "8"
    },
    ...
  ]
}
```

The following `split-jsonpath` transformation replaces the JSON value of the property `product.shoeSize` with values that it extracts from the JSON value:

```
{
  "ecr:type": "attributes-owner-folder",
    "product.shoeSize": {
       "ecr:type": "dimension",
       "isAutogen": true,
       "indexingTransforms": [
           {
                "transform": "split-jsonpath",
                "sourcePropertyNames": [
                    "product.shoeData"
                 ],
                "splitPaths": [
                    "$['product.childSKUs'][*]['shoe.size']"
                ]
           }
       ]
    }
}
```

The `split-jsonpath` transformation assigns the extracted values to `product.shoeData` as follows:

```
"product.shoeSize": "8"
```

The following table lists the properties of a `split-jsonpath` transformation:

Property	Data type	Default value	Comments
`"transform"`	String	N.A.	Required. Must be set to `"split-jsonpath"` Example: `"transform": "split-jsonpath"`

Property	Data type	Default value	Comments
"sourcePropertyNames"	JSON Array	The attribute name to which this transform belongs	Optional. Example: "sourcePropertyNames": ["product.data"]
"removeSourcePropertyValues"	Boolean	True when the source property is identical to the attribute. False otherwise.	Optional. Example: "removeSourcePropertyValues": true
"ignoreDuplicatePropertyValues"	Boolean	True	Optional. Example: "ignoreDuplicatePropertyValues": false
"splitPaths"	JSON Array	Not applicable	Required when it is not an identity transform. The array values must comply with com.jayway.jsonpath.JsonPath. Example: "splitPaths": ["$['childSKUs'[*]['id']"] **Note:** When splitPaths is not specified for an identity transform, then the source property values will be parsed as a JSON Path and the resulting JSON key:value pairs will be added to the record.

Configure a concatenate transformation

A concatenate transformation can combine the following:

- The different values of a multi-assigned property, or the values of two or more different properties.
- The values of two or more dimensions.

The concatenated values are assigned to a single new dimension or property.

To include the new property in searches, you must add it to the `fields` array of your search interface.

You can concatenate localized properties with non-localized properties. The resulting property is localized. You cannot, however, concatenate properties that are localized to different locales.

For example, the following JSON example creates a record property named `all.colors` and assigns to it, as a single unitary value, the concatenated values of the existing record properties `shirt.color`, `dress.color`, and `pants.color`:

```
{
"ecr:type": "attributes-owner-folder",
"all.colors": {
    "propertyDataType": "ALPHA",
    "ecr:type": "property",
    "indexingTransforms": [
        {
            "transform": "concatenate",
            "sourcePropertyNames": [
                "shirt.color",
                "dress.color",
                "pants.color"
            ]
        }
    ]
  }
}
```

For example, suppose that values are assigned to the source properties as follows:

```
"shirt.color": "Red",
"dress.color": "Yellow, Black, Red",
"pants.color": "Orange"
```

The concatenate transformation above assigns the follow value to the output property `all.colors`:

```
"all.colors": "Red Yellow, Black, Red Orange"
```

The following table lists the properties of a concatenate transformation:

Property	Data type	Default value	Comments
`"transform"`	String	Not applicable	Required. Must be set to `"concatenate"` Example: `"transform": "concatenate"`

`"sourcePropertyNames"`	JSON Array	The attribute name to which this transformation belongs.	Optional.
			Example:
			`"sourcePropertyNames":`
			`[ "product.brand_name", "product.color" ]`

Property	Data type	Default value	Comments
`"removeSourcePropertyValues"`	Boolean	True when the source property is equivalent to the attribute name. False otherwise	Optional. Example: `"removeSourcePropertyValues": true`
`"ignoreDuplicatePropertyValues"`	Boolean	True	Optional. Example: `"ignoreDuplicatePropertyValues": false`

Disable transformations on a property or a dimension

The system owner can disable transformations on a property or dimension specified by non-system owners. To do this, specify an empty `indexingTransforms` attribute.

The following example disables transformations by non-system owners on the property named `product.color`:

```
"product.color": {
     "ecr:lastModifiedBy": "admin",
        "propertyDataType": "ALPHA",
        "indexingTransforms": [],
        "ecr:type": "property"
    }
```

Applying transformations

Transformations are applied by POST or PUT REST API calls of the following form:

```
POST | PUT /gsadmin/v1/${appName}/attributes/${owner}
```

For example:

```
POST /gsadmin/v1/attributes/owner1
```

When both the system owner and a non-system owner specify transformations, the transformation specified by the system owner is used. However, when only a non-system owner specifies a transformation, the non-system owner's transformation is used.

For more information about how to make POST and PUT calls, see Understand how to execute endpoints.

Configure which properties of aggregated records and their members are accessible to front end applications

You can configure which properties of aggregated records and their member records are accessible to services such as `guidedsearch` (which produces search results) and `typeahead` (which controls the typeahead function).

This can reduce the amount of data in the JSON returned by calls to these services. In particular, it can make inaccessible the data that is irrelevant to shoppers or is otherwise sensitive, such as profit margins.

Note: By default, all properties of aggregated records are accessible to services.

To configure which properties are accessible, execute the following endpoint with JSON content that configures which properties are accessible:

```
PUT /gsadmin/v1/cloud/pages/Default/services/service_name
```

Where *service_name* is the name of the service, such as `guidedsearch` or `typeahead`, whose access to properties is limited by this call.

For example:

```
PUT /gsadmin/v1/cloud/pages/Default/services/guidedsearch
```

In the JSON content, list the properties that you want to make accessible, as follows:

- List properties of aggregated records in the `attributes` field of the `resultsList` element. These properties hold values such as minimum and maximum prices. `fieldNames` is deprecated in this release.
- List properties of member records in the `childRecordAttributes` field of the `resultsList` element. These properties hold values that shoppers search on, such as names, descriptions, brands, and SKU-level data per product, such as swatches. `subRecordFieldNames` is deprecated in this release.
  The `maxChildRecords` specifies the number of child records to be returned. This configuration expects an integer from the following list:
  - 0 (no child records)
  - 1 (one child record)
  - -1 (all matching child records)

  If the specified value is greater than one (> 1) or less than negative one (< -1), it is reset to -1 indicating that all matching child records are to be included.

For example, execute the following endpoint with the JSON content shown to cause all matching SKU level data of member records to be accessible to the `guidedsearch` service:

```
PUT /gsadmin/v1/cloud/pages/Default/services/guidedsearch
"resultsList": {
            "@type": "ResultsList",
            "maxChildRecords": -1,
            "attributes": ["product.repositoryId", "product.displayName" ],
            "childRecordAttributes": [ "product.repositoryId",
"sku.repositoryId", "product.displayName",
            "sku.listPrice", "sku.activePrice",
"product.primaryFullImageURL", "product.route" ]
            }
```

You can see your changes to the configuration of aggregated records in you Preview storefront. To see the changes in your Production storefront, you must first publish the changes.

Note: Setting `maxChildRecords` to -1 can impact performance both in terms of response time and the response payload size. We recommend that you use `attributes`, `childRecordAttributes` and `maxChildRecords` together to minimize impact on performance.

Configure the order of facets

Oracle Commerce uses faceted navigation to support shoppers navigating on the storefront. The facets are returned in a defined order, which you can change with either the administration console or Admin API.

Facets are also returned dynamically, meaning if the results in the response contain appropriate data and that data can change the results. For example, a Brand facet is not returned if either no products had a brand, or all products had the same brand. If you want to hide a facet in certain situations, you can exclude it from the list for those situations and set the `showAll` flag to `false`.

This section describes how to configure the order of facets with the Admin API. To use the administration interface to perform this task instead, see Refine and order search results.

Note: Some Commerce REST APIs use the term dimension instead of facet, but the two are interchangeable from a functional perspective.

Specify a custom order for facets

To specify a non-default display order for selected facets in your application, follow these steps:

1. Use one of the following endpoints to export configuration of the Guided Navigation catalog in JSON format or in ZIP format:

   ```
   GET /gsadmin/v1/cloud/content/facets/default (JSON format)
   ```

   ```
   GET /gsadmin/v1/cloud/content/facets/default.zip (ZIP format)
   ```

   The default configuration of the Guided Navigation content item is returned. The `navigation[]` attribute is empty. When the navigation attribute is empty, the facets in the refinements list are not explicitly ordered; instead, they appear in an order determined by the system. For information about the attributes in this configuration, see Attributes of the Guided Navigation Content Item.

2. In the navigation attribute, include a `RefinementMenu` element for each facet that you want to include among the explicitly ordered facets. Arrange the `RefinementMenu` elements in the order in which you want the corresponding facets to appear. See Example of custom facet ordering.

3. Use one of the following endpoints to import configuration of the Guided Navigation content item in JSON format or in ZIP format:

   ```
   PUT /gsadmin/v1/cloud/content/facets/default (JSON format)
   ```

   ```
   PUT /gsadmin/v1/cloud/content/facets/default.zip (ZIP format)
   ```

Note: If the `GuidedNavigation` content item does not exist, you must use the POST method to create it. You cannot, however, use POST to update an existing content item.

4. View the changes in your Preview environment to verify that they are correct.
5. Publish your catalog to promote your changes to your production environment.

Example of custom facet ordering

For example, if you want the four following facets to appear in the order `Category`, `Price Range`, `Brand`, and `Color`, modify the `navigation[]` attribute as follows:

```
"navigation": [
   {
        "@type": "RefinementMenu",
        "dimensionName": "product.category"
   },
   {
        "@type": "RefinementMenu",
        "dimensionName": "product.priceRange"
   },
   {
        "@type": "RefinementMenu",
        "dimensionName": "product.brand"
   },
   {
        "@type": "RefinementMenu",
        "dimensionName": "product.color"
   }
]
```

Attributes of the Guided Navigation Content Item

The following table summarized that attributes of the `GuidedNavigation` content item.

Attribute	Value
`@type`	Required. Must be set to `GuidedNavigation`
`@name`	Optional. A name for this rule that is displayed in tools.
`showAll`	Optional. `True` (the default) causes all facets applicable to the shopper's current navigation state to be displayed in refinement lists, after the facets explicitly ordered by the navigation attribute. The facets not included in the navigation attribute are ordered by the system. False limits the facets included in the refinements list to those included in the `navigation` attribute.

Attribute	Value
navigation	Required. A list of facet objects. Each facet object has the following: • a @type attribute (required) set to RefinementMenu. • a dimensionName attribute (required). The name should be the attribute name as specified under the /attributes folder of your application. In the refinements list, the facets appear in the order in which the facet objects are listed in the navigation attribute.
triggers	Not applicable. Do not change default.
priority	Not applicable. Do not change default.

Configure the order of facet values

You can configure how facets of any given dimension are sorted in refinement lists at run time.

You can sort the facet values either alphabetically, by frequency (number of matching results for each facet value), by display order (previously known as rank), or by statistical significance. In addition, these can be sorted in ascending or descending order.

To change the ordering of a facet's values, you use a REST API to specify a sort option and, optionally, a sort order to apply to all the facet values in the facet.

The sort option determines the order of the facet values with respect to each other. The following sort options are available:

- displayName or alpha (alphabetical) – Sorts facet values in alphabetical order using the facet value display name. This is the default order for the Category facet
- count or freq - Sorts facet values by the number of records that are tagged to each. This is the default order for new facets and for facets other than Category and Price Range.
- displayOrder or rank – Sorts facet values in the Category facet by the static value provided separately using the facet data API.
- sig – Sorts facet values according to their statistical significance. For more information, see Order facet values by statistical significance.

The sort order can be either asc (ascending) or desc (descending). If you do not specify an order, the sort order is ascending for displayName and descending for count and displayOrder, by default.

In some cases, a combination of sort option and sort order can assign two or more facet values to the same location in a refinements list. To further sort the facet values in such cases, you can specify a second sort option and sort order pair.

The sort option and sort order are specified as values of the facet's `displayConfig` parameter, as in the following example:

```
"displayConfig": {
     "sort":
        "count,desc;displayName,asc"
},
```

The example above specifies that facet values first be sorted in descending order of frequency; any facet values that require further sorting are then sorted in ascending alphabetical order.

Note: You can specify any number of pairs of sort option and sort order values, but more than two pairs are seldom needed.

Dynamically order a facet's values

To configure how the facet values of a specified facet are to be ordered, follow these steps:

1. Export the configuration of the ATG owner's attributes, using the following endpoint:
   `GET /gsadmin/v1/cloud/attributes/ATG`
2. In the exported `_.json` file, copy the facet attribute that you are interested in.
3. Add or update this attribute's `displayConfig` parameter, and specify the sort option and sort order for the facet values in that facet. For example:
   `"displayConfig": {"sort": "count,desc;displayName,asc"},`
4. Upload this updated configuration definition to the following endpoint:
   `PUT /gsadmin/v1/cloud/attributes/system/<facet-name>`
5. Verify your changes have taken effect by calling the following endpoint and ensuring it shows the new `displayConfig` property in the list of system attributes:
   `GET /gsadmin/v1/cloud/attributes/system`
6. Run an index to have your changes take effect:
   `POST /ccadmin/v1/search/index { "op": "partial" }`

Facet values ordered by `displayOrder`

If the facet is ordered by `displayOrder` a `displayConfig` is specified as follows:

```
"displayConfig": {  "sort":
        "displayOrder,desc"
    }
```

Then, an additional step is required: You must provide the relative order of each value. To do so, follow these steps:

1. Update the Facets data store with a `displayOrder` for each value using the API endpoint `POST /gsdata/v1/cloud/facets/<facet-name>`. For example, to update a Decade facet with values of 80s, 90s, 00s, 10s, and so on in logical order, you would use the following endpoint and payload:

   ```
   POST /gsdata/v1/cloud/facets/product.x_decade
   {
       "items": [
   ```

```
{
    "key" : "60s",
    "displayOrder" : "6"
},
{
    "key" : "70s",
    "displayOrder" : "5"
},
{
    "key" : "80s",
    "displayOrder" : "4"
},
{
     "key" : "90s",
    "displayOrder" : "3"
},
{
     "key" : "00s",
    "displayOrder" : "2"
},
{
     "key" : "10s",
    "displayOrder" : "1"
}
] }
```

2. Run an index either by triggering a publish event, or by calling the index endpoint, for example:
   `POST /ccadmin/v1/search/index { “op”: “partial” }`
3. Once the index has completed, as the order is descending (`sort: displayOrder,desc`) values will be returned by the value of `displayOrder` sorting from high to low, for example:

   1. 60s
   2. 70s
   3. 80s
   4. 90s
   5. 00s
   6. 10s

Notes:

- Values will only be available once an index has run, so if the facet is new and a publish has not been run, this will return an error.
- If you need to retrieve the values for the facet, you can call the “dimvals” endpoint, for example:
  `GET /gsadmin/v1/cloud/dimvals/product.x_decade/children`

As noted above, multiple sort options can be provided, therefore you can configure `displayOrder` as the primary sort option, and have a secondary sort defined too. With this combination, facet values will be divided into two buckets, the first containing values with a `displayOrder` and ordered using this property, followed by all values that

do *not* have a `displayOrder` and ordered by the secondary sort (for example, alphabetically or by count).

Facet configuration example

This section describes an excerpt from an example `_.json` file and illustrates how a facet is configured. The facet is named `camera.color`.

The `displayConfig` parameter specifies that facet values in this facet are sorted in descending order of frequency; ties are sorted in ascending alphabetical order. For example, suppose that two facet values, “Action Sports Cameras” and “Binoculars” each have 44 records tagged to them. After being assigned the same location in the list by descending order of frequency, they are sorted in ascending alphabetical order. Thus “Action Sports Cameras” comes immediately before “Binoculars” in the list.

The configuration of the `camera.color` facet includes the parameter `"mergeAction" : "update"`. This parameter must be included when the owner of the facet is system, as is currently required.

```
{
   "camera.color": {
     "isWildcardEnabledInRecordSearch": true,
     "displayOrder": 4,
     "displayConfig": {
       "sort": "freq,desc;alpha,asc"
     },
     "sourcePropertyNames": ["camera.Color of product"],
     "isAutogen": true,
     "isRecordSearchEnabled": true,
     "ecr:type": "facet",
     "mergeAction" : "update"
   }
}
```

Note: Do not modify facet parameters not discussed in this section unless you are certain that you have complete and accurate knowledge about those parameters.

Order facet values by statistical significance

To make it easier for shoppers to find the products that best meet their requirements, you can sort facet values (also known as dimension values) according to their statistical significance.

When facet values are sorted by statistical significance, shoppers are first presented with the facet values that are most relevant to their current navigation state.

Highlight relevant facet values

Sorting facet values by statistical significance is a useful technique when you want to highlight facet values that are relevant to the shopper’s current search rather than facet values that are generally popular. Sorting facet values by their statistical significance is especially useful when a facet has a large number of facet values. In such a case, the shopper is aided by having the facet values that are most relevant to the current navigation state presented first.

Sorting facet values by frequency, on the other hand, can be useful when the number of facet values is smaller -- for example, small enough to be displayed in a single facet values list.

The value of sorting by statistical significance is illustrated by the following use cases.

In a catalog with a `feature` facet containing 2,000 possible values, you can do the following:

- Highlight the features that are most relevant to a search for “waterproof camera”, such as “waterproof”, “shockproof”, “dustproof”, or “GPS-enabled”.
- Highlight the features that are most relevant to a search for “compact camera”, such as “built-in flash”, “autofocus”, “portrait mode”, or “landscape mode”.

In a catalog with a facet containing 20,000 possible values, you can do the following:

- Highlight the product tags that are most relevant to a search for “landscape photograph”, such as “mountain”, “forest”, “ocean”.
- Highlight the product tags that are most relevant to a search for “Venice canvas print”, such as “canal”, “boat”, or “reflection”.

In a catalog with a `brand` dimension containing 200 possible values, you can do the following:

- Highlight the brands that specialize in digital SLR cameras when a shopper searches for “dslr camera”.
- Highlight the brands that specialize in sports cameras when a shopper searches for “waterproof camera”.

Calculate statistical significance

A facet value's statistical significance is based on the difference between the background frequency of the facet value from its foreground frequency. Frequencies are defined as follows:

- The background frequency is the number of records in the entire catalog that match the facet value.
- The foreground frequency is the number of records in the current search that match the facet value.

A facet value's statistical significance is calculated only after all specified record filters and security filters have been applied to the set of records in the catalog. Because different sets of filters can be applied to a catalog on different sites, the statistical significance of a facet value can vary from site to site.

A facet value is considered statistically significant if its foreground frequency is higher than its background frequency. The statistical significance of a facet value increases as the relative difference between foreground and background frequencies of that facet value increases.

Note that the number of records tagged to each facet value is ignored when facet values are sorted by statistical significance. For example, three facet values might be sorted by statistical significance as follows:

```
naugahyde (10)
polyester (7)
leather (23)
```

For example, suppose that a brand facet has facet values named Rugged Cameras and Acme Camera Corporation, and that a shopper is shopping for “waterproof cameras”. The catalog, which contains a total of 100,000 records, includes the following:

- 100 records for waterproof cameras manufactured by Acme Camera Corporation, out of a total of 2,000 records for cameras of any type manufactured by Acme Camera Corporation.
- 20 records for waterproof cameras manufactured by Rugged Cameras. There are no other records in the catalog for cameras manufactured by Rugged Cameras, which makes only waterproof cameras

Selecting the facet value ‘Acme Camera Corporation’ would produce 100 matching records.

Selecting the facet value ‘Rugged Cameras’ would produce only 20 matching records. Nevertheless, you might want to list Rugged Cameras in the facet values list before Acme Camera Corporation, because it has a higher statistical significance.

Rugged Cameras has a higher statistical significance because its foreground frequency (20 matches out of 20 records in the search) is so much greater than its background frequency (20 matches out of 100,000 records in the catalog). Acme records shows a much lesser increase in foreground frequency over background frequency.

In most cases, a higher statistical significance of a facet value is a sign of a quality that has relevance or value for the shopper’s search. In the example above, the waterproof cameras manufactured by Rugged Cameras, which have a greater statistical significance, are preferable to those manufactured by Acme Camera Corporation because waterproof cameras are the specialty of Rugged Cameras, while Acme Camera Corporation is a generic manufacturer that does not specialize in one type of camera.

As with all configuration of facet sorting, sorting by statistical significance must be configured on a per-dimension basis. There is no mechanism to configure a default sort behavior across all dimensions.

Configure sorting by statistical significance.

To configure sorting by statistical significance, use the REST API for setting attributes. For more information, see Sample search and navigation REST API.

To import configuration for sorting facet values by statistical significance, execute an endpoint similar to the following:

```
http://host:port/gsadmin/v1/cloud/attributes/system/facet_name
```

where `facet_name` is the name of the facet whose facet values are to be sorted.

The body of the request must be a `dimension` object definition that includes a `displayConfig` attribute; for example:

```
 {
    "ecr:type": "dimension",
    "mergeAction": "UPDATE",
    "displayConfig": {
      "sort": "sig,desc"
    }
}
```

where `sig,desc`, specifies that facet values are sorted in descending order of their statistical significance. Because descending order is the default for sorting by statistical significance, you can specify `sig` instead of `sig,desc`.

Get statistical significance values for debugging

When you sort facet values by statistical significance, the statistical significance of each facet value is assigned to its `DGraph.Significance` property. The value of the `DGraph.Significance` property can be useful for debugging.

The following example illustrates the `Dgraph.Significance` property of the Rugged Sports Camera and Acme Camera Corporation facet values:

```
{
    "@type": "Facet valueMenu",
    "displayName": "Brand",
    "dimensionName": "product.brand",
    "multiSelect": true,
    "refinements": [
      {
        "label": "Rugged Sports Cameras",
        "link": "?Ntt=waterproof+cameras",
        "count": 20,
        "properties": {
        "DGraph.Significance": "99.9",
        "DGraph.Spec": "Rugged Sports Cameras"
        }
     },
     {
        "label": "Acme Camera Corporation",
        "link": "?Ntt=waterproof+cameras",
        "count": 100,
        "properties": {
           "DGraph.Significance": "12.0",
           "DGraph.Spec": "Acme Camera Corporation"
         }
      }
    ]
}
```

Add metadata to facet values

You can associate metadata with facet values so it can be displayed on your store.

Associating metadata with facet values is most commonly used to display images instead of text. For example, you might want to display swatch images instead of names for Color facets, or display stars instead of a numeric value for an Average Customer Rating facet.

Any key:value pair can be added as metadata to be returned along with the standard facet value information; however, the key name must contain an underscore character, for example `x_imageUrl` or `tool_tip_text`, to ensure it does not conflict with existing system keys. JSON objects cannot be used as a value, however multiple values can be provided as a list, for example:

```
"x_imageUrl" : ["/images/one.png", "/images/two.png" ]
```

Add metadata to a facet value

The following procedure shows an example of adding images to an “Average Customer Rating” facet.

1. Update the Facets data store with metadata for each value using the following - `POST /gsdata/v1/cloud/facets/<facet-name>`, using the system property of `key` to define which value the metadata should be added to, for example, `"key": "Red"` for the value `Red`.

```
  POST /gsdata/v1/cloud/facets/product.x_averageCustomerRating
{
    "items": [
 {
     "key" : "5",
     "x_imageUrl" : "/general/5-star.png"
 },
 {
     "key" : "4",
     "x_imageUrl" : "/general/4-star.png"
 },
 {
     "key" : "3",
     "x_imageUrl " : "/general/3-star.png"
 },
 {
     "key" : "2",
     "x_imageUrl " : "/general/2-star.png"
 },
 {
     "key" : "1",
     "x_imageUrl " : "/general/1-star.png"
 }
] }
```

2. Run an index either by triggering a publish event, or by calling the index endpoint, for example, `POST /ccadmin/v1/search/index { "op": "partial" }`

Notes:

- Values will only be available once an index has run, so if the facet is new and a publish has not been run, this will return an error.
- If you need to retrieve the values for the facet, you can call the “dimvals” endpoint, for example: `GET /gsadmin/v1/cloud/dimvals/product.x_decade/children`

Use metadata on the storefront

Once you've added the metadata to facet values and the search indexing operation has completed, the metadata will be returned in the search API endpoint responses alongside the existing facet value data. Metadata can also be easily accessed in storefront widgets.

The following snippet shows a knockout `if` binding that displays an image from the Commerce media library. When this binding is added to the Product Listing widget on the Search Results layout, displays an image when iterating through the facet values.

```
<ko:if: $data.properties['x_imageUrl']-->
      <img data-bind="attr:{src: 'file/'
+$data.properties['x_imageUrl']}" />
</ko>
```

Create custom range facets

By default, a facet displays all the unique values as a list. It can be useful to group all values between a minimum and maximum value and display a range of values as a single link. A common example of this is price ranges, for example, $10 - $20.

You create a custom range facet to display a group of values in a single link. To define a custom range facet, you must first create a new facet by calling the following Admin API endpoint:

```
POST /gsadmin/v1/cloud/attributes/system/<facet-name>.
```

Specify the following properties for the facet:

- `ecr:type`: Required. Value must be set to `dimension`.
- `isAutogen`: Required. Value must be set to `false`.
- `displayConfig`: See Configure the order of facet values for more information about this property.
- `context`: Optional. Set to `priceGroup` if the ranged facet is for pricing, or `locale` if the facet is multi-language.
- `rangeComparisonType`: Required. Value must be set to `FLOAT`.
- `sourcePropertyNames`: Required. List of one or more source properties, for example `sku.activePrice`, or `product.x_megapixels`.

Next, define each value in the Facets data store with the following Admin API endpoint:

```
POST /gsdata/v1/cloud/facets/<facet-name>
```

Specify the following properties for the facet values:

- `key`: Required. System ID for the facet value. Must be unique.
- `displayName`: Required. The text to display for the facet value.
- `priceGroup`: Optional. If the new facet will display prices, the price group needs to be explicitly defined. See Understand Price Navigation for more information.
- `displayOrder`: Optional. A `displayOrder` value can be assigned to sequence the facet value relative to the other values. See Configure the order of facet values for more information.
- `lowerBound`: Required. Minimum value to match for the range.

- `upperBound`: Required. Maximum value to match for the range.

Example: Create a custom price range

This section describes a step-by-step example of adding a custom price range facet.

First, create a new facet by issuing a `POST` request to the Attributes system endpoint:

```
POST /gsadmin/v1/cloud/attributes/system/Price
{
        "ecr:type": "dimension",
        "isAutogen" : false,
        "displayConfig": {
            "sort": "displayOrder,asc"
        },

        "context" : ["priceGroup"],
        "rangeComparisonType" : "FLOAT",
        "sourcePropertyNames" : [ "sku.activePrice" ]
}
```

Next, create the associated ranged values by issuing a `POST` request to the Facets endpoint:

```
POST /gsdata/v1/cloud/facets/Price
{
  "items": [
    {
      "displayName": "$0 - $25",
      "priceGroup" : "defaultPriceGroup",
      "displayOrder": "1",
      "key" : "0-25",
      "lowerBound" : "0.00",
      "upperBound" : "25.00"
    },
    {
      "displayName": "$25 - $50",
      "priceGroup" : "defaultPriceGroup",
      "displayOrder": "2",
      "key" : "25-50",
      "lowerBound" : "25.00",
      "upperBound" : "50.00"
    },
   {
      "displayName": "$50 - $100",
      "priceGroup" : "defaultPriceGroup",
      "displayOrder": "3",
      "key" : "50-100",
      "lowerBound" : "50.00",
      "upperBound" : "100.00"
    },
   {
      "displayName": "$100+",
      "priceGroup" : "defaultPriceGroup",
      "displayOrder": "4",
      "key" : "over100",
```

```
      "lowerBound" : "100.00",
      "upperBound" : "999999.00"
    },
    {
      "record.action": "OCCForceFlush"
    }
  ]
}
```

Finally, initiate an indexing operation, either by triggering a publish event, or by calling the `index` endpoint. The following code sample performs incremental indexing:

```
POST /ccadmin/v1/search/index { "op": "partial" }
```

Once the indexing operation is complete, the new facet will be returned in the response.

Understand Price Navigation

Price properties in search, such as `sku.activePrice` and `product.listPrice`, are assigned dynamically, based on the relevant price group. If your site uses multiple price groups, either the default price group or a non-default price group will be dynamically assigned for the current shopper, and all price properties in search will contain the values for products and SKUs based on that price group.

For price navigation, you can assign each custom price range facet to only one price group. If your site has many price groups, a better approach is to use the range filter parameter `Nf` and let shoppers specify the minimum and maximum price directly, using a slider, text boxes, or From and To dropdowns, then use a range filter to restrict the products by price. For example: `Nf=sku.activePrice|BTWN+100+200`.

Configure the ranking of records in search results

You can determine the order in which records appear in search results by configuring a relevance ranking strategy.

The relevance ranking strategy is an ordered list of one or more relevance ranking modules, each of which uses different criteria to sort the records in the search results.

Note: Oracle Commerce includes a basic default relevancy ranking strategy. Modify this default to order search results in a way that is helpful to your shoppers. Successful strategies typically include the modules `Phrase`, `MaxField`, `Glom`, `Static`, and `WFreq`.

This section covers the following topics:

- Understand how relevance ranking modules sort results
- Configure your relevance ranking strategy
- Understand each relevance ranking module in detail

Understand how relevance ranking modules sort results

Relevance ranking modules are applied in the order in which they are listed in the relevance ranking strategy. For example, the following JSON format configuration of a relevance ranking strategy,

```
"relRankStrategy":
"exact(considerFieldRanks),glom,static(quantity_sold,descending)"
```

invokes modules named `exact`, `glom`, and `static`, in that order.

The first module applies its criteria to sort records into various strata. Each stratum contains records that have the same relevance ranking according to the first module's criteria. The next module sorts the records in each stratum into substrata according to its own criteria; each substratum contains records of the same relevance ranking.

The sorting continues in this fashion until all modules have been applied, or until there are no further ties among records. If any ties remain after all modules have been invoked, the ties are resolved by a default sorting rule.

The `field` and `maxfield` modules take into account the priority of records. A record's priority corresponds to the position in the search interface's `fields` array of the member that the record matches. Records matched to members closer to the beginning of the `fields` array have higher priority than records matched to members closer to the end. If a record matches more than one member, its priority is based on the member that is closest to the beginning of the `fields` array.

The `exact`, `first`, `nterms`, and `proximity` modules can optionally take a parameter named `considerFieldRanks`. The `considerFieldRanks` parameter indicates that the module should further sort records according to their priority, after the module has sorted records according to its own criteria. In the relevance ranking strategy, the parameter is specified in parentheses after the module name; for example:

```
"relRankStrategy": "exact(considerFieldRanks),glom,static(quantity_sold,
   descending)"
```

For information about the relevance ranking modules, see Understand each relevance ranking module in detail.

Understand which modules are more useful for commerce applications

The most commonly used modules in commerce applications are as follows:

- `phrase` (all options turned on)
- `glom`
- `maxfield`
- `static`

The following modules are less commonly used:

- `field`
- `wfreq`

The following modules are not ordinarily useful in commerce applications:

- `first`
- `interp`
- `nterms`
- `proximity`
- `stem`
- `thesaurus`

Oracle recommends against the use of the following modules:

- `exact`, because of its effect on performance. Use `phrase` instead.
- `freq`, because it adds up the counts of every word, affecting performance.

Configure your relevance ranking strategy

You configure your relevance ranking strategy by exporting the entire search configuration for the cloud application, editing the part of the exported configuration that applies to the relevance ranking strategy, and then re-importing the entire search configuration. To do this, follow these steps:

1. Issue the following `GET` command, which exports the entire search configuration for the cloud application in a `ZIP` file:

   ```
   GET /gsadmin/v1/cloud/content/system/rankingRules/
   defaultStandardSearch
   ```

   For more information about the GET endpoint, see Export all configuration in ZIP format.
2. Back up the ZIP file before opening it or extracting any of its contents.
   IMPORTANT: When you re-import the edited search configuration, you overwrite all existing search configuration. For this reason, it is important to keep a back-up of the original configuration.
3. Unzip the zip file and extract the JSON file containing the search configuration.
4. Open the JSON file containing the search configuration and find the `relRankStrategy` attribute of the `resultsList` object, in the `contentItem` object named “Guided Search Service”.
   Note: If the `resultsList` object or `therelRankStrategy` attribute is not defined, you must add them in the location shown in this example.
5. Edit the value of the `relRankStrategy` attribute to specify the relevance ranking modules that you want the strategy to comprise. The order in which you specify the modules is significant. For more information, see Understand how relevance ranking modules sort results .
6. Zip up the entire search configuration, including the edits that you made to the configuration of the relevance ranking strategy.
7. Initiate the following POST command, which imports the search configuration in the ZIP file:

   ```
   POST /gsadmin/v1/cloud.zip
   ```

8. If your relevance ranking strategy does not produce that results that you intended, make a copy of your backed up search configuration, edit the relevance ranking strategy in the copy to produce the intended results, and import the copy.

Understand each relevance ranking module in detail

This section contains detailed descriptions of the relevance ranking modules.

exact module

The `exact` module groups results into three strata based on how well they match the query string. This includes the following:

- The highest stratum contains results whose complete text matches the user's query exactly.
- The middle stratum contains results that contain the user's query but are not an exact match.
- The lowest stratum contains all other types of matches, such as matches that would not be matches without synonyms.

The `exact` module can optionally be specified with the `considerFieldRanks` parameter, as follows:

```
exact(considerFieldRanks)
```

Specifying this parameter causes the `exact` module to sort records according to their priorities in your search interface after it has sorted them according to its own criteria.

Important: The `exact` module is computationally expensive, especially on large text fields. It is intended for use only on small text fields (such as dimension values or small property values such as part IDs). Use of this module in these cases will result in very poor performance and/or application failures due to request timeouts. The `phrase` module, with and without approximation turned on, does similar but less complex ranking that can be used as a higher performance substitute.

field module

The `field` module ranks records according to their priority in your search interface. A record's rank is the priority of the search interface member that the record matches. If a record matches more than one member, the record's rank corresponds to the highest priority among the matching members.

For example, suppose that:

- Record A matches the third, sixth, and eighth members in the fields array of your search interface.
- Record B matches the first, fourth, and seventh members.

The earliest member that Record B matches is the first member, and the earliest member that Record A matches is the third member. As a result, Record B has the higher priority and appears before Record A in the search results.

first module

Note: The `first` module is not commonly used in commerce applications.

Designed primarily for use with unstructured data, the first module ranks documents by how close the query terms are to the beginning of the document. The first module groups its

results into strata of different sizes. The strata are not the same size, because while the first word is probably more relevant than the tenth word, the 301st is probably not significantly more relevant than the 310th word. This module assumes that the closer a word is to the beginning of a document, the more likely it is to be relevant.

The `first` module works as follows:

When the query has a single term, the `first` module retrieves the first absolute position of the word in the document, then calculates which stratum contains that position. The score for this document is based upon that stratum; earlier strata are better than later strata.

When the query has multiple terms, `first` determines the first absolute position for each of the query terms, and then calculates the median position. This median is treated as the position of this query in the document and can be used with stratification as described in the single word case.

With query expansion (using stemming or the thesaurus), the `first` module treats expanded terms as if they occurred in the source query. For example, the phrase glucose intolerance would be corrected to glucose intloerance (with intloerance spell-corrected to intolerance). first then continues as it does in the non-expansion case. The first position of each term is computed and the median of these is taken.

In a partially matched query, where only some of the query terms cause a document to match, `first` behaves as if the intersection of terms that occur in the document and terms that occur in the original query were the entire query. For example, if the query cat bird dog is partially matched to a document on the terms cat and bird, then the document is scored as if the query were cat bird. If no terms match, then the document is scored in the lowest strata.

The `first` module is supported for wildcard queries.

The `first` module can optionally be specified with the `considerFieldRanks` parameter. Specifying this parameter causes the exact module to sort records according to their priorities in your search interface after it has sorted them according to its own criteria.

freq module

The `freq` (frequency) module provides result scoring based on the number of occurrences of the user's query terms in the result text.

Results with more occurrences of the user search terms are considered more relevant.

The score produced by the `freq` module for a result record is the sum of the frequencies of all user search terms in all fields (properties or dimensions in the search interface in question) that match a sufficient number of terms. The number of terms depends on the match mode, such as all terms in a `MatchAll` query, a sufficient number of terms in a `MatchPartial` query, and so on. Cross-field match records are assigned a score of zero. Total scores are capped at 1024; in other words, if the sum of frequencies of the user search terms in all matching fields is greater than or equal to 1024, the record gets a score of 1024 from the freq module.

For example, suppose we have the following record:

```
{Title="test record", Abstract="this is a test", Text="one test this
is"}
```

A `MatchAll` search for “test this” causes `freq` to assign a score of 4, because this and test occur a total of 4 times in the fields that match all search terms (`Abstract` and `Text`, in this case). The number of phrase occurrences (just one in the `Text` field) does not matter, only the sum of the individual word occurrences. Also note that the occurrence of test in the `Title` field does not contribute to the score, since that field did not match all of the terms.

A `MatchAll` search for one record would hit this record, assuming that cross field matching was enabled. But the record would get a score of zero from `Freq`, because no single field matches all of the terms. `Freq` ignores matches due to query expansion (that is, such matches are given a rank of 0)

`glom` module

The `glom` module ranks single-field matches ahead of cross-field matches and also ahead of non-matches (records that do not contain the search term). It serves as a useful tie-breaker function in combination with the `maxfield` module and is commonly used in commerce applications.

If you want a strategy that ranks single-field matches first, cross-field matches second, and no matches third, then use the `glom` module followed by the `nterms` module. `glom` treats all matches the same, whether or not they are due to synonyms or other forms of query expansion.

The `glom` module considers a single-field match to be one in which a single field has enough terms to satisfy the conditions of the match mode. or this reason, in `MatchAny` search mode, cross-field matches are impossible, because a single term is sufficient to create a match. Every match is considered to be a single-field match, even if there were several search terms.

For `MatchPartial` search mode, if the required number of matches is two, the `glom` module considers a record to be a single-field match if it has at least one field that contains two or more or the search terms. You cannot rank results based on how many terms match within a single field.

interp module

The `interp` (interpreted) module assigns a score to each result record based on the query processing techniques used to obtain the match. These matching techniques include partial matching, cross-attribute matching, thesaurus, and stemming matching.

Specifically, the Interpreted module ranks results as follows:

1. All non-partial matches are ranked ahead of all partial matches.
2. Within the above strata, all single-field matches are ranked ahead of all cross-field matches.
3. Within the above strata, all thesaurus matches are ranked below all non-thesaurus matches.
4. Within the above strata, all stemming matches are ranked below all non-stemming matches.

Note: Because the `interp` module comprises the matching techniques of the `spell`, `glom`, `stem`, and `thesaurus` modules, there is no need to add them to your relevance ranking strategy if you are using `interp`.

proximity module

The `proximity` module ranks how close the query terms are to each other in a document by counting the number of intervening words. It is designed primarily for use with unstructured data.

Like the first module, the `proximity` module groups its results into variable sized strata, because the difference in significance of an interval of one word and one of two words is usually greater than the difference in significance of an interval of 21 words and 22. If no terms match, the document is placed in the lowest stratum.

Single words and phrases get assigned to the best stratum because there are no intervening words. When the query has multiple terms, `proximity` behaves as follows:

1. All of the absolute positions for each of the query terms are computed.
2. The smallest range that includes at least one instance of each of the query terms is calculated. This range's length is given in number of words. The score for each document is the stratum that contains the difference of the range's length and the number of terms in the query; smaller differences are better than larger differences.

Under query expansion (that is, stemming and the thesaurus), the expanded terms are treated as if they were in the query, so the proximity metric is computed using the locations of the expanded terms in the matching document.

For example, if a user searches for "big cats" and a document contains the sentence, "Big Bird likes his cat" (stemming takes cats to cat), then the proximity metric is computed just as if the sentence were, "Big Bird likes his cats." The `proximity` module scores partially matched queries as if the query contains only the matching terms. For example, if a user searches for "cat dog fish" and a document is partially matched that contains only cat and fish, then the document is scored as if the query "cat fish" had been entered.

Note: The `proximity` module does not work with Boolean searches, cross-field matching, or wildcard search. It assigns all such matches a score of zero.

`maxfield` module

This module ranks based on field priority and gives equal weight to cross-field matches.

The `maxfield` (Maximum Field) module behaves in the same way as the field module, except in how it scores cross-field matches. Unlike `field`, which assigns a static score to cross-field matches, `maxfield` selects the score of the highest-ranked field that contributed to the match.

`nterms` module

The `nterms` (number of terms) module assigns rank based on the number of terms that it finds.

The `nterms` module ranks matches according to how many query terms they match. For example, in a three-word query, results that match all three words will be ranked above results that match only two, which will be ranked above results that match only one, which will be ranked above results that had no matches.

numfields module

The `numfields` (number of fields) module ranks results based on the number of fields in the associated search interface in which a match occurs.

Note that the whole-field is counted rather than cross-field matches. Therefore, a result that matches two fields matches each field completely, while a cross-field match typically does not match any field completely.

`numfields` treats all matches the same, whether or not they are due to query expansion. The `numfields` module is only useful in conjunction with record search operations.

phrase module

The `phrase` module states that results containing the user's query as an exact phrase, or a subset of the exact phrase, should be considered more relevant than matches simply containing the user's search terms scattered throughout the text.

Records that have the phrase are ranked higher than records which do not contain the phrase.

The `phrase` module has a variety of options that you use to customize its behavior. The phrase options are as follows:

- Rank based on length of subphrases
- Use approximate subphrase/phrase matching
- Apply spell correction, thesaurus, and stemming

The various options can go in parentheses, including `considerFieldRanks`.

Ranking based on length of subphrases

When you configure the phrase module, you have the option of enabling subphrasing.

Subphrasing ranks results based on the length of their subphrase matches. In other words, results that match three terms are considered more relevant than results that match two terms, and so on. A subphrase is defined as a contiguous subset of the query terms the user entered, in the order that he or she entered them. For example, the query "fax cover sheets" contains the subphrases "fax", "cover", "sheets", "fax cover", "cover sheets", and "fax cover sheets", but not "fax sheets".

Content contained inside nested quotes in a phrase is treated as one term. For example, consider the following phrase:

the question is "to be or not to be"

The quoted text ("to be or not to be") is treated as one query term, so this example consists of four query terms even though it has a total of nine words.

When subphrasing is not enabled, results are ranked into two strata: those that matched the entire phrase and those that did not.

Using approximate matching

Approximate matching provides higher-performance matching, as compared to the standard phrase module, with somewhat less exact results.

With approximate matching enabled, the phrase module looks for phrase matches in a limited number of positions in each result, rather than all the positions. Only this limited number of

possible occurrences is considered, regardless of whether there are later occurrences that are better, more relevant matches.

The approximate setting is appropriate in cases where the runtime performance of the standard phrase module is inadequate because of large result contents and/or high site load.

Applying thesaurus and stemming

Applying thesaurus and stemming adjustments to the original phrase is generically known as query expansion.

With query expansion enabled, the phrase module ranks results that match a phrase's expanded forms in the same stratum as results that match the original phrase. Consider the following example:

- A thesaurus entry exists that expands "US" to "United States".
- The user queries for "US government".

The query "US government" is expanded to "United States government" for matching purposes, but the phrase module gives a score of two to any results matching "United States government" because the original, unexpanded version of the query, "US government", only had two terms.

Summary of phrase option interactions

The three configuration settings for the phrase module can be used in a variety of combinations for different effects. The following table summarizes the behavior of each combination.

Subphrase	Approximate	Expansion	Description
Off	Off	Off	Default. Ranks results into two strata: those that match the user's query as a whole phrase, and those that do not.
Off	Off	On	Ranks results into two strata: those that match the original, or an extended version, of the query as a whole phrase, and those that do not.
Off	On	Off	Ranks results into two strata: those that match the original query as a whole phrase, and those that do not. Look only at the first possible phrase match within each record.

Subphrase	Approximate	Expansion	Description
Off	On	On	Ranks results into two strata: those that match the original, or an extended version, of the query as a whole phrase, and those that do not. Look only at the first possible phrase match within each record.
On	Off	Off	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched subphrase.
On	Off	On	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched subphrase. Extend subphrases to facilitate matching but rank based on the length of the original subphrase (before extension). Note This combination can have a negative performance impact on query throughput.
On	On	Off	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched subphrase. Look only at the first possible phrase match within each record.

Subphrase	Approximate	Expansion	Description
On	On	On	Ranks results into N strata where N equals the length of the query and each result's score equals the length of its matched subphrase. Expand the query to facilitate matching but rank based on the length of the original subphrase (before extension). Look only at the first possible phrase match within each record. **Note:** You should only use one phrase module in any given search interface and set all of your options in it.

Results with multiple matches

If a single result has multiple subphrase matches, either within the same field or in several different fields, the result is slotted into a stratum based on the length of the longest subphrase match.

Stop words and phrase behavior

When using the `phrase` module, stop words are always treated like non-stop word terms and stratified accordingly.

For example, the query "raining cats and dogs" will result in a rank of two for a result containing "fat cats and hungry dogs" and a rank of three for a result containing "fat cats and dogs" (this example assumes subphrase is enabled).

Cross-field matches and phrase behavior

An entire phrase, or subphrase, must appear in a single field in order for it to be considered a match. (In other words, matches created by concatenating fields are not considered by the phrase module.)

Treatment of wildcards with the phrase module

The phrase module translates each wildcard in a query into a generic placeholder for a single term.

Note: Only the asterisk (*) is supported as a wildcard.

For example, the query "sparkling w* wine" becomes "sparkling * wine" during phrase relevance ranking, where "*" indicates a single term. This generic wildcard replacement causes slightly different behavior depending on whether subphrasing is enabled.

When subphrasing is not enabled, all results that match the generic version of the wildcard phrase exactly are still placed into the first stratum. It is important, however, to understand what constitutes a matching result from the phrase module's point of view.

Consider the search query "sparkling w* wine" with the `MatchAny` mode enabled. In `MatchAny` mode, search results only need to contain one of the requested terms to be valid, so a list of search results for this query could contain phrases that look like this:

```
sparkling white wine

sparkling refreshing wine

sparkling wet wine

sparkling soda

wine cooler
```

When phrase relevance ranking is applied to these search results, the phrase module looks for matches to "sparkling * wine" not "sparkling w* wine." Therefore, there are three results —"sparkling white wine," "sparkling refreshing wine," and "sparkling wet wine"—that are considered phrase matches for the purposes of ranking.

These results are placed in the first stratum. The other two results are placed in the second stratum. When subphrasing is enabled, the behavior becomes a bit more complex. Again, we have to remember that wildcards become generic placeholders and match any single term in a result. This means that any subphrase that is adjacent to a wildcard will, by definition, match at least one additional term (the wildcard). Because of this behavior, subphrases break down differently. The subphrases for "cold sparkling w* wine" break down into the following (note that w* changes to *):

```
Cold

sparkling

* wine

cold sparkling *

sparkling * wine

cold sparkling * wine
```

Notice that the subphrases "sparkling," "wine," and "cold sparkling" are not included in this list. Because these subphrases are adjacent to the wildcard, we know that the subphrases will match at least one additional term.

Therefore, these subphrases are subsumed by the "sparkling *", "* wine", and "cold sparkling *" subphrases. Like regular subphrase, stratification is based on the number of terms in the subphrase, and the wildcard placeholders are counted toward the length of the subphrase. To continue the example above, results that contain "cold" get a score of one, results that contain "sparkling *" get a score of two, and so on. Again, this is the case even if the matching result phrases are different, for example, "sparkling white" and "sparkling soda." Finally, it is important to note that, while the wildcard can be replaced by any term, a term must still exist. In other words, search results that contain the phrase "sparkling wine" are not acceptable matches for the phrase "sparkling * wine" because there is no term to substitute for the wildcard. Conversely, the phrase "sparkling cold white wine" is also not a match because each wildcard can be replaced by one, and only one, term. Even when wildcards are present, results must contain the correct number of terms, in the correct order, for them to be considered phrase matches by the phrase module.

static module

The `static` module assigns rank based on a configurable sort key.

The `static` module assigns a static or constant data-specific value to each search result, depending on the type of search operation performed and depending on optional parameters that can be passed to the module.

For record search operations, the first parameter to the module specifies a property, which will define the sort order assigned by the module. The second parameter can be specified as ascending or descending to indicate the sort order to use for the specified property.

For example, using the module

```
static(Availability,descending)
```

sorts the result records in descending order with respect to their assignments from the `Availability` property. Using the module

```
static(Title,ascending)
```

sorts the result records in ascending order by their `Title` property assignments.

In a catalog application, setting the static module by `Price`, descending leads to more expensive products being displayed first.

For dimension search, the first parameter can be specified as `nbins`, `depth`, or `rank`:

- Specifying `nbins` causes the static module to sort result dimension values by the number of associated records in the full data set.
- Specifying `depth` causes the static module to sort result dimension values by their depth in the dimension hierarchy.
- Specifying `rank` causes dimension values to be sorted by the ranks assigned to them for the application.

stem module

The `stem` module ranks matches due to stemming below other kinds of matches.

The `stem` module assigns a rank of 0 to matches from stemming, and a rank of 1 from all other sources. That is, it ignores all other sorts of query expansion.

thesaurus module

The `thesaurus` module ranks matches due to thesaurus entries below other sorts of matches. It a rank of 0 (the lowest possible priority) to matches from the thesaurus, and a rank of 1 from all other sources. That is, it ignores all other sorts of query expansion.

weighted frequency module

Like the `freq` module, the `wfreq` (weighted frequency) module scores results based on the frequency of user query terms in the result.

Additionally, the `wfreq` module weights the individual query term frequencies for each result by the information content (overall frequency in the complete data set) of each query term. Less frequent query terms (that is, terms that would result in fewer search results) are weighted more heavily than more frequently occurring terms.

Note: The `wfreq` module ignores matches due to query expansion; that is, it assigned the lowest possible priority to records included in the search results list because of such matches.

Link additional content to search results

You can obtain and display additional information, that is, non-catalog and non-indexed content, along with the search results. Examples include banner images and links to other pages.

To display additional information, you must take the following steps:

- Verify Guided Search service definition - Verify that the guided search service definition is located at:

  `/gsadmin/v1/cloud/pages/Default/services/guidedsearch`

  and ensure that it is configured to get `additionalContent` information. If not, use HTTP PUT to update the service definition.
- Configure a rule trigger - Define a content item, or trigger:

  `/gsadmin/v1/cloud/content/additionalContent/`*`TriggerName`*

  This trigger is fired when a shopper does a search for the trigger name, for example, abbreviated form. A business user can make a HTTP POST or PUT request to create or update the rule trigger definition.
- Make a search from the storefront application using search term `aboutus` to `/ccstore/v1/search` and observe the response. You should see a JSON object with key `additionalContent`.

Verify Guided Search service definition

You use REST endpoints exposed at `/gsadmin/v1/cloud/pages/Default/services/guidedsearch` to ensure that service is updated to return additional content along with search results.

The following table describes the JSON attributes required to configure the Guided Search service definition.

Attribute	Value
`additionalContent`	Key name to be used to retrieve the additional content from the search response.
`@type`	Cartridge type to be passed. This must be set as `ContentSlot`.

Attribute	Value
`contentPaths`	Resource path to rule definitions. This should point to the folder containing rules to evaluate the additional content. By default this points to `/content/additionalContent`.
`ruleLimit`	Number of rules matches will not be more than the value specified here. The rule engine will evaluate all the rules in `contentPaths` based on the priorities defined.

Example 47-1 Updated Guided Search Service definition

The following JSON illustrates the default guided search service definitions. It can be updated, if required, by using REST endpoints defined for `ecr` type page.

```
{
   "ecr:type" : "page",
   "contentItem":
   {
     "@name": "Guided Search Service",
     "@type": "GuidedSearchService",
     "additionalContent":
     {
     "ruleLimit": 1,
     "@type": "ContentSlot",
     "@contentPaths": ["/content/additionalContent"]
     }
   }
}
```

Configure a rule trigger

You use REST endpoints exposed at `/gsadmin/v1/cloud/content/additionalContent/`*`TriggerName`* to define an additional content item, or a trigger. You can have this trigger either on a search term or on a navigation state, that is, using dimension value IDs.

Example 47-2 Rule trigger definition

You can use the following call to create a new additional content item to trigger on a search for “about us”.

```
POST /gsadmin/v1/cloud/content/additionalContent/aboutus
          {
           "ecr:type" : "content-item",
           "priority" : 10,
           "contentItem":
            {
               "@type": "UnstructuredContent",
               "title": "Looking for information about our company?",
               "link-title": "About Us",
               "link-url": "/about-us",
               "image": "company-logo.png"
            },
```

```
 "triggers":[
{
   "searchTerms": "about",
   "matchmode": "MATCHEXACT"
},
   {
   "searchTerms": "about us",
   "matchmode": "MATCHEXACT"
   }
]
}
```

Verify search responses

After configuring the Guided Search service definition and the rule trigger, when a shopper performs a search or navigates to a specific configured state, it will give back `additionalContent` information along with search results.

Search non-catalog data

You can index and search data that is not part of the product catalog.

For example, you may want the Help page on your site to be included in a search. If a customer searches for “help”, the Help page is returned along with any products that match. Note the Help page will be returned in addition to the matching product catalog results.

To index and search additional data, you must take the following steps:

1. Set up the search attributes by performing the following:
   - Add attributes using Attributes API
     The properties/dimensions used for adding records to newly created record collections, are either already defined in the application or can be imported manually using `/gsadmin/v1` rest end-point.
   - Add attributes to Search interface using Search Interface API
   - Create page definition with record filter
2. Update the index by taking the following steps:
   - Upload records using the Search Data API endpoints
   - Manually Trigger Baseline/Partial update
3. Query the index by doing the following:
   Make request to new page.

Create a data record collection

You use REST endpoints exposed at `/gsdata/v1/cloud/data/{recordCollection}` to create / populate a data record collection.

```
POST /gsdata/v1/cloud/data/{recordCollection}
Content-Type: application/vnd.oracle.resource+json; type=collection - if
creating/updating bulk of records
<json data>
```

1. `recordCollection` - A valid record collection name, as defined by the regular expression
2. `[a-zA-Z][a-zA-Z0-9_-]*. CAS` has a limitation that the `recordStore` name cannot have more than 128 characters.

Example: Import records to the record collection

You can use the following call to import the specified records to the record collection.

```
POST /gsdata/v1/cloud/data/storeLocations
"Content-Type: application/vnd.oracle.resource+json; type=collection"
{
  links :[{
    "rel" : "self",
    "href" : "/gsdata/v1/cloud/data/storeLocations"
  }],
  items : [
    {
      "record.action": "deleteAll"
    },
    {
      links :[{
        "rel" : "self",
        "href" : "/gsdata/v1/cloud/data/storeLocations/store1000"
      }],
      "record.id": "store1000",
      "record.action": "upsert",
      "store.state": "CA",
      "store.amenities": [
        "Coffee Shop",
        "Pharmacy"
      ]
    },
    {
      "record.action": "delete",
      "record.id": "store1011"
    },
    ...
  ]
}
```

Use Case: Search buying guides

For the sample store, Company A has a range of cameras and camcorders for sale. To help their customers find the right camera, they have some buying guides that walk through the features, and they want to include summary information on this so the information are returned along with the products when relevant.

1. Call the `gsdata` endpoint to create the new buying guide items.
   The following are sample data representing two of these buying guides:

```
{
        "content.type": "buyingGuide",
        "guides.department": "Electrical",
        "guides.productType": "DSLR Cameras",
```

```
        "guides.keywords": "DSLR, Camera, Professional, Zoom,
interchangeable lens",
        "record.locale": "en-US",
        "record.id": "guide1001",
        "guides.description": "Looking for more information on our DSLR
cameras?  Read on to learn how to pick the right camera for you."
    },
    {
        "content.type": "buyingGuide",
        "guides.department": "Electrical",
        "guides.productType": "Compact Cameras",
        "guides.keywords": "compact, camera, holiday, small, point-and-
shoot, automatic",
        "record.locale": "en-US",
        "record.id": "guide1002",
        "guides.description": "Looking for a compact camera to take on
holiday?  We compare the best compacts!"
    }
```

First they upload the following JSON to the server using the new `gsdata` endpoint. They use POST to create a new resource, and use the syntax of `/gsdata/v1/cloud/data/buyingGuides`, where `buyingGuides` is the name for the new entry.

```
POST /gsdata/v1/cloud/data/buyingGuides
{
    "items": [{
        "content.type": "buyingGuide",
        "guides.department": "Electrical",
        "guides.productType": "DSLR Cameras",
        "guides.keywords": "DSLR, Camera, Professional, Zoom,
interchangeable lens",
        "record.locale": "en-US",
        "record.id": "guide1001",
        "guides.description": "Looking for more information on our DSLR
cameras?  Read on to learn how to pick the right camera for you."
    },
    {
        "content.type": "buyingGuide",
        "guides.department": "Electrical",
        "guides.productType": "Compact Cameras",
"guides.keywords": "compact, camera, holiday, small, point-and-shoot,
automatic",
        "record.locale": "en-US",
        "record.id": "guide1002",
        "guides.description": "Looking for a compact camera to take on
holiday?  We compare the best compacts!"
    },
    {
        "record.action": "OCCForceFlush"
    }]
}
```

2. They create new attributes.

After the data have been created, they define the attributes that were used. To do this, they use the standard Attributes API, and use POST to create new attributes corresponding to the data they uploaded previously. This is the code to create new attributes of:

- `guides.department` (as a facet)
- `guides.productType` (as a searchable property)
- `guides.description` (as a searchable property)
- `content.type` (as a filterable property)

```
POST /gsadmin/v1/cloud/attributes/system/guides.department
{
"ecr:lastModifiedBy": "admin",
"ecr:type": "dimension",
"isAutogen": true
}

POST /gsadmin/v1/cloud/attributes/system/guides.productType
{
"ecr:lastModifiedBy": "admin",
"ecr:type": "property",
"isRecordSearchEnabled": true
}

POST /gsadmin/v1/cloud/attributes/system/guides.description
{
"ecr:lastModifiedBy": "admin",
"ecr:type": "property",
"isRecordSearchEnabled": true
}


POST /gsadmin/v1/cloud/attributes/system/content.type
{
"ecr:lastModifiedBy": "admin",
"ecr:type": "property",
"isRecordFilterable": true
}
```

3. They create new services using the pages endpoint, and add the new fields as being returned.
   They want to define a separate endpoint that they can call to return this information. This is done using the Pages Admin API. They name the page "`buyingGuide`", and define that they want to return four properties:

   ```
   guides.description

   guides.productType

   guides.department

   record.id

   POST /gsadmin/v1/cloud/pages/Default/buyingGuide
   {
   ```

```
  "contentType": "Page",
  "ecr:type": "page",
  "contentItem": {
    "@name": "Content Search Service",
    "@type": "GuidedSearchService",
    "@appFilterState": {
      "@type": "FilterState",
      "recordFilters": [
        "content.type:buyingGuide"
      ]
    },
    "navigation": {
      "@type": "NavigationContainer",
      "contentPaths": [
        "/content/facets"
      ]
    },
    "resultsList": {
      "@type": "ResultsList",
      "fieldNames": [
                "guides.description",
                "guides.productType",
                "guides.department",
                "record.id"
            ],
      "rankingRules": {
        "merchRulePaths": [
          "/content/rankingRules"
        ],
        "systemRulePaths": [
          "/content/system/rankingRules"
        ],
        "systemRuleLimit": 10
      }
    },
    "searchAdjustments": {
      "@type": "SearchAdjustments"
    }
  }
}
```

4. Add the properties created above to the “All” search interface.
5. Add the searchable attributes they have just added to the “All” searchable field ranking. To do this:
   a. Open the Oracle Commerce administration interface.
   b. Click to open the Search section.
   c. Click **Searchable Field Rankings**.
   d. Click **All**.
   e. Use the dropdown to add the new attributes to the end of the list, in this order:
      `guides.department`

      `guides.productType`

```
guides.description
```

6. Index the data. They use the following API endpoint to run a search index to update the data:

   ```
   POST /ccadmin/v1/search/index?op=baseline
   ```

   They monitor the progress of this using:

   ```
   GET /ccadmin/v1/search
   ```

   When `success: true` is displayed, the indexing has completed.

7. Verify that this works correctly. Query the data by calling the following URL on their storefront:

   ```
   /ccstore/v1/assembler/pages/Default/services/buyingGuide?Ntt=camera
   ```

Create a widget to support searching data

Custom product listing widget can support searching data that is not part of the product catalog. For detailed information about creating widgets, see Create a Widget.

Create the widget structure for the product listing sample widget

Widgets that include user interface elements must include display templates. The following shows an example of the files and directories in a product listing widget.

```
Widget/
    ext.json
    widget/
        ProductistingForAdditionalContent
            widget.json
            js/
                product-listing.js
            less/
                widget.less
            locales/
            en/
                ns.multicart.json
            templates/
                display.template
                paginationControls.template
```

The JavaScript code you write extends the `createsearchViewModel` class. For more information about the widget structure and the contents of the `ext.json` and `widget.json` files, see Understand widgets .

Create the JavaScript file for the product listing sample widget

The widget's JavaScript file includes functions that let shoppers search data that is not part of the product catalog.

The following example shows sample JavaScript that implements the `createsearch` functionality:

```
////Non-catalog content
        widget.assemblerPagesPath = "services/storeLocations"; //This should
be services/searchService
        widget.ntk = "stores"; //This should point to the search interface
created
        //Created a function to process the non-catalog content to display
them
        widget.amenitiesToText = function(obj){
          var amenitiestext = obj[0];
          for(var i=1; i < obj.length; i++ ){
                amenitiestext = amenitiestext + ", " +obj[i];
              }
          return amenitiestext;
        };
///////End non-catalog content        }
```

Create template files for the product listing sample widget

The widget's `display.template` file contains the following code for rendering the page (the following is an excerpt of the `display.template`):

```
<!-- ko if: (listingViewModel().display) -->
<div id="CC-productListing" role="alert">
  <!--  ko if: listType() == 'search' -->
  <!--  ko with: listingViewModel -->
  <div class="sr-only" data-bind="text :pageLoadedText"></div>
  <!-- /ko -->
  <!-- /ko -->
  <div id="CC-product-listing-controls" class="row">
    <div class="col-sm-12">
      <!-- ko with: listingViewModel -->
      <!-- ko if: $parent.listType() == 'search' -->
      <h2 id="search-results" class="sr-only" role="alert" data-
bind="widgetLocaleText: 'searchResultsText'"></h2>
      <!-- /ko -->
      <!-- ko if: titleText -->
      <div class="row">
        <div class="col-xs-12">
          <h2 id="cc-product-listing-title" data-bind="text: titleText"></h2>
        </div>
      </div>
      <!-- /ko -->
      <!-- ko if: $parent.listType() == 'search' -->
      <!-- ko if: noSearchResultsText -->
      <div class="row">
        <div id="cc-productlisting-noSearchResults" class="col-xs-12">
          <span data-bind="text: noSearchResultsText"></span>
        </div>
      </div>
      <!-- /ko -->
      <!-- ko if: suggestedSearches().length > 0 -->
```

```
      <div id="cc-productlisting-didYouMean">
        <span data-bind="widgetLocaleText:'didYouMeanText'"></span>
        <div id="cc-productlisting-didYouMeanTerms" data-
bind="foreach : suggestedSearches">
          <a data-bind="attr: {id: 'cc-productlisting-didYouMean-
Suggestion-'+$index()}, widgetLocaleText: {value:'dYMTermAriaLabel',
attr:'aria-label'}, click: $data.clickSuggestion" href="#">
            <!-- ko if: ( $index() <
($parent.suggestedSearches().length - 1)) -->
            <span data-bind="widgetLocaleText :
{value:'dYMTermTextHasNext', attr:'innerText', params:
{label: $data.label}}"></span>
            <!-- /ko -->
            <!-- ko if: ( $index() ==
($parent.suggestedSearches().length - 1)) -->
            <span data-bind="widgetLocaleText : {value:'dYMTermText',
attr:'innerText', params: {label: $data.label}}"></span>
            <!-- /ko -->
          </a>
        </div>
      </div>
      <!-- /ko -->
Pro      <!-- /ko -->
      <!-- /ko -->
      <!-- ko if: (listingViewModel().totalNumber() > 0) -->
      <div data-bind="text: resultsText" class="sr-only"
role="alert"></div>
      <!-- ko if: listType() == 'search' -->
      <h3 class="sr-only" role="alert" data-bind="widgetLocaleText:
'viewingOptionsText'"></h3>
      <!-- /ko -->
      <div class="row">
        <div class="col-sm-12" id="cc-area-controls">
          <div class="row">
```

The widget's `display.template` calls another template file, `paginationControls.template`. This template file contains the following code for rendering multiple pages when the list of products is long:

```
<div class="btn-group">
    <a class="btn btn-default" data-bind="ccNavigation: '',  attr :
{href: firstPage()}, widgetLocaleText : {value:'goToFirstPageText',
attr:'aria-label'}, css: { disabled: $data.currentPage() == 1 }"
><span data-bind="widgetLocaleText: 'goToFirstPagePaginationSymbol'"></
span></a>
    <a href="#" class="btn btn-default" data-bind="ccNavigation: '',
attr: {href: previousPage()}, widgetLocaleText :
{value:'goToPreviousPageText', attr:'aria-label'}, css:
{ disabled: $data.currentPage() == 1 }" rel="prev"><span data-
bind="widgetLocaleText: 'goToPreviousPagePaginationSymbol'"></span></a>

    <!-- ko foreach: pages -->
      <a href="#" class="btn btn-default" data-bind="ccNavigation: '',
attr: {href: $parent.goToPage($data)}, css:
```

```
{active: $data.pageNumber===$parent.currentPage() }">
      <!-- ko if: $data.selected === true -->
          <span class="sr-only" data-bind="widgetLocaleText :
'activePageText'"></span>
        <!-- /ko -->
        <!-- ko if: $data.selected === false -->
          <span class="sr-only" data-bind="widgetLocaleText :
'goToPageText'"></span>
        <!-- /ko -->
        <span data-bind="ccNumber: $data.pageNumber"></span>
       </a>
      <!-- /ko -->

    <a href="#" class="btn btn-default" data-bind="ccNavigation: '', attr:
{href: nextPage()}, widgetLocaleText : {value:'goToNextPageText', attr:'aria-
label'}, css: { disabled: currentPage() == $data.totalNumberOfPages() }"
rel="next"><span data-bind="widgetLocaleText:
'goToNextPagePaginationSymbol'"></span></a>
    <a href="#" class="btn btn-default" data-bind="ccNavigation: '', attr:
{href: lastPage()}, widgetLocaleText : {value:'goToLastPageText', attr:'aria-
label'}, css: { disabled: currentPage()
== $data.totalNumberOfPages() }"><span data-bind="widgetLocaleText:
'goToLastPagePaginationSymbol'"></span></a>

</div>
```

Machine learning for search

Commerce can identify popular products associated with popular searches based on a number of factors, including shopper behavior and purchasing history, trend analysis, and view data, and automatically generate a corresponding set of ranking rules that boost these products in the related search results.

The machine-learning feature works in conjunction with dynamic curation and search relevancy. Manually-boosted products take precedence, followed by products identified by machine learning, with all remaining products ordered by relevancy and dynamic-curation criteria.

This feature affects only keyword search results and any subsequent navigation. It does not affect pages that contain only navigation results.

Understand auto-generated ranking rules

When the feature is enabled, Commerce periodically uploads the auto-generated popular products ranking rules to the `/system/rankingRules` endpoint under `internal-keywords`. You can view the auto-generated ranking rules by querying the following endpoint:

```
GET /gsadmin/v1/cloud/content/system/rankingRules/internal-keywords
```

Important: Do not attempt to modify the internal-keywords auto-generated ranking rules. Commerce will overwrite your changes.

The auto-generated popular products ranking rules take effect in your production environment whenever changes are published.

Only Boost/bury rules have higher precedence than the auto-generated popular products ranking rules. Popular products ranking rules take precedence over the default Relevancy ranking, Dynamic Curation, and static ranking rules.

Enable auto-generated popular products ranking rules

To enable this feature, you must use the Admin API:

1. Run `GET /gsadmin/v1/cloud/configuration/services/internal-keywords`.
2. If `boostDisabled` is present in the response and set to `true`, then set it to `false`.
3. Issue a `PUT` command to `/gsadmin/v1/cloud/configuration/services/internal-keywords` to save your modifications.

Changes will take affect once the overnight process has run.

Disable auto-generated popular products ranking rules

To disable this feature, you must use the Admin API:

1. Run `GET /gsadmin/v1/cloud/configuration/services/internal-keywords`.
2. If `boostDisabled` is present in the response and set to `false`, then add it if necessary and set it to `true`.
3. Issue a `PUT` command to `/gsadmin/v1/cloud/configuration/services/internal-keywords` to save your modifications.
4. Issue a `DELETE` command to `/gsadmin/v1/cloud/content/system/rankingRules/internal-keywords` to delete previous rules.

Changes will take affect once the overnight process has run.

Identify Promoted Products

To identify when products have been promoted, you can access your site using the Preview environment and inspect the underlying `ccstore/v1/search` endpoint call. Each promoted product includes an additional property of `DGraph.RankLabel.bstratify.merch`.

Sample Search and Navigation REST API requests using cURL

This section provides several examples of how common Search and Navigation REST API requests can be made through the `cURL` command-line utility.

For general information about how to make requests, see Understand how to execute endpoints.

Note: Oracle does not recommend the use of any particular UI tool or utility for making REST API calls.

Log in and get an access token

Use the following POST request to log in to the Admin API on the Oracle Commerce administration server, using an account that has the Administrator role:

```
curl -X POST
-- data "grant_type=password&username=user_name&password=password&totp_code=
  passcode" http://host:port/ccadmin/v1/mfalogin > /pathname/filename.txt
```

Note that the request must include a user name, password, and passcode. To obtain passcodes, the login account must be registered with the Oracle Mobile Authenticator app. See Access the Commerce administration interface for more information.

This POST request returns an `OAuth` access token, which you must provide in the authorization headers of all other requests. The following is an example of an access token, greatly abbreviated:

```
"token_type":"bearer","access_token":"Authorization:Bearer
  eyJhbGciOiJSUzI1NiIsImprdSI6IjVkYmQyTA4MC9jY2FkbWluL=="
```

where `eyJhbGciOiJSUzI1NiIsImprdSI6IjVkYmQyTA4MC9jY2FkbWluL==` is the `OAuth` access token (abbreviated) that you must use in the authorization headers of other requests.

Note: Because the `OAuth` access token returned by this request must be used by all other requests, it is convenient to redirect it to a text file, as shown in the example above. The access token can then be copied from the text file into the other requests.

Export configuration of a search resource

Issue a GET command to export configuration of a search resource. Note that GET is the default command and need not be specified.

For example, the following request exports the configuration of the search interface named ALL. The exported JSON content is displayed on the screen, immediately beneath the command line:

```
curl [-X GET] -H "Authorization:Bearer access_token"
http://host:port/gsadmin/v1/searchInterfaces/All.json
{
    "ecr:lastModifiedBy": "admin",
    "ecr:lastModified": "2016-10-17T05:25:35.764-07:00",
    "ecr:createDate": "2016-10-17T05:25:35.764-07:00",
    "ecr:type": "search-interface",
    "crossFieldMatch": "always",
    "fields": [
      {"attribute": "product.displayName"},
      {"attribute": "sku.displayName"},
      {"attribute": "parentCategory.displayName"},
      {"attribute": "product.brand"},
      {"attribute": "parentCategory.keywords"},
      {"attribute": "sku.description"},
      {"attribute": "product.description"},
      {"attribute": "product.category"},
      {"attribute": "ecr:crossField"},
      {"attribute": "product.longDescription"},
```

```
    {"attribute": "product.repositoryId"}
  ]
```

The following request redirects the exported configuration of the search interface `All` to a file:

```
curl [-X GET] -H "Authorization:Bearer access_token"
http://host:port/gsadmin/v1/searchInterfaces/All.json > /pathname/
filename.txt
```

The following request exports the configuration of the search interface `All` in a ZIP file:

```
curl [-X GET] -H "Authorization:Bearer access_token"
http://host:port/gsadmin/v1/searchInterfaces/All.zip -o pathname/
filename.zip
```

where:

`-o (lower case) pathname/filename.zip` causes the ZIP file specified by *filename* to be downloaded to the location specified by *pathname*.

Note: ZIP format exports not only the configuration of the specified object to be exported, but also the configuration of all its child objects, in separate `_.json` files. JSON format, in contrast, exports only the configuration of the specified object to be exported.

Create a search resource

In JSON format, use the following POST request to configure the `searchInterfaces` folder. The configuration to be imported is provided in a JSON file:

```
curl -X POST -H "Authorization:Bearer access_token"
-d@pathname/filename.json
-H "Content-Type:application/json"
http://host:port/gsadmin/v1/searchInterfaces
```

In ZIP format, use the following POST request to configure the `searchInterfaces` folder. The configuration to be imported is provided in a ZIP file. The ZIP file must contain a file named `_.json`:

```
curl -X POST -H "Authorization:Bearer access_token"
-F ":file=@pathname/filename.zip"
http://host:port/gsadmin/v1/searchInterfaces
```

Note: When the POST request is executed in JSON format, the POST request can create a resource but not overwrite an existing resource. When the POST request is executed in ZIP format, the POST request can either create the resource or overwrite it if it already exists.

Modify a search resource

Use the PATCH method to add attributes to an object or change the values of existing attributes in the object. The PATCH method can be used only in JSON format.

For example, the following PATCH request can modify the values of existing attributes of a `thesaurus-entry` object, or add new attributes to that object:

```
curl -X PATCH -H "Authorization:Bearer access_token"
-d@pathname/filename.json
-H "Content-Type:application/json"
http://host:port/gsadmin/v1/thesaurus/my-thesaurus-entry
```

Replace configuration of a search resource

In JSON format, execute the following command to replace existing configuration of the search interface `All`; in this example, the configuration containing the replacement is provided in a JSON file named `SearchInterface_put.json`:

```
curl -X PUT -H "Authorization:Bearer access_token"
-d@pathname/SearchInterface_put.json
-H "Content-Type:application/json"
http://host:port/gsadmin/v1/cloud/searchInterfaces/All
```

In ZIP format, execute the following command to replace existing configuration of the search interface `All`; in this example, the configuration containing the replacement is provided in a ZIP file named `temp.zip`:

```
curl -X POST -H "Authorization:Bearer access_token"
-F ":file=@temp.zip"
http://host:port/gsadmin/v1/cloud/searchInterfaces/All
```

Install cURL

The examples within this chapter use the `cURL` command-line tool to demonstrate how to access the Commerce Search and Navigation REST API. To connect securely to a server, you must install a version of cURL that supports SSL.

The following procedure demonstrates how to install `cURL` on a Windows 64–bit system.

1. In your browser, navigate to the `cURL` home page at `http://curl.haxx.se` and click **Download** in the left navigation menu.
2. On the `cURL` Releases and Downloads page, locate the SSL-enabled version of the `cURL` software that corresponds to your operating system, click the link to download the ZIP file, and install the software.

You are now ready to send requests to the Commerce Search and Navigation REST API using `cURL`.

This table summarizes the `cURL` options used in the command examples.

cURL Option	Description
`-d, --data @filename.json`	Identifies the request document, in JSON format, on the local machine.
`-H`	Defines one or both of the following: • - Content type of the request document • - Custom header to pass to server

cURL Option	Description
`-o, --output filename`	Writes output to a file instead of `stdout`
`-X`	Indicates the type of request (for example, GET, POST, and so on).

For example:

```
curl -X GET -H <request-header>:<value>
https://<subdomain>.<domain>.com/<path>/resource-path>
```

48

Use Developer Utilities

Oracle Commerce includes a number of utilities you can use during your development process.

This chapter provides information on working with development tools including the Cloud Commerce SDK, the Design Code Utility, and the JavaScript Code Layering User Interface.

Download the Commerce SDK

The Commerce SDK helps you create server-side integrations with Commerce.

The Commerce SDK includes a REST client designed to connect to an Oracle Commerce server from a `Node.js` application. The REST client supports the Commerce Store API and Admin API, as well as APIs for external systems you want to integrate with Commerce.

To download the Commerce SDK, do the following:

1. Click the **Settings** icon menu, click **Web APIs**, then click the **Commerce SDK** tab.
2. Click the **download** link.
3. Specify whether to open the ZIP file or save it.

The ZIP file contains the following files that you can read to learn about the SDK:

- `README.md` describes how to configure the REST client.
- JSDoc descriptions of the SDK's classes and global variables. To access the JSDoc files, open the file `/oracle-commerce-sdk/docs/index.html` in the location where you saved the Commerce SDK ZIP file.

Use the Design Code Utility

Oracle Commerce includes the Design Code Utility, a suite of command-line tools that allows you to integrate Commerce with the IDE or code editor of your choice.

The Design Code Utility suite is comprised of the following:

- `dcu` – The core code and metadata synchronization utility.
- `ccproxy` – The development and debugging proxy.
- `ccw` – The Content Creation Wizard.
- `plsu` - The Page Layout Synchronization Utility, which enables you to manage and edit page layouts.

Download and install the Design Code Utility suite

Before you can use it, you must download the Design Code Utility suite and install it using Node Package Manager.

To download and install the Design Code Utility suite:

1. Log in to the Commerce administration interface.
2. Click the **Design** icon.
3. Under **Developer**, click **Developer Tools**.
4. Under **Design Code Utility**, click the **Download** button.
5. Save the ZIP file to a location on your local machine and extract its contents.
6. Change directories to the extracted location and run the following command to install the Design Code Utility script:
   `npm install -g`

The install process makes the Design Code Utility and other utility scripts available through the `dcu.cmd|sh` command. At this point, you can either run the `dcu` command via a command prompt or by configuring the integrated development environment (IDE) of your choice to run it.

On Linux systems, performing the following command will put the command line utilities on your PATH:

```
sudo npm link
```

Use the dcu to grab and upload source code and metadata

The `dcu` is the core code and metadata synchronization utility, which allows you to grab all of the user modifiable source code and metadata from a Commerce server, edit it as necessary in your chosen tool, and then upload it back to the server. The utilities also makes it possible to transfer content between Commerce server instances. The dcu supports these main operations:

- `--grab` takes a copy of all, or some of, the available user modifiable source code and metadata from the specified Commerce server and creates a directory tree on the local disk.
- `--put` sends the contents of a single file back to the specified Commerce server.
- `--putAll` sends all the files underneath a named directory back to the specified Commerce server.
- `--transferAll` enables the transfer of content between Commerce server instances.
- `--compileLess` combines all off the downloaded less resources for widgets, elements, stacks and themes and posts them to the node for compilation.
- `--clean` deletes all local files that have been previously created.

The basic command for grabbing files is:

```
dcu --grab
```

The basic command for uploading a single file is:

```
dcu --put <path to file>
```

For example:

```
dcu --put "widget/Add Product To Wish List/instances/
      Add Product To Wish List Widget/display.template"
```

The basic command for uploading all files under a specified directory is:

```
dcu --putAll <directory>
```

For example:

```
dcu --putAll theme
```

During each of these operations, the utility provides messages to tell you the status of the grab or upload. You can modify these basic commands with the options described in the Understand dcu options section.

Refresh previously grabbed content

When developing widgets you may find it necessary to refresh your local directory tree with the latest content from the server. While this can be done by issuing another grab, this may take a few minutes on a larger site and can be wasteful if you are only interested in specific changes. You can refresh all of the files within a specified directory by using the `--refresh` command. For example, the following would only refresh all of the files from the widget directory:

```
dcu –refresh widget
```

You can run incremental refreshes on the following directories:

- The application-level JavaScript directory, for example `global`
- The global snippets directory, for example `snippets`
- Individual global snippets locale directories, for example, `snippets/en`
- The stack directory, for example `stack`
- Individual stacks directories, for example `stack/MyStack`
- The themes directory, for example, `themes`
- Individual themes directories, for example `theme/BlueTheme`
- The global elements directory, for example `element`
- Individual global element directories, for example `element/image`
- The widgets directory, for example `widgets`
- Individual widget directories, for example `widgets/Header`
- The site settings directory.

For example, to refresh a specific stack from within the stack directory, you could perform the following command:

```
dcu --refresh stack/MyStack
```

Generate a partial grab

For certain workflows, for example when DCU is being used as part of a continuous integration pipeline, it was be advantageous to grab only objects of a certain type. This is not the same as a refresh however as refresh requires there to have been at least one grab in the target directory. You can do a partial grab by supplying a directory with the –grab option

For example, the following would only grab only the element related content from server: dcu –--grab element

You can run incremental grabs on the following directories:

- The application-level JavaScript directory, for example, `global`
- The global snippets directory, for example, `snippets`
- Individual global snippets locale directories, for example, `snippets/en`
- The stack directory, for example, `stack`
- Individual stacks directories, for example, `stack/MyStack`
- The themes directory, for example, `themes`
- Individual themes directories, for example, `theme/BlueTheme`
- The global elements directory, for example, `element`
- Individual global element directories, for example, `element/image`
- The widgets directory, for example, `widgets`
- Individual widget directories, for example, `widgets/Header`
- The site settings directory

For example, to grab only a single element you could perform the following command:

```
dcu --refresh element/MyElement
```

Transfer source code and metadata between Commerce instances

You can use the `dcu` to transfer source code from one Commerce instance to another. This can be useful for promoting source code from a test environment to production.

To use the transfer feature, you must execute a grab from the source instance, for example:

```
dcu  --grab --clean --node https://sourceInstance --user username --
password pwd
```

After the grab is finished, you transfer the files to the destination instance, using a command similar to the following:

```
dcu --transferAll path --node https://destinationInstance --user
username --password pwd
```

The `dcu` takes a “best effort” approach to transferring content between instances. In general, it matches by name so that if, for example, a widget instance on the destination instance has the same name as a widget instance on the source instance, they are assumed to be the same. The matching rules for each entity type are described in the table below.

Type	Matching rules
Global elements	Matches on the `<element>` tag. If the element does not exist, a warning is issued.

Type	Matching rules
Application-level JavaScript modules	Matches on the JavaScript file name. If the file name does not exist, it is created on the destination instance.
Text snippets	Matches on locale.
Stacks	Matches on the stack instance's display name. If the stack instance does not exist, a warning is issued.
Themes	Matches on the theme name. If a matching theme does not exist, one is created on the target instance, using the same name and source code.
Widgets	Widgets are matched on version number and display name. If the widget does not exist, a warning is issued. Widget instances are matched on display name. If a matching widget instance does not exist, a new widget instance is created on the target instance using the same display name and source code.
Elements	Widget elements are matched on version number, display name and `<element>` tag. If the element does not exist, a warning is issued.
Site Settings	Site Settings are matched on display name. If a matching Site Settings group does not exist on the target server, one is created with the same name and source code.

Understand dcu options

The following table describes the options you can use with the `dcu`.

Option	Description
`-h, --help`	Provides usage information for the utility.
`-V, --version`	Provides the utility's version number.
`-n, --node <node>`	The URL for the administration interface on the target Commerce instance, for example, `https://localhost:9080`. Used with `--grab`, `--put`, and `--putAll`. If `--node` is not specified, the utility attempts to use the most recently specified node.
`-k, --applicationKey <key>`	The application key to use to log into the target Commerce administration interface. It is recommended that you create an application key for authentication purposes. For detailed information on creating an application key, refer to the Use the application key for authentication section. It is also possible to specify the application key using the CC_APPLICATION_KEY environment variable.

Option	Description
-u, --username <userName> -p, --password <password>	These options are no longer available, having been replaced by the applicationKey option.
-g, --grab [<directory>]	If no directory is specified, the grab option copies all available content from the target Commerce instance into the current working directory, or the base directory if one has been specified. If a directory is specified, then only content associated with that directory will be grabbed. **Note:** During a grab, a new directory called .ccc is created on disk alongside the grabbed directories. This is called the Tracking Directory and holds important metadata used by the utility. Do not modify this directory.
-t, --put <path to file>	Sends the specified file back to the specified Commerce instance. The <path to file> can be either a relative or absolute path, in either Windows or POSIX format. Relative paths are relative to the base directory, which you can specify using the --base option. If the --base option is not provided, the base directory is assumed to be the current working directory. If a full path is provided, then any value specified in the --base option is ignored.
-b, --base <directory>	Specifies the base directory. This can be either a relative or absolute path, in either Windows or POSIX format.
-l, --locale <locale>	Retrieves text snippets and resources for the specified locale only. Also, the informational messages displayed by the Design Code Utility will appear in the locale specified.
-a, --allLocales	Gets text data for all locales rather than just the current locale for the target Commerce instance.
-e, --refresh <directory>	Refreshes content from the Commerce instance within the specified directory.
-c, --clean	Deletes all local files that have been previously grabbed. Used only with the --grab option.
-m, --putAll <directory>	Sends all files back to the target Commerce instance, beneath the specified directory. The <directory> can be either a relative or absolute path, in either Windows or POSIX format. Relative paths are relative to the base directory, which you can specify using the --base option. If the --base option is not provided, the base directory is assumed to be the current working directory. If a full path is provided, then any value specified in the --base option is ignored.

Option	Description
-x, --transferAll <path>	Transfers all files from the local machine to the specified Commerce instance, matching by name. For example, if a widget instance is called Red Footer and the utility finds a widget instance on the target that is also called Red Footer, the utility assumes they are the same.
-i, --updateInstances	Updates all instance templates, less and locale strings using their respective base assets. By default, when a base widget template, less or locale file is modified, the changes affect only widget instances that are subsequently created. When you select this option, all widget instance assets of a specific type are overwritten with their corresponding base contents. You can use this option to correct issues affecting a large number of existing instances. It is recommended to use this option sparingly.
-o, --noInstanceConfigUpdate	By default, when performing a put, putAll or tranferAll, widget instance metadata is updated on the Commerce instance. However, you may want to have widget instance configuration metadata different between Commerce instances. Use this option to suppress widget instance metadata updates.
-C, --compileLess [auto]	This option, used in conjunction with ccproxy, prepares a local copy of the rendered storefronts CSS file. The CCAdminUI endpoint ensures that all standard storefront styles are included, as well as any less files you may have created. If the active site theme is restricted, the Less Compiler also includes the current active theme in the CSS output.
-N, --noThemeCompile	When DCU puts or transfers a Less file, it will trigger a Theme recompilation. However, for large scale transfers, this can significantly increase transfer time. Therefore, specifying this option means that a Theme recompilation is not automatically triggered and, as such, you are required to manually trigger it once the transfer completes. See Manage Large Scale Transfers for further details.
-A, --autofix	This option is recommended for use on large scale transfer operations, and helps to avoid issues that may arise when transferring instances of Oracle widgets. See Manage Large Scale Transfers for further details.
-d, -delete	The delete option enables the deletion of widget instances.

Add JavaScript to existing user-created widgets

You can add JavaScript to an existing user-created widget by creating a new .js file under the widget's /js directory and issuing a put, putAll or transferAll.

Use the ccproxy utility

The `ccproxy` utility is the proxy that allows you to develop and debug your code. The script allows you to route your storefront requests through a local `node.js` application, which intercepts certain resources and then substitutes them with a local modified version. When the server starts up, it looks at your widget, element, stack and theme metadata and then builds mappings of the server data to local files to the local files downloaded by the `dcu` utility.

Note: Before you can use ccproxy, you must redirect your browser HTTP traffic to the port on which ccproxy is listening. Refer to your browser documentation for information on browser-specific proxy switchers.

Work with HTTPS traffic

When using a proxy to connect to a secure HTTPS site, ccproxy automatically generates self-signed certificates. These certificates may cause a warning in the browser the first time you visit the proxied storefront. However, it is safe to proceed past this warning.

Note: The browser proxy switcher that you use must also be set up to redirect HTTP and HTTPS traffic to the `ccproxy` host.

The following table describes the options you can use with the `ccproxy` utility.

Option	Description
`-h, --help`	Provides usage information for the utility.
`-V, --version`	Provides the utility's version number.
`-n, --node <node>`	The URL for the administration interface on the target Commerce instance, for example, `https://localhost:9080`. Used with `--grab`, `--put`, and `--putAll`. If `--node` is not specified, the utility attempts to use the most recently specified node.
`-P, --port <number>`	Changes the port on which the ccproxy listens. The default is 8088. Alternatively, you can set the `CC_DEVPROXY_PORT` environment variable.
`-b, --base <directory>`	Uses the directory indicated as the base directory.
`-l, --list`	List proxy substitution rules.
`-d, --disable <list>`	Specify a set of substitution rules to disable (overrides `-e` flag).
`-e, --enable <list>`	Specify a set of substitution rules to enable. **Note:** Substitution rules can be enabled/disabled by name (you can view the rule names using the -l flag). The -e/-d parameters will take one or more rules names as a comma separated list. e.g. `$ ccproxy -e Javascript,WidgetTemplate` To enable only substitution of JavaScript files and widget HTML templates.

Use the ccw utility

Commerce includes the ccw utility, a command-line tool that allows you to create new content on Commerce with the IDE or code editor of your choice. This tool allows you to create new widgets, stacks and elements, and then upload them to the server.

For detailed information on working with widgets, stacks and elements, refer to Understand widgets.

The format for invoking the `ccw` is:

```
ccw [options]
```

Understand ccw utility options

The following table describes the options you can use with the `ccw` utility.

Option	Description
`-b, --base <directory>`	Identifies the base directory.
`-h, --help`	Provides usage information for the utility.
`-l, --locale <locale>`	Uses the supplied locale as the default locale. Also, the informational messages displayed by the `ccw` will appear in the locale specified.
`-n, --node <node>`	The URL for the administration interface on the target Commerce instance, for example, `https://localhost:9080`. If `--node` is not specified, the utility attempts to use the most recently specified node.
`-k, --applicationKey <key>`	The application key to use to log into the target Commerce administration interface. It is recommended that you create an application key for authentication purposes. For detailed information on creating an application key, refer to the Use the application key for authentication section. It is also possible to specify the application key using the `CC_APPLICATION_KEY` environment variable.
`-V, --version`	Provides the utility's version number.
`-w, --createWidget`	Creates a new widget.
`-e, --createElement <widgetOrElementDirectory>`	Create a new element - optionally under a widget directory or as child of another element.
`-g, --generateMarkup [elementDirectory]`	Generate template mark-up to match the supplied element including sub-elements.
`-s, --createStack`	Create a new stack.
`-t,, --createSiteSettings`	Create a new group of site settings.

Create a widget with the utility

When you create a new widget, the `ccw` asks a number of questions so that it may create a widget. The extension is created by calling the appropriate endpoints.

Initiate the `ccw` with the option to create a new widget:

```
ccw -w
```

This brings up the following wizard interface:

Understand the ccw utility widget creation questions

When creating the widget, you are prompted to answer a number of questions. The way that you answer these questions affects the way that the widget is created. The following section provides details of the questions asked during the creation of the widget.

Question	Explanation
Enter the name of the new widget	The textual name used to create the widget directory. It is also the name used when the widget is created on the Commerce instance.
Is the new widget global? (Global widgets are loaded on every page but have NO display template.)	Global widgets are a way to include common JavaScript code across every page layout in the store. A global widget can have only a single instance. Note that a global widget is not the same as a global `ViewModel`. Should you create observable properties in your global widget, they will not be automatically available in any `ViewModel`.
Does the new widget require internationalization?	Internationalized widgets require additional resource bundles, which are created if you answer yes.
Is the new widget configurable?	If you answer yes , you are indicating that the widget requires user configurable meta data. Refer to Understand widgets for information on configuring widgets.
Should the new widget contain example source code?	If you respond yes, the generated widget will contain various instructions and examples to assist those new to widget development.
Should the new widget be sent to the instance immediately?	If you respond yes, the new widget will be created as an extension on the target Commerce instance. If you answer no, the widget is created the next time you do a `putAll` or `transferAll`.
Does the new widget require fragmentation?	If you answer yes, the widget will be usable with elements.

After you have answered all of the questions, the responses are validated and a skeleton widget is created. Once the widget has been created, it is sent to the server as an extension, unless you requested that it should not. Once you get the response that the upload is successful, you can drag the widget onto a layout.

Create a simple global element with the ccw utility

When you create a new element, the `ccw` asks a number of questions to determine the kind of element needed. The new global element will be usable with all widget types.

Initiate the `ccw` with the `-e` option to create a global element:

```
ccw -e
```

This will bring up the following wizard interface:

Create a simple widget element with the ccw utility

New elements can also be created under widgets, provided that the widget has not already been placed on the Commerce instance. The resultant element will only be usable with that widget.

Initiate the `ccw` with the `-e` option to create a widget element and include a path to the widget the element is to be added to:

```
ccw -e widget/Galaxy
```

This will bring up the same wizard interface used for global elements.

Create a rich element with the ccw utility

Elements can also be created as children of other elements. This enables the creation of configurable rich elements similar to “Price” or “Inventory Status” in the “Product Details” widget.

Creating a rich element follows a standard pattern best thought of as a tree:

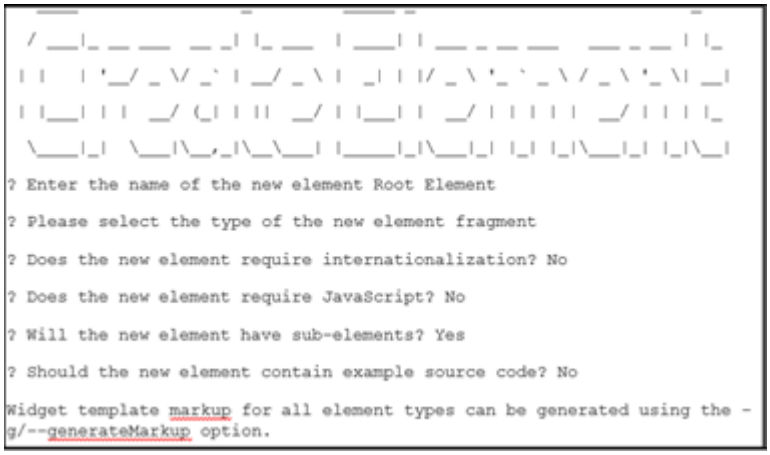
- At the top level (or root node) must be an element of type “fragment” with the single `configOption` “available” selected.
- At middle level, there must be one or more elements of type “container”. These must be children of the previously described top level element. Only the `configOptions` “available”, “actual”, “currentConfig” and “preview” should be used with container elements.
- At the lowest level (leaf nodes) must be one or more elements of type “subFragment”, “staticFragment” or “DynamicFragment”. These must be children of a container element. Only the configOptions ”border”, “collectionPicker”, “elementName”, “fontPicker”, “horizontalAlignment”, “image”, “padding”, “preview”, “richText”, “textBox”, and “wrapperTag” can be used with elements of this type.

Also, the markup for a rich element needs to be manually placed in a widget template - it cannot merely be dragged to a widget within the **Design** page. The markup for a rich element has the following general form:

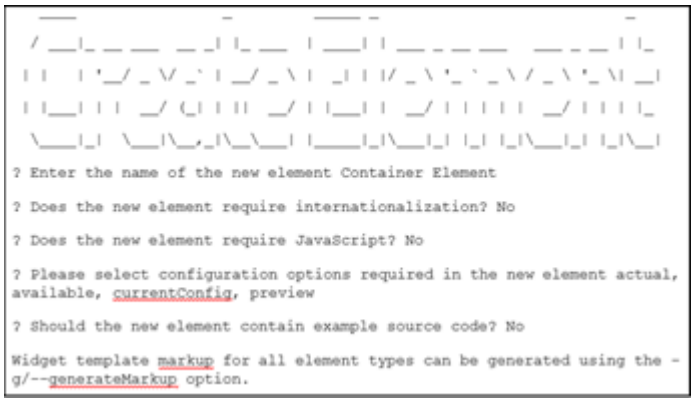
```
<!-- oc section: root-element-tag -->

<div data-oc-id="container-element-tag">

  <span data-bind="element: 'leaf-element-tag'"></span>

</div>

<!-- /oc -->
```

Creating a rich element is a multi-stage process and `ccw` tries to make this as smooth as possible. To create a rich element with `ccw`:

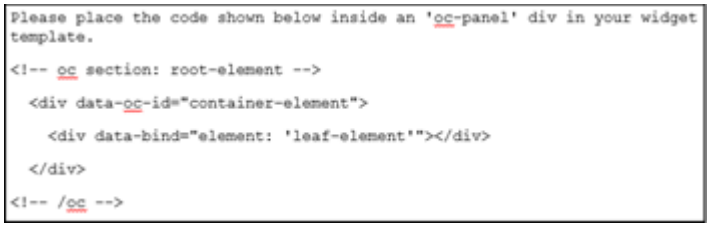
1. Create the top level element by selecting a type of “fragment” and answering yes to the sub-elements question:



2. Create one or more child container elements by specifying the path to the previously created root element on the `ccw` command line:



3. Create one or more child leaf elements by specifying the path to the previously created container element on the `ccw` command line:



If you do a `putAll/transferAll` on a directory containing multiple elements, DCU will send the leaf elements first, then the containers, and finally the top level elements.

Note that if a user creates an elementized widget, or an element containing sub-elements using `ccw` and answers yes to the “example source code” question, a sample rich element will be created.

Understand the ccw utility element creation questions

When creating the element, you are prompted to answer a number of questions. The way that you answer these questions affects the way that the element is created. The following section provides details of the questions asked during the creation of the element.

Question	Explanation
`Enter the name of the new element`	The textual name used to create the element directory. It is also the name used when the element is created on the Commerce instance.
`Please select the type of the new element`	The type of the element you require. Depending on the context, the available options may vary.
`Does the new element require internationalisation?`	Internationalized elements require additional resource bundles, which are created if you answer yes.
`Does the new element require JavaScript?`	If you respond yes, a suitably formatted JavaScript file will be created and associated with the new element.
`Will the new element have sub-elements?`	If you respond yes, the generated element will be a rich element like the “Product Price” element using in the “Product Details” widget.
`Please select any configuration options required in the new element`	This enables you to select how the element can be configured by the user inside the Design page. The options displayed depend on your response to the “Please select the type of the new element” question.
`Should the new element be inserted as a span (if not, it will be inserted as a div)?`	This question determines how the element mark-up will be inserted in the HTML DOM.
`Should the new element contain example source code?`	If you respond yes, the generated element will contain various instructions and examples to assist those new to widget development. If you answered yes to the “Will the new element have sub-elements?” question, example sub-elements will be created.
`Should the new element be sent to the instance immediately?`	If you respond yes, the new element will be created as an extension on the target Commerce instance. If you answer no, the element is created the next time you do a `putAll` or `transferAll`.

Create a stack with the ccw utility

When you create a new stack, the `ccw` script asks a number of questions in order to create a stack.

Understand the ccw utility stack creation questions

When creating the stack, you are prompted to answer a number of questions. The answers provided to these questions will determine how the Site Settings are created. The following section provides details of the questions asked during the creation of the stack.

Question	Explanation
`Stack name`	Name of the new stack type.
`Max number of variants`	Maximum number of sub-regions that the stack supports.
`Default variants`	Number of sub-regions a new instance of the stack will have by default.
`Requires internationalization`	Generate additional language resources at creation time.
`Include example code`	If yes, generate some example template and style code.
`Server Sync`	If yes, automatically send the new stack to the server.

Create Site Settings with the ccw utility

Site settings are parameters that are used by the code. They are added to your site view model, and are available to all widgets. Using site settings, you can create a single setting that controls a feature used by multiple widgets. Site settings are contained within extensions. For information on working with extensions, refer to Develop Server-Side Extensions. Once you have uploaded, the site settings allow you to create custom settings in the administration interface. You can use the Admin REST API to work with site settings, as described in the REST API Site Settings documentation.

You can use the ccw utility to create a new group of Site Settings. To do this, the ccw script asks a number of questions in order to create a Site Settings group.

Question	Explanation
`Site Settings name`	Display name of the new Site Settings group.
`Include example code`	If yes, then generate some example metadata.

Understand the ccw utility Site Settings creation questions

When creating the Site Settings, you are prompted to answer a number of questions. The answers provided to these questions will determine how the Site Settings are created. The following table provides details of the questions asked during the creation of Site Settings:

Delete content with the ccw utility

The `ccw` utility creates widgets, stacks and elements as extensions, allowing you to delete previously created widget, stacks, or element content by deactivating and deleting the corresponding extension.

To delete the widget, stack, or element from your local disk after you have deleted the extension, issue the following command:

```
ccw --grab --clean
```

Use the plsu utility

The Page Layout Synchronization Utility (PLSU) enables you to manage and edit page layouts on your server. You can use the following operations with the pls utility:

- `--list` – Displays the names of all of the layouts on the specified server

- `--delete` – Deletes one or more page layouts on the specified server
- `--transfer` – Copy one or more page layouts from a specified server to another

List page layouts

The following command lists all of the page layouts available on a server:

```
plsu --list --node https://commerceCloudInstance --applicationKey applicationKey
```

The response might resemble:

```
Home Layout
Collection Layout
Product Layout
Cart Layout
Cart Layout With Shipping
Checkout Layout
Checkout Layout with GiftCard
Order Confirmation Layout
About Us Layout
Contact Us Layout
Privacy Layout
Returns Layout
Shipping Layout
Error Layout
Search Results Layout
No Search Results Layout
Payer Authentication Layout
Wish List Layout
Purchase List Layout
Profile Layout
Account Based Profile Layout for B2B
Order Details Layout
Order History Layout
Wish List Profile Layout
Scheduled Order Layout
Assets Layout
Asset Details Layout
```

Delete page layouts

You can delete a page layout using a command similar to the following:

```
plsu --delete --name "page layout to delete" --node  https://
commerceCloudInstance --applicationKey applicationKey
```

You can specify the `--name` option more than once. Note that default layouts cannot be deleted.

Transfer page layouts between servers

You can transfer a single page layout by using the following command:

```
plsu --transfer --node https://commerceCloudInstance --applicationKey
applicationKey --name "Page Layout To Transfer" --destinationNode
https://commerceCloudInstance --destinationNode
destinationApplicationKey
```

You can specify the `--name` option more than once.

To transfer all page layouts, use the following command:

```
plsu --transfer --node https://commerceCloudInstance --applicationKey
applicationKey --all --destinationNode https://commerceCloudInstance
 --destinationNode destinationApplicationKey
```

Page layouts that do not exist on the target system will be created. Note that, unlike dcu `--grab`,

`plsu --transfer` does not create anything on your local disk.

Certain page layout-related features may refer to other content on your Commerce instance. If this is the case, plsu will not transfer a page layout if it relies on data that is not already present on the destination system. As such:

- Widget instances used within the layout must already exist. These can be created by performing a dcu `--put` or `--transfer` before calling plsu
- Sites associated with page layouts must already exist. These can be created using the administration interface
- Audiences associated with page layouts must already exist. These can be created using the administration interface

Before transferring page layouts, plsu `--transfer` performs all error checking. If the utility finds missing data, no page layouts will be transferred.

Understand plsu options

The following options are available with the plsu utility:

Question	Explanation
`-h, --help`	Provides information to assist with running the utility.
`-V, --version`	Provides the utility's version number.
`-n, --node <node>`	The URL for the administration interface on the Commerce instance. For example, `https://localhost:9080`. This is a required option. When used with `--transfer`, this value indicates the source instance. The destination instance should be specified by using `--destinationNode`.

Question	Explanation
-k, --applicationKey <key>	The application key to use to log into the Commerce administration interface. This is required for authentication. For detailed information on creating an application key, refer to the Use the application key for authentication section. You can also specify the application key using the CC_APPLICATION_KEY environment variable.
-l, --locale <locale>	Forces the default locale to the supplied value.
-d, --detinationNode <node>	The URL of the destination administration interface on the Commerce instance, for example: https://localhost:9080. This option is used with --transfer.
-a, --destinationApplicationKey <key>	The application key to use when logging into the destination Commerce administration interface. This key is required for authentication purposes. For detailed information on creating an application key, refer to the Use the application key for authentication section.
-y, --name <name>	The name of the page layout. You can use this option multiple times, and with –-delete and –-transfer. You cannot use this option with --all.
-s. --all	Specifies that all page layouts should be transferred. This option is not available with --all.
-i, --list	This lists all of the page layouts available on the system.
-e, --delete	This option, which must be used in conjunction with the --name option, deletes one or more page layouts from the Commerce system.
-t, --transfer	This option transfers one or more page layouts from one system to another. This option must be used in conjunction with --name or --all options.
-g, --ignoreVersions	This option enables plsu to transfer layouts between Commerce instances that are not at the same release version and, as such, should be used with caution.

Manage Large Scale Transfers

While DCU is primarily a developer toolchain that bridges the gap between a developer's machine and a Commerce instance, you can also use it for large scale transfers, however, there are some limitations.

The quickest and easiest way to promote widget code and assets between Commerce instances is via a database copy. The promotion of client code between two Commerce instances will typically require the use of both the plsu and dcu utilities. **Note:** when performing a large scale transfer you should run dcu first, this is because plsu expects all the widget instances used in the page layouts being transferred to be made available. You should

also note that large scale transfers using dcu can take several hours to run, depending on the network speed and the number and size of assets being copied.

It is recommended that you should identify what has actually changed and transfer only those assets, which is a quicker method than copying all assets every time. When transferring assets between Commerce instances, it is also recommended that they are at the same release version.

New widgets and other assets should always be created with ccw and not via an extension. If new widgets are being created using an extension, the version number should be set to one and never changed. Version numbers are intended to be used with Oracle widgets only and can cause transfer issues if misused. Adding non-standard files and directories under widgets is only partly supported by dcu. When a widget is initially created on the target instance during a `transferAll` or `putAll`, any unrecognized files found under the widget directory are included in the extension. However, these non-standard additional files, such as, graphic files, cannot be changed on the server without first deactivating the corresponding extension and re-running a `transferAll` or `putAll`. Performing a `put` or transfer on these individual files will have no effect. **Note:** you can use dcu to delete unwanted widget instances, as described later in this section.

Use dcu for large scale transfers

To transfer all Design Studio assets between two Commerce instances, you would typically call dcu as shown below:

```
dcu -x . -n https://destinationInstance -k <destinationInstance
Application Key>
```

However, dcu has a number of additional command line options that may be useful in large scale transfer scenarios.

The first of these is the `--noThemeCompile` option. By default, when dcu updates a less file, the server will trigger a Theme recompilation which can significantly slow down large scale transfers. Specifying the `--noThemeCompile` option will mean that no Theme recompilation is triggered and you will have to manually trigger the Theme recompilation after the transfer is complete. You can manually trigger a Theme recompilation either via the UI or by transferring a single less file with dcu when complete.

The second option to consider with large scale transfers is the `--autofix` option. Normally, when dcu finds a missing widget instance during a transfer, it will attempt to create a new widget instance at the requisite version and apply the changes. For user created widgets, this works as expected but for Oracle widgets on systems which may not have exactly the same widget versions, this can cause a conflict if a widget instance already exists of the same name but a different version. If you supply the `--autofix` option and dcu finds a conflicting widget instance, it will attempt to create that instance at the same version number as on the source instance after first renaming the conflicting widget instance. The latter assumes that the widget exists on the target system at the desired version. You may find transfers are somewhat slower when the `--autofix` option is used. This option can result in a lot of widget instances existing on the target system so use it with discretion. Once the DCU transfer is complete, you should review the output and check for any errors or warnings.

Use plsu for large scale transfers

When all widget and related assets have been transferred, plsu can then be called to transfer page layouts. To transfer all page layouts between two instances, you would typically call plsu as shown below:

```
plsu –n https://sourceInstance –k <sourceInstance Application Key> –d https://
destinationInstance –a <destinationInstance Application Key> -ts
```

In general, it is recommended that transfers take place between Commerce instances which are at the same release version. While the plsu utility does have an `–ignoreVersions` option, this should be used with care. Transfers using plsu will typically take much less time than with dcu. Any missing stack instances will be created during a plsu transfer. **Note:** plsu does not require a grab to be run first; it does a point to point data transfer.

Use dcu to delete widget instances

In order to remove a widget instance from an Oracle Commerce instance, you would typically call dcu as shown below:

```
dcu –k <destinationInstance Application Key> --delete widget/WidgetName/
instances/WidgetInstanceToDelete
```

This removes all traces of the widget instance from both the local grab directory and the Commerce instance.

Use the JavaScript Code Layering User Interface feature

The JavaScript Code Layering feature in the administration interface lets you extend the JavaScript of an Oracle Commerce provided widget with your own custom JavaScript.

Previously, if you wanted to customize or extend the JavaScript for a provided widget, you had to download the widget and its related files, edit the code with your own custom JavaScript, and upload it back to the system as a custom widget type. With the JavaScript Code Layering User Interface feature you can open an additional user interface that lets you layer custom JavaScript on top of the provided widget and which then has the benefit of staying on the provided widget.

Extend JavaScript in a provided widget

As an example of how this feature works, do the following:

1. Open the **Design** page and click the **Components** tab.
2. Filter to find the Header widget.
3. Highlight the Header widget.
4. Click the widget name to expand its details.
5. Click **Extend JavaScript**. This action opens a new JavaScript file user interface window that acts as a template to be edited and associated with the original widget. You can edit this file with your own custom code and then save the file. This custom JavaScript file is then shared between all instances of the widget after it is saved.

In the case of this Header widget example, when you click Extend JavaScript, a new JavaScript file is generated that acts as the template for you to begin adding your own custom JavaScript. With the Header widget the template JavaScript file looks like this:

```
/**
 * @fileoverview extendheaderWidget_v10.js
```

```
 *
 * @author
*/
define(
  //
  // DEPENDENCIES
  //
  [],

  //-------------------
  // MODULE DEFINITION
  //-------------------
  function() {

     "use strict";
      return {
      onload: function(widget) {
      // console.Log('extendheaderWidget_v10.js onLoad');
      },

      beforeAppear: function () {
       //console.log('extendheaderWidget_v10.js before appear');
      }
    };
  }
);
```

With the Header widget, there are two functions in the template that it to the current widget model/lifecycle and these two function are:

- `onLoad` – runs custom logic when the widget is instantiated
- `beforeAppear` – used when you want to run custom logic each time the widget appears on the page

You could now edit this file with some custom code and save it to associate it with the widget. This code shows an example of some custom JavaScript that was added to the original template file. The custom code changes create a new knockout observable on the widget and updates the existing widget knockout observable variable `logoAltText`.

```
/**
 * @fileoverview extendheaderWidget_v10.js
 *
 * @author
*/
define(
  //
  // DEPENDENCIES
  //
  ['knockout'],

  //-------------------
  // MODULE DEFINITION
  //-------------------
```

```
  function(ko) {

     "use strict";
      return {
      onload: function(widget) {
        console.Log('extendheaderWidget_v10.js onLoad');
      // declare a new variable
      widget.myTestVariable = ko.observable('');
      // update an existing variable
      widget.logoAltText('Updated logo text in extension');
      },

      beforeAppear: function (Page) {
       console.log('extendheaderWidget_v10.js before appear');
      }
    };
  }
);
```

Toggle JavaScript minification in preview

By default, widget and element JavaScript files are minified when you view a storefront in preview mode.

To aid in debugging your custom widgets, you can turn minification off, allowing you to see the complete widget and element JavaScript source code in your browser's debugging tools. To toggle JavaScript minification, do the following:

1. Click the **Preview** button to preview your storefront and its unpublished changes in Preview mode.
2. Click the **Debug Tools** menu.
3. Use the Minify JavaScript option to enable or disable minification.
4. Refresh the page or navigate to a new page to force a full page refresh.

Note: The Minify JavaScript option only affects the widget and element JavaScript files. Other JavaScript libraries will continue to be minified.

Reduce the size of page responses

The first time a page loads in Commerce, the server returns all of the data associated with the specified page layout, including the widget template source, element source, and so on.

On subsequent page loads, it is possible to limit the returned data to only those widgets that have not been previously rendered. This feature is not enabled out of the box because you should understand the trade-offs involved before using it.

When this feature is disabled, all of the data for the page is returned for every page call and it is cached, giving you the performance improvements associated with caching. When the feature is enabled, the system tracks the URLs that have been visited to determine which widgets have not been previously rendered and then limits the returned data to those widgets. Caching all of these URLs is not feasible, however, so caching is turned off when this feature is enabled. Instead, the performance improvements that are gained are driven by

the drastically reduced amount of data that is returned from the server. This approach is especially beneficial on mobile devices.

To enable this feature, you must create an extension that uploads an application-level JavaScript module that depends on the `cc-store-configuration-1.0.js` library and sets the `enableLayoutsRenderedForLayout` flag to `true`. The following code sample shows what the contents of this JavaScript module might look like (for general information on creating an application-level JavaScript extension, see Create an Extension):

```
define(
  //-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
  ['ccStoreConfiguration'],
  //-------------------------------------------------------------------
  // Module definition
  //-------------------------------------------------------------------
  function(CCStoreConfiguration) {

    "use strict";

    return {

      storeConfiguration: CCStoreConfiguration.getInstance(),

      enableLayoutsRenderedFeature: function() {
        storeConfiguration.prototype.enableLayoutsRenderedForLayout =
true;
      }

    };
});
```

In addition to enabling the feature, you may also choose to override the `storeLayoutIdsRendered()` and `getLayoutIdsRendered()` methods by doing the following:

- The `storeLayoutIdsRendered()` method stores IDs for the layouts visited by the shopper until the page is refreshed or the browser is closed. The stored layout IDs let the server know which pages have been cached so it can limit the returned data to new widgets. The number of IDs that the `storeLayoutIdsRendered()` method stores is determined by the size of the `layoutsRenderedArraySize` object and is set to 15 out of the box.
  You can also choose to override the `storeLayoutIdsRendered()` method for other purposes, for example, so that layout IDs are only stored when the store is accessed on a mobile device.
- The `getLayoutIdsRendered()` method returns the stored layout IDs to the server. You can modify this method so that it ignores certain layouts, effectively disabling the feature for those pages.

The following code sample shows the type of modifications you might choose to make to the `storeLayoutIdsRendered()` and `getLayoutIdsRendered()` methods:

```
define(
  //-------------------------------------------------------------------
  // DEPENDENCIES
  //-------------------------------------------------------------------
  ['ccStoreConfiguration'],
  //-------------------------------------------------------------------
  // Module definition
  //-------------------------------------------------------------------
  function(CCStoreConfiguration) {
    "use strict";
    return {
      storeConfiguration: CCStoreConfiguration.getInstance(),
      enableLayoutsRenderedFeature: function() {
         storeConfiguration.prototype.enableLayoutsRenderedForLayout = true;
       },
      storeConfiguration.storeLayoutIdsRendered: function(pLayout) {
         // Store layouts only if the device is mobile
         storeConfiguration.prototype.layoutIdsRendered.push(pLayout.layout);
       },
      storeConfiguration.getLayoutIdsRendered: function() {
              // Don't return the home page's layout ID, effectively
disabling this
              // feature for the home page.
              if
(storeConfiguration.prototype.layoutIdsRendered.indexOf('home') > -1) {
                    storeConfiguration.prototype.layoutIdsRendered.splice(
                      storeConfiguration.prototype.layoutIdsRendered.indexOf(
                         'home'), 1);
              }
              return storeConfiguration.prototype.layoutIdsRendered;
       }

    };
})
```

View client-side error logs

Client-side errors are stored in log files that roll over every day.

Commerce stores 30 days' worth of these logs files. To access the log files, both the Store API and the Admin API have a `logging` endpoint. The syntax for requests sent to these endpoints is the same, as shown in the following example:

```
GET /logging?type=clientErrors
```

A request to the Store API `logging` endpoint should be sent to a storefront server where it retrieves client errors logged on that server, for example:

```
["log line 1",
"log line 2",
"log line 3"]
```

A request to the Admin API logging endpoint should be sent to the administration server. The Admin API logging endpoint executes concurrent REST requests to retrieve client errors from each registered storefront server. The Admin API endpoint collates the results from the store requests and returns them as one JSON object, keyed on the hostname of the storefront server, for example:

```
{
  "server 1": ["log line 1",
  "log line 2",
  "log line 3"],
  "server 2": ["log line 1"]
}
```

The log information returned by either endpoint can be filtered using the optional query parameters described in the following table:

Parameter	Format	Default	Description
`since`	ISO 8601 timestamp format OR millisecond value	null	The date and time to start searching from.
`until`	ISO 8601 timestamp format OR millisecond value	null	The date and time to search to.
`includeArchives`	boolean	false	Controls whether the search includes archived log files.
`numArchiveFiles`	int	7	Searches the most recent N archived log files. A value of 0 searches all archived log files.
`queryText`	String	null	Limits the results to log entries containing the supplied query text.

Parameter	Format	Default	Description
`localLogs`	boolean	false	This parameter is supported by the Admin API endpoint only. It returns errors from the local log file on the administration server, without calling out to the storefront servers. You can use it to retrieve client-side errors thrown by the storefront preview feature.

Restore or upgrade the storefront framework version

Commerce's storefront framework provides the foundation for store functionality and facilitates client-side storefront development.

The framework comprises JavaScript and CSS code and gives access to common view models, the PubSub event mechanism, third-party libraries, themes, Commerce-specific libraries and so on. It consists of the following files: `require.js`, `main.js`, `store-libs.js`, and `storefront.css`.
The storefront framework is upgraded automatically with each release. While much care is taken to avoid backwards compatibility issues, the ability to revert an automatic upgrade provides a stop-gap measure in case an automatic upgrade is incompatible with custom widget or theme code you have written. This feature allows you to restore the previous version of the framework while you make the necessary changes to your custom code and then upgrade to the latest version.

Note: After you restore or upgrade, the changes are effective immediately.

To change the storefront framework version:

1. Click the **Design** icon.
2. Click the **Developer** option and select **Storefront Framework**.

   You are presented with one of the following scenarios:

   a. The latest version is currently running but no previous version exists. In this case you will see a message informing you that you cannot be upgraded as you are running the latest version. This is most likely the case for new customers.

   b. The latest version is currently running, and a previous version exists which can be restored. Click the **Restore** button to restore the previous version and then confirm your choice.

   c. An older version is currently running, and the latest version is available as an upgrade. The Storefront Framework page displays details of the version currently being used, and details of the latest version. Click the **Upgrade** button and then confirm your choice.

Once upgraded, the storefront framework version is set to the latest version by default.

49

Improve System Performance

This section provides information on ways you may be able to improve system performance by considering specific coding strategies during your development process.

We recommend that you review this information before your site goes live in production.

Measure performance often

From the beginning of your project, regular performance measurement using several tools are key to regularly measure your page load performance.

If you are using http://www.webpagetest.org, you should target a maximum speed index metric of 3000, ideally striving for a value of less than 2000. Web Page test can show you how much content you are loading and how your site appears from multiple locations and in multiple devices. You should also consider using the Chrome Developer Tools timeline to review how different elements in your pages load under different circumstances, Google Page Speed, as a best practices reporting tool, and the Chrome Lighthouse extension to perform an audit of performance, SEO, accessibility, etc., and provide a best practices report.

After you go live with you project, Chrome User Experience reports can show you real world measurements of site performance.

Use of these tools and your implementation of the recommendations that follow in this chapter are your best chance of improving your customer experience.

Monitor your Commerce environments

External monitoring of your Commerce environment, while permitted, should be used carefully.

External monitoring of your Commerce environments is permitted. Oracle reserves the right to remove or disable access to any tools or technologies that are, in Oracle's judgement, impacting the performance or the availability of the Commerce Service or your own Commerce environments. External monitoring performed by you does not replace or supersede Oracle's own monitoring for availability measurement and SLA calculation purposes.

Recommendations for effective monitoring

While monitoring generally generates a low level of traffic to your site, minimizing it is still worthwhile from a performance perspective. Configuring the monitoring tool to make `HTTP HEAD` requests instead of `GET` or `POST` requests can minimize the impact monitoring. Also, choose reasonable monitoring intervals such as 1 or 5 minutes.

Use tools that have multiple, geographically spread out monitoring hosts to help you separate out issues with reachability of the Commerce service from a given location from issues with the Commerce service itself. Given the complexity of the Internet there are many reasons why Commerce could be temporarily unreachable (or reachable with delays) from a single, specific location while Commerce is still available to the rest of the globe and is running

without issues. Examples of this type of reachability issue include local ISP outages or outages with Public DNS services.

Improve performance in REST API Calls

You may be able to improve performance by coding REST API calls in a way that minimizes the size and number of response from the server.

There are specific query parameters provided in support of this, and we recommend that use them wherever they are applicable. Avoid using multiple endpoint calls where a bulk option exists, for example, fetching data per SKU instead of per product, which for some endpoints would give data for all child SKUs of that product. As discussed in the section that follows, consider a review of your code.

Review code for possible bulk operation on endpoints

To potentially improve performance, review your endpoint calls to check for code that invokes the same endpoint multiple times and consider using a bulk version for that given endpoint.

For example, you can make one call with a comma-separated list of product IDs, as shown in the example below:

```
/ccstore/v1/products?productIds=Product_19Cxy,Product_15CD
```

You could do this instead of making individual calls, as shown below:

```
/ccstore/v1/products?productIds=Product_19Cxy
 /ccstore/v1/products?productIds=Product_15CD
```

For more information, see REST API query parameters.

Use `cc-storage` for Safari private browsing mode

Note that `localStorage` is not available in Safari in private browsing mode.

For this use case, use `cc-storage` instead.

Avoid `console.log()` statements

Check your code for `console.log()` statements which can reduce performance and introduce bugs if the log are not available.

Avoid using `ko.observable()`

To improve performance, do not create `ko.observable()` for data that does not change.

Update observable JavaScript arrays

When working with observable JavaScript arrays, update the underlying JavaScript array and then call `valueHasMutated()` on the array.

Use Knockout data-binds syntax to attach events to DOM elements

You may be able to improve performance by avoiding using JQuery to access the DOM (Document Object Model) and attaching events to DOM elements.

Instead, use the variables on the widget view models to access and modify any content on the template. You should never access the DOM directly. For more information, seeUnderstand widgets .

Use `onLoad` and `beforeAppear` correctly

You may be able to improve performance by limiting the amount of processing that has to be done by the browser by avoiding additional coding when using `onLoad` and `beforeAppear`, since the more code that is executed at this point, slower and longer the page load time may appear.

Also, do not make synchronous endpoint calls in `beforeAppear` or `onLoad` since rendering will be blocked until they return. An example of good practice of `beforeAppear` follows:

```
beforeAppear: function(page) {
    var self = this;
    setTimeout(function () {
        self.pause(2000)
        self.messagge("Pause Completed")
    },
    500);

},
pause function(millis) {
    var date = new Date();
    var curDate = null;
    do {
        curDate = new Date();
    }
    while (curDate - date < millis);
    }
```

For more information, see Understand widgets.

Use the fields parameter

You may be able to improve performance by using the `fields` parameter on endpoint calls wherever possible and by limiting the amount of data you request from the server for each endpoint call to a specific set.

You should avoid using the default with endpoint calls, since that will fetch all fields. For example, if you use the default when calling the `getCollections` and `listProducts` endpoints, this would result in a large payload and longer server response time. Remember that you usually only need a subset of fields for the UI. For more information, see Understand widgets.

Use persistent filters

You may be able to improve performance by using persistent filters that store the set of properties to include or exclude in a given REST call's response, which can make optimizing the performance of these calls easier and more maintainable.

For more information, see Response filters.

Use minified versions of libraries and widget JavaScript

Use minified versions of JavaScript libraries and widget JavaScript files to help the libraries load faster.

Also, avoid inline JavaScript since it can be difficult to maintain and instead consider using widgets and `RequireJS` modules.

Localize endpoints

You can localize some endpoints to page scope, so that when a page is changed the success callback is not executed.

This decreases unnecessary view model activities not related to the current page. You can eliminate issues such as load of previous page and display of redundant error notifications when you make this change.

To include this behavior, set the `discardPageScopedCallResponses` flag to true in `cc-store-configuration-1.0.js` file. The calls included in the `pageScopedCalls` array will be page scoped once you enable this flag.

Enable queueing simultaneous endpoint calls

You can decrease multiple simultaneous calls to the same endpoint and serialize some of the endpoints.

This will decreases the number of calls to the server as well as decreases the chances of write locks on the same resource on the server. This will be particularly useful in the case of pricing calls.

To include this behavior, set the `enableQueueingSimultaneousCalls` flag to true in `cc-store-configuration-1.0.js` file. Include the calls you want to be queued in the `queueableCalls` array of the same file.

Improve performance in custom widgets

To improve performance in custom widgets, use the unique line item ID distinguisher `commerceItemId`.

This field becomes required for split shipping when `combineLineItems` is set to no.

Optimize Search

Performance gains can be achieved by optimizing search.

The following topics consider search optimization.

Response Optimization for Product Data

The search response will contain any property flagged in the catalog as either searchable or as a facet. Many of these are not needed for rendering the products in the search results pages, and are therefore unnecessarily increasing the response size. The following article on medium.com outlines how to do this:

https://medium.com/oracledevs/limiting-fields-returned-by-search-in-oracle-commerce-cloud-bcc6e86ec614

Response Optimization for Faceted Navigation

Faceted navigation appears in Oracle Commerce anywhere that products are listed. By default, any facet that has data for the products being shown will automatically be displayed. If some of these facets are not required either globally or in a specific destination (for example, you may choose to only display Size when particular collections have been selected), the Facet Ordering functionality in Search Admin can be used to limit this via the following steps:

1. In Oracle Commerce Admin, select the **Search** section
2. Choose the **Facet Ordering** option.
3. Edit the "default" entry to limit facets globally:
    a. List the facets you do want to return.
    b. De-select the **Include remaining facets** option.
    c. Save and Publish.
4. Alternatively, you can restrict the list of facets for a specific collection or search:
    a. Create a new facet list by selecting **New Facet Ordering Rule**.
    b. Edit the Collection or entering one or more search term(s).
    c. Choose the facets to be displayed.
    d. De-select the **Include remaining facets** option.
    e. Save and Publish.

Use `preFilter` parameter with fields parameter to improve endpoint performance

The `preFilter` boolean parameter can be used with fields parameter to improve endpoint performance, but this can only improve the performance when the resource being requested has structural depth.

With flat structure resources, your performance may not improve. By default this feature is not enabled at endpoints. On store side it is enabled on these endpoints.

```
listSkus: /ccstoreui/v1/skus GET

listProducts: /ccstoreui/v1/products GET

listOrganizations: /ccstoreui/v1/organizations GET
```

Following is the example of how to use `preFilter` parameter with the `sku` endpoint:

```
/ccstoreui/v1/skus/?
skuIds=ProdId&fields=listPrices.US_Dollar:repositoryId&storePriceListGr
oupId=US_Dollar&catalogId=reseller&preFilter=true
```

You can also use `preFilter` for `listProducts` and `listOrganizations` endpoints same as the `/skus` endpoint:

```
/ccstoreui/v1/skus/?
skuIds=ProdId&fields=listPrices.US_Dollar:repositoryId&storePriceListGr
oupId=US_Dollar&catalogId=reseller

/ccstoreui/v1/skus/?
skuIds=ProdId&fields=listPrices.US_Dollar:repositoryId&storePriceListGr
oupId=US_Dollar&catalogId=reseller&preFilter=true
```

Note: The response of these examples may not be same.

Issues with fields parameter

The fields parameter does not provide a performance improvement for the first endpoint invocation and with non-cached endpoints. If you set `preFilter=true`, and if the response contains a nested structure and you do not want nested data always, in those cases if you are using fields parameter with `preFilter` flag, you will see a performance improvement.

Speed up system response on Product Listing and Product Details

A global configuration setting can be used to skip loading certain information from the product type data, which can significantly reduce the amount of data returned and speed up system response on Product Listing and Product Details.

Set the global client-configuration endpoint setting `skipLoadingProductTypes` flag set to true to skip loading and using variant information and SKU properties from the product type data. To enable this flag, invoke this admin endpoint:

```
PUT /ccadmin/v1/merchant/clientConfiguration
```

with

```
{
"skipLoadingProductTypes":true
}
```

When `skipLoadingProductTypes` is set to `true`, Commerce will skip loading all the `productTypes` on page endpoint response.

Note: If `skipLoadingProductTypes` is set to `true`, the `productTypesRequired` query parameter will be set to `false` in page calls.

Skip retrieving Parent Category information from related products

A global configuration setting can be used to skip retrieving Parent Category information from related products when not needed. This can significantly reduce the amount of data returned and speed up system response on Product Listing and Product Details. Note that the parent category information will be retrieved along with product information when shopper selects to view a related product.

Set the global client-configuration endpoint settings `optimizeRelatedProductsForPageEndpoint` and `optimizeRelatedProductsForListProductsEndpoint` flags to `true`, to skip retrieving Parent Category information from related products. This flag is set to false by default. To enable this flag, invoke this admin endpoint:

```
/ccadminui/v1/merchant/cloudConfiguration
```

with

```
{
"optimizeRelatedProductsForPageEndpoint": true
"optimizeRelatedProductsForListProductsEndpoint": true
}
```

When set to `true`, `optimizeRelatedProductsForPageEndpoint` removes `parentCategories`, `parentCategory`, `parentCategoryIdPath` from Product Details page endpoint and `optimizeRelatedProductsForListProductsEndpoint` removes `parentCategories`, `parentCategory`, `parentCategoryIdPath` from Product Listing page endpoints, that is, the `listProducts` and `listProductsForLargeCart` endpoints.

Clear `JSONStoreCache` to reflect it in storefront.

Enable asynchronous orders flow

You can receive orders and process them asynchronously without waiting for the client response. Processing performance can be improve in situations where you have orders with large numbers of items (greater than 250).

To enable async order support feature in production, perform the following steps:

1. Set `enableUpdateSettings` property to true on the `OrderQueueSettings` component.

2. Set the `enabled` flag using `orderRetrySettings` admin endpoint:

```
/ccadminui/v1/merchant/orderRetrySettings PUT
{
"enabled": true
}
```

3. Use `placeOrderAsync` flag in the submit order request at order level. Set it as true to enable async submit for an order. While placing an order through UI, merchant has to override method `isPlaceAsyncOrder` in `cc-store-configuration.js` using `applicationJS`. This method will contain the logic to decide whether to place an order asynchronously or not. OOTB, the order is placed synchronously through UI (unless resilient orders feature is enabled, in which case, order will follow the resilient order submit flow).
4. Set appropriate `asyncSubmitRetryDelays` using `orderRetrySettings` admin endpoint:

```
/ccadminui/v1/merchant/orderRetrySettings PUT
{
"asyncSubmitRetryDelays": [15,20,...]
}
```

Note: The default `asyncSubmitRetryDelays` values are 10,25,65,125,185

5. To move a QUEUED order to INCOMPLETE state rather than SUSPENDED state(default) once retries are exhausted and the latest error is of input kind:

```
/ccadminui/v1/merchant/orderRetrySettings PUT
{
"moveToIncompleteStateOnInputError" : true
}
```

Improve Storefront Performance for Large Carts

When a shopper creates an order by adding a large number of items to their shopping cart, performance can suffer, sometimes to the point where the shopper cannot complete or submit the order.

This document describes how to configure Commerce to improve performance for large-cart orders. This section includes the following topics:

- Understand large cart support
- Enable support for large carts
- Understand view model support for large carts
- Sample widgets and elements

Understand large cart support

From a performance perspective, a large cart is a shopper's cart that contains 200 items or more. Carts this large may experience performance issues. Shoppers typically create orders this large using the Commerce Quick Order widget.

You can enable large cart support by configuring the `CCStoreConfiguration.largeCart` property. This reduces the number of situations where large cart information is refreshed (price calls), improving overall performance. All cart information always refreshes during the checkout phase.

New endpoints also enable large cart batching during GET product/SKU calls.

Enable support for large carts

To enable deferred pricing, set this property in `application.js`:

```
CCStoreConfiguration.largeCart
```

By default, this property is not set.

To override deferred pricing, use this method:

```
CCStoreConfiguration.isLargeCartCCStoreConfiguration.isLargeCart
```

You do not need to do anything to enable the use of large-cart endpoints, but you can change the number of items in a cart that triggers large-cart endpoints by using these properties:

- `CCStoreConfiguration.batchSizeForProdAndSkuData` is a property that specifies the number of items that must be in a cart before Commerce automatically uses `listProductsForLargeCart` and `listSkusForLarge` Cart instead of `listSkus` and `listProducts`.
- `CCStoreConfiguration.threshholdSizeForStockStatusData` is a property that specifies the number of items that must be in a cart before Commerce Cloud automatically uses `listStockStatusesForLargeCart` instead of `getStockStatuses`.

Understand view model support for large carts

This section describes the Cart view model properties and methods that support deferred pricing calls and batching the Store API Products and SKUs endpoints.

`CartViewModel.hardPricing` is a property that triggers cart pricing when the `CCStoreConfiguration.largeCart` flag is set to true. See Sample widgets and elements for a code sample that uses this property to trigger pricing. The default value of this property is false.

`CartViewModel.userActionPending` is a property that you should set to true whenever you update the cart and the `largeCart` flag is set to true. Use this flag to display messages reminding the shopper to save the cart. You can also use this flag to display messages that remind customers that prices are approximate and they will have to click the save cart button to trigger pricing and see the correct prices.

`CartViewModel.updateItemPriceForLargeCart` is a method to calculate the item total with an approximate price when you defer pricing calls. This method calculates prices by using list prices and sale prices. You can override this method to accommodate manual pricing calculations.

Use the method `CartViewModel.updateItemShippingGroupRelationShipForLargeCart` to update shipping group relationships from the client side when you have deferred pricing calls.

Use `CartViewModel` to implement these functions to update cart data when you defer pricing calls:

- `CartViewModel.updateCartItemDataForLargeCart` is a method to update the cart total and subtotal list price and the sale price of an item added to the cart when you defer pricing calls. It also updates `numberOfItems` for the Cart Summary widget. You can override this method to accommodate manual pricing calculations.
- `CartViewModel.updateCartAfterLoginForLargeCart` is a method to update cart data after a registered shopper logs into their account.

`CartViewModel.priceCartBeforeRefreshInLargeCart` is a method to reprice the entire cart.

`CartViewModel.getBatchSizeForProdAndSkuData` is a method that returns the number of products or SKUs whose data you need to retrieve in each `listProduct` or `listSku` request. If this is set to 1, it indicates that all products or SKUs will be sent to `listProduct` or `listSku` at once. Any other positive value indicates the number of products/SKUs sent in one call to the `listProduct` or `listSku` endpoint. In this case, multiple calls to these endpoints retrieve data for all IDs.

`CartViewModel.getThreshholdSizeForStockStatusData` returns the threshold on the number of SKUs beyond which a POST equivalent of the stock call would trigger.

Sample widgets and elements

This section describes sample widget code that you can use to implement support for large carts.

Create an element to manually trigger pricing

The following sample button uses the `hardPricing` flag to let the shopper manually price the cart.

```
saveCartButtonHandler :function(){
 this.cart().hardPricing = true;
 this.cart().markDirty();
},
```

When pricing calls are deferred and you enable the large cart feature, all the item update/delete/add operations normally triggered from the Product Details widget, Shopping Cart widget, Header widget, Quick Order widget, the copy order operation from Order Details widget, and the Purchase list widget are handled from client side.

Shopping Cart Widget and Header widget implementation

You can remove line items from the Header mini cart and Shopping Cart widget. The following code sample is the basic implementation of client side handling for the removal of line items. Line items here correspond to shipping group relationship and the `removeShippingGroupRelationShip` method as the handler for the remove items event.

```
removeShippingGroupRelationShip :function(cartItem,shippingGroupRelatio
nship){
  if(this.storeConfiguration.isLargeCart() === true){

    var
quantityChange=(-1)*shippingGroupRelationship.updatableQuantity();
    var price=cartItem.productData().childSKUs[0].salePrice ?
cartItem.productData().childSKUs[0].salePrice :
```

```
cartItem.productData().childSKUs[0].listPrice;
    //remove the SGR
    cartItem.shippingGroupRelationships.remove(shippingGroupRelationship);
    if(cartItem.shippingGroupRelationships().length === 0){
      this.cart().items.remove(cartItem);
    }else{
      cartItem.quantity(data.quantity()+ quantityChange);
      cartItem.itemTotal(data.itemTotal()+price*quantityChange);
    }
    //update the number of items
    this.cart().numberOfItems(this.cart().numberOfItems()+quantityChange);
    this.cart().updateAllItemsArray();
    this.cart().subTotal(this.cart().subTotal()+price*quantityChange);
    this.cart().total(this.cart().total()+(price)*quantityChange);
    this.cart().userActionPending(true);
    this.cart().saveCartCookie();
    $.Topic(pubsub.topicNames.CART_REMOVE_SUCCESS).publishWith([{message:"suc
cess"}]);
    return true;
  }
}
```

Widget implementation for update

Line item quantity can be updated from the Shopping Cart widget. The following code is the basic implementation of client side handling for the update of line items. These line items correspond to the shipping group relationship and `updateQuantity` methods that you can use as a handler for the update item event.

```
updateQuantity: function(data, event, id, shippingGroup) {

  if('click' === event.type || ('keypress' === event.type && event.keyCode
=== 13)) {
    // update the 'updatableQuantity' to cart-item.
    var cartItemTotal = 0;
    for(var index=0; index < data.shippingGroupRelationships().length; index+
+) {
      cartItemTotal = parseInt(cartItemTotal) +
parseInt(data.shippingGroupRelationships()[index].updatableQuantity());
    }
    data.updatableQuantity(parseInt(cartItemTotal));

    if(data.updatableQuantity && data.updatableQuantity.isValid()) {
      if(this.storeConfiguration.isLargeCart() === true){
        var quantityChange=data.updatableQuantity()-data.quantity();
        var price=data.productData().childSKUs[0].salePrice ?
data.productData().childSKUs[0].salePrice :
data.productData().childSKUs[0].listPrice;
        this.cart().numberOfItems(this.cart().numberOfItems()
+quantityChange);
        data.quantity(data.updatableQuantity());
        data.itemTotal(data.itemTotal()+price*quantityChange);
        shippingGroup.price(shippingGroup.price()+price*quantityChange);
        shippingGroup.quantity(shippingGroup.quantity()+quantityChange);
        this.cart().updateAllItemsArray();
```

```
        this.cart().subTotal(this.cart().subTotal()
+price*quantityChange);
        this.cart().shippingSurcharge(this.cart().shippingSurcharge()
+data.shippingSurcharge*quantityChange);

this.cart().total(this.cart().total()+
(price+data.productData().shippingSurcharge)*quantityChange);
        this.cart().userActionPending(true);
        this.cart().saveCartCookie();
        return true;
      }
      $.Topic(pubsub.topicNames.CART_UPDATE_QUANTITY).publishWith(
      data.productData(),[{"message":"success", "commerceItemId":
data.commerceItemId, "shippingGroup": shippingGroup}]);

      var button = $('#' + id);
      button.focus();
      button.fadeOut();
    }
  } else {
    this.quantityFocus(data, event);
  }

  return true;
},
```

Product Details widget

Adding the product from Product Details widget will not trigger pricing. The added item updates in the cart view model first and after that, pricing triggers. The new methods are only called during nested view model logic when the large cart flag is turned on, just before system shunts the pricing. This ensures that the shopper has a view of the approximate prices of the items added to cart.

When calculating prices, sale and list prices are used. In the absence of pricing data, you can override `application.js` to update the specific item price using other special parameters such as bundled pricing, volume pricing, or tiered pricing from existing product data.

`updateItemPriceForLargeCart`

The following code is the method to update items during the “single item-add” operation of `updateItemPriceForLargeCart`.

```
CartViewModel.prototype.updateItemPriceForLargeCart =
function(data,cartItem){
  var self =this;
  var price = data.childSKUs[0].salePrice ?
data.childSKUs[0].salePrice :data.childSKUs[0].listPrice;
  cartItem.itemTotal(cartItem.itemTotal()+price*data.orderQuantity);
};
```

`updateCartItemDataForLargeCart`

When calculating prices, sale and list prices are used. You can implement other scenarios based on your own requirements.

The following code shows the method to update the cart properties during the “single item-add” operation:

```
CartViewModel.prototype.updateCartItemDataForLargeCart = function(data){
  var self =this;
  var price = data.childSKUs[0].salePrice ?
data.childSKUs[0].salePrice :data.childSKUs[0].listPrice;
  self.numberOfItems(self.numberOfItems()+data.orderQuantity);
  self.events=[];
  self.shippingSurcharge(self.shippingSurcharge()+data.shippingSurcharge);
  self.subTotal(price*data.orderQuantity+self.subTotal());
  self.total(self.total()+price+data.shippingSurcharge);
  self.saveCartCookie();
};
```

You can extend this process by adding any other business related view model properties.

Adding multiple items at once (Quick order widget, Copy order, Purchase list)

Adding multiple items at once will not trigger pricing calls when large cart flag is true. Without modification, added items update in cart view model first and then, pricing triggers. The new methods are called during the nested view model logic when large cart flag is turned on, just before system shunts the pricing. This ensures that the shopper has a view of the approximate prices of the item added to cart.

When calculating prices, sale and list prices are used. In the absence of pricing data, you can override these from `application.js` to update the item price taking into account such special parameters such as bundled pricing, volume pricing, or tiered pricing from existing product data.

`updateCartItemsDataForLargeCart`

The following code is a method to update the multiple item during “multiple item-add” operation:

```
    CartViewModel.prototype.updateCartItemsDataForLargeCart = function(data){
  var self =this;

  //NOT REUSING updateCartItemDataForLargeCart to minimise mulple
notifications due to multiple update
  var
i,newQuantitiesAddedToCart=0,newAddedshippingSurcharge=0,newAddedsubTotal=0;
  for(i=0;i<data.length;i++){
    var price = data[i].childSKUs[0].salePrice ?
data[i].childSKUs[0].salePrice :data[i].childSKUs[0].listPrice;
    newQuantitiesAddedToCart+=data[i].orderQuantity;
    newAddedshippingSurcharge+=data[i].shippingSurcharge;
    newAddedsubTotal+=price*data[i].orderQuantity;
  }
 self.numberOfItems(newQuantitiesAddedToCart+self.numberOfItems());
 self.events=[];
 self.shippingSurcharge(self.shippingSurcharge()+newAddedshippingSurcharge);
 self.subTotal(newAddedsubTotal+self.subTotal());
```

```
 self.total(self.total()+newAddedsubTotal+newAddedshippingSurcharge);
 self.saveCartCookie();
 if (self.callbacks &&
self.callbacks.hasOwnProperty(ccConstants.ADD_ITEMS_SUCCESS_CB)
     && typeof self.callbacks[ccConstants.ADD_ITEMS_SUCCESS_CB] ===
'function') {
   self.callbacks[ccConstants.ADD_ITEMS_SUCCESS_CB](data);
 }
}
```

When calculating prices, sale and list prices are used. You can implement other scenarios based on your own requirements.

updateCartItemDataForLargeCart

The following code is a method to update the cart properties during a “multiple item-add” operation:

```
CartViewModel.prototype.updateCartItemDataForLargeCart = function(data)
{
      var self =this;
      var price = data.childSKUs[0].salePrice ?
data.childSKUs[0].salePrice :data.childSKUs[0].listPrice;
      self.numberOfItems(self.numberOfItems()+data.orderQuantity);
      self.events=[];
      self.shippingSurcharge(self.shippingSurcharge()
+data.shippingSurcharge);
      self.subTotal(price*data.orderQuantity+self.subTotal());
      self.total(self.total()+price+data.shippingSurcharge);
      self.saveCartCookie();
    };
```

You can implement other scenarios based on your own requirements. In the Quick Order widget, instead of using the existing default widget batching logic, you can invoke view model methods to add multiple items. You will need to modify the Quick Order widget /Purchase List widget to accommodate large carts.

Prevent Site Traffic Slowdowns

This section describes the landing page throttling feature.

To prevent traffic from slowing down a website, OCC allows servers to deflect requests. This prevents the servers from becoming overloaded with traffic. Note that this deflection does not remove your existing shoppers from the site, nor does it impede your site functionality.

If your service does exceed its limits during a traffic spike, new shoppers will be redirected to a temporary waiting room page. Existing shoppers may continue uninterrupted. The waiting room page displays a white page with a CSS-based spinner without text. Every 15 seconds, the shopper’s browser will check if the site is accepting new shoppers. When new shoppers are again accepted, the browser will redirect back to the original URL the shopper intended to visit. This URL is stored in a target query parameter as a URL encoded value.

By default, the waiting room page looks for a background image in /file/general/occ-site-busy.jpg. You can create your own branded waiting room experience by providing your own background image. Use the Media tab in the administration console to upload your branded image to `/file/general/occ-site-busy.jpg`. After you publish this image, it will be automatically used as your background for the waiting room page.

Improve performance with large numbers of addresses for profiles or accounts

When a Commerce environment includes a large number of profile or account addresses, performance can suffer on the storefront and in the administration interface when searching for and working with addresses.

This section describes API features that you can use to improve performance when working with a large number of profile or account addresses.

Filter endpoint results with query parameters

Several query parameters for Admin, Store, and Agent API endpoints let you search for specific addresses. In a Commerce environment includes a large number of profile or account addresses, these query parameters help you filter addresses that are returned in endpoint responses.

The Agent API `listAddresses`, and `searchProfiles` endpoints and the Store API `getAddresses` and `listProfileAddresses` endpoints, and the Admin API `listAddresses` endpoint all support these query parameters.

For details about the query parameters, supported by these endpoints, see the REST API for Oracle Commerce documentation. For details about how to use query parameters in a request, see REST API query parameters.

Exclude addresses from endpoint responses

If you want to exclude secondary addresses and shipping addresses from endpoint responses that would normally return them, use the `updateCloudConfiguration` endpoint in the Admin API to set the `excludeAddressList` property to `true`. By default, `excludeAddressList` is set to `false`.

The following Admin API endpoints are affected by setting the `excludeAddressList` property to `true`: `createOrganization`, `updateOrganization`, `listOrganizations`, `getOrganization`, `listProfiles`, `getProfile`, `updateProfile`, `getOrganizationRequest`, and `updateOrganizationRequest`.

The following Store API and Agent API endpoints are affected by setting the `excludeAddressList` property to `true`: `getOrganization`, `updateOrganization`, `listOrganizations`, `getProfile`, `updateProfile`, `getMember`, `listMembers`, `updateMember`, `createMember`, `getAddresses`, `deleteAddress`, `getPage`, `getCloudConfiguration`, and `getOrganizationRequest` (Agent API only).

Because address details are not included in a number of endpoint responses when you set `excludeAddressList` to `true`, widgets that retrieve addresses from the responses of endpoints listed above may not behave as expected and may require customization.

For example, the Agent Console `account-addresses.js` widget's `fetchAddressesSuccess()` method expects `secondaryAddresses` to be present. When `secondaryAddresses` are suppressed from the endpoint response, the widget throws an error.

- For account-based commerce shoppers, `checkout-address-book.js` should be modified to fetch the addresses based on the secondary addresses. Update the widget to retrieve addresses using the `/ccagent/v1/profiles/{id}/addresses` endpoint.
- For individual shoppers, addresses should be loaded using the `/ccagent/v1/addresses` endpoint.
- Make sure that `updatedShippingAddressBook` is not accessed without validating whether address data exists in the `cart-shipping-details.js` widget.

50

Improve Storefront Performance

This section provides information on ways you may be able to improve storefront performance by considering specific coding strategies during your development process.

We recommend that you review this information before your site goes live in production.

Optimize First Meaningful Paint

First Meaningful Paint is the time it takes you page's primary content to appear on the screen and is considered a primary metric for gauging how your customer perceives the loading experience.

Reports using web page test can give you metrics on this loading. To improve your page First Meaningful Paint timing, you should prioritize above the fold content, such as marketing banners and delay secondary calls for inventory or data and frequently retest this metric using web page test.

Lazy load images

Lazy loading of images helps pages to provide more content by recognizing which images will be presented to the shopper, and loading them first. These images are known as *in focus*. Lazy loading is not used for images that are already in focus. It is instead used for images that will be seen by the shopper only when scrolling, and do not need to be fully loaded during the initial page draw. Instead, the shopper might see an initial generic image, such as a blurred/transparent image of the product, or a spinner. These images are known as *out of focus*. When the image comes into view on the page, the required image is rendered.

Enable lazy loading

By default, lazy loading is disabled. To enable this feature, use the `loadImagesLazily` client configuration setting in the Admin REST API. For example:

```
PUT  /ccadmin/v1/merchant/clientConfiguration
{
 "loadImagesLazily" : true
}
```

You can also set the time that out of focus images are loaded. By using `delayBeforeLoadingOutOfFocusImages`, you can identify the number of seconds to wait before loading any remaining out of focus images. Using the Admin REST API, you could do the following to set a 10 second delay before the out of focus images are loaded:

```
{
"loadImagesLazily" : true,
"delayBeforeLoadingOutOfFocusImages" : 10
}
```

Once lazy loading is enabled, images rendered using the following image bindings will be lazily loaded:

- `ccResizeImage` - Used in the Product Listing widget.
- `productVariantImageSource` - Used in the Cart Summary and Shipping Options widgets.
- `imageSource` - Used in the Cart Summary widget.
- `productImageSource` - Used in the Scheduled Order and Related Products widgets.
- `Image` - This binding can be used to display simple images. Used in the Product Details widget to display images in the product's image carousel.

You can specify an initial place holder image that is displayed while the image is out of view from the shopper. When the shopper scrolls through the page, the initial place holder may be displayed briefly while the requested image is obtained and then rendered. To set an initial place holder, use the `initialSrc` attribute in one of the above image bindings. If the `initialSrc` attribute is not specified, then the image specified in site-specific `noimage` setting will be used as the initial placeholder image.

The following binding attributes can be used with the image bindings listed above:

- `lazyLoadingImageClass` - The default value for this attribute is `ccLazyLoad`. This binding defined the CSS class to use with the temporary image while the actual image is loaded. By default, this class reduces the opacity of the initial image to 0.1.
- `lazyLoadedImageClass` - The default value for this attribute is `ccLazyLoaded`. This binding defined the CSS class to use with the final loaded image.
- `lazyLoadingParentClass` - The default value for this attribute in `ccLazyload-background`. This attribute defines the CSS class to use with the parent element of the initial image. By default, the background color of this image is light gray.
- `initialSrc` - Specifies the initial image source that should be used while the main image remains out of focus. If the `initialSrc` attribute is not used, the default error image is displayed.
- `disableLazyImageLoading` - Disables lazy loading for the image.

Disable lazy image loading

You can disable lazy image loading for an individual image as follows:

```
<img data-bind = "image: {src: yourImageSource,
disableLazyImageLoading: true}, ...">
```

To disable lazy image loading for all images, set the `loadImagesLazily` client configuration value back to false.

Improve Storefront Performance for Large Carts

When a shopper creates an order by adding a large number of items to their shopping cart, performance can suffer, sometimes to the point where the shopper cannot complete or submit the order.

This document describes how to configure Commerce to improve performance for large-cart orders. This section includes the following topics:

- Understand large cart support
- Enable support for large carts
- Understand view model support for large carts
- Sample widgets and elements

Understand large cart support

From a performance perspective, a large cart is a shopper's cart that contains 200 items or more. Carts this large may experience performance issues. Shoppers typically create orders this large using the Commerce Quick Order widget.

You can enable large cart support by configuring the `CCStoreConfiguration.largeCart` property. This reduces the number of situations where large cart information is refreshed (price calls), improving overall performance. All cart information always refreshes during the checkout phase.

New endpoints also enable large cart batching during GET product/SKU calls.

Enable support for large carts

To enable deferred pricing, set this property in `application.js`:

```
CCStoreConfiguration.largeCart
```

By default, this property is not set.

To override deferred pricing, use this method:

```
CCStoreConfiguration.isLargeCartCCStoreConfiguration.isLargeCart
```

You do not need to do anything to enable the use of large-cart endpoints, but you can change the number of items in a cart that triggers large-cart endpoints by using these properties:

- `CCStoreConfiguration.batchSizeForProdAndSkuData` is a property that specifies the number of items that must be in a cart before Commerce automatically uses `listProductsForLargeCart` and `listSkusForLarge` Cart instead of `listSkus` and `listProducts`.
- `CCStoreConfiguration.threshholdSizeForStockStatusData` is a property that specifies the number of items that must be in a cart before Commerce Cloud automatically uses `listStockStatusesForLargeCart` instead of `getStockStatuses`.

Understand view model support for large carts

This section describes the Cart view model properties and methods that support deferred pricing calls and batching the Store API Products and SKUs endpoints.

`CartViewModel.hardPricing` is a property that triggers cart pricing when the `CCStoreConfiguration.largeCart` flag is set to true. See Sample widgets and elements for a code sample that uses this property to trigger pricing. The default value of this property is false.

`CartViewModel.userActionPending` is a property that you should set to true whenever you update the cart and the `largeCart` flag is set to true. Use this flag to display messages reminding the shopper to save the cart. You can also use this flag to display messages that remind customers that prices are approximate and they will have to click the save cart button to trigger pricing and see the correct prices.

`CartViewModel.updateItemPriceForLargeCart` is a method to calculate the item total with an approximate price when you defer pricing calls. This method calculates prices by using list prices and sale prices. You can override this method to accommodate manual pricing calculations.

Use the method `CartViewModel.updateItemShippingGroupRelationShipForLargeCart` to update shipping group relationships from the client side when you have deferred pricing calls.

Use `CartViewModel` to implement these functions to update cart data when you defer pricing calls:

- `CartViewModel.updateCartItemDataForLargeCart` is a method to update the cart total and subtotal list price and the sale price of an item added to the cart when you defer pricing calls. It also updates `numberOfItems` for the Cart Summary widget. You can override this method to accommodate manual pricing calculations.
- `CartViewModel.updateCartAfterLoginForLargeCart` is a method to update cart data after a registered shopper logs into their account.

`CartViewModel.priceCartBeforeRefreshInLargeCart` is a method to reprice the entire cart.

`CartViewModel.getBatchSizeForProdAndSkuData` is a method that returns the number of products or SKUs whose data you need to retrieve in each `listProduct` or `listSku` request. If this is set to 1, it indicates that all products or SKUs will be sent to `listProduct` or `listSku` at once. Any other positive value indicates the number of products/SKUs sent in one call to the `listProduct` or `listSku` endpoint. In this case, multiple calls to these endpoints retrieve data for all IDs.

`CartViewModel.getThreshholdSizeForStockStatusData` returns the threshold on the number of SKUs beyond which a POST equivalent of the stock call would trigger.

Sample widgets and elements

This section describes sample widget code that you can use to implement support for large carts.

Create an element to manually trigger pricing

The following sample button uses the `hardPricing` flag to let the shopper manually price the cart.

```
saveCartButtonHandler :function(){
 this.cart().hardPricing = true;
 this.cart().markDirty();
},
```

When pricing calls are deferred and you enable the large cart feature, all the item update/delete/add operations normally triggered from the Product Details widget, Shopping Cart widget, Header widget, Quick Order widget, the copy order operation from Order Details widget, and the Purchase list widget are handled from client side.

Shopping Cart Widget and Header widget implementation

You can remove line items from the Header mini cart and Shopping Cart widget. The following code sample is the basic implementation of client side handling for the removal of line items. Line items here correspond to shipping group relationship and the `removeShippingGroupRelationShip` method as the handler for the remove items event.

```
removeShippingGroupRelationShip :function(cartItem,shippingGroupRelationship)
{
  if(this.storeConfiguration.isLargeCart() === true){

    var quantityChange=(-1)*shippingGroupRelationship.updatableQuantity();
    var price=cartItem.productData().childSKUs[0].salePrice ?
cartItem.productData().childSKUs[0].salePrice :
cartItem.productData().childSKUs[0].listPrice;
    //remove the SGR
    cartItem.shippingGroupRelationships.remove(shippingGroupRelationship);
    if(cartItem.shippingGroupRelationships().length === 0){
      this.cart().items.remove(cartItem);
    }else{
      cartItem.quantity(data.quantity()+ quantityChange);
      cartItem.itemTotal(data.itemTotal()+price*quantityChange);
    }
    //update the number of items
    this.cart().numberOfItems(this.cart().numberOfItems()+quantityChange);
    this.cart().updateAllItemsArray();
    this.cart().subTotal(this.cart().subTotal()+price*quantityChange);
    this.cart().total(this.cart().total()+(price)*quantityChange);
    this.cart().userActionPending(true);
    this.cart().saveCartCookie();
    $.Topic(pubsub.topicNames.CART_REMOVE_SUCCESS).publishWith([{message:"suc
cess"}]);
    return true;
  }
}
```

Widget implementation for update

Line item quantity can be updated from the Shopping Cart widget. The following code is the basic implementation of client side handling for the update of line items. These line items

correspond to the shipping group relationship and `updateQuantity` methods that you can use as a handler for the update item event.

```
updateQuantity: function(data, event, id, shippingGroup) {

  if('click' === event.type || ('keypress' === event.type &&
event.keyCode === 13)) {
    // update the 'updatableQuantity' to cart-item.
    var cartItemTotal = 0;
    for(var index=0; index < data.shippingGroupRelationships().length;
index++) {
      cartItemTotal = parseInt(cartItemTotal) +
parseInt(data.shippingGroupRelationships()[index].updatableQuantity());
    }
    data.updatableQuantity(parseInt(cartItemTotal));

    if(data.updatableQuantity && data.updatableQuantity.isValid()) {
      if(this.storeConfiguration.isLargeCart() === true){
        var quantityChange=data.updatableQuantity()-data.quantity();
        var price=data.productData().childSKUs[0].salePrice ?
data.productData().childSKUs[0].salePrice :
data.productData().childSKUs[0].listPrice;
        this.cart().numberOfItems(this.cart().numberOfItems()
+quantityChange);
        data.quantity(data.updatableQuantity());
        data.itemTotal(data.itemTotal()+price*quantityChange);
        shippingGroup.price(shippingGroup.price()
+price*quantityChange);
        shippingGroup.quantity(shippingGroup.quantity()
+quantityChange);
        this.cart().updateAllItemsArray();
        this.cart().subTotal(this.cart().subTotal()
+price*quantityChange);
        this.cart().shippingSurcharge(this.cart().shippingSurcharge()
+data.shippingSurcharge*quantityChange);

this.cart().total(this.cart().total()+
(price+data.productData().shippingSurcharge)*quantityChange);
        this.cart().userActionPending(true);
        this.cart().saveCartCookie();
        return true;
      }
      $.Topic(pubsub.topicNames.CART_UPDATE_QUANTITY).publishWith(
      data.productData(),[{"message":"success", "commerceItemId":
data.commerceItemId, "shippingGroup": shippingGroup}]);

      var button = $('#' + id);
      button.focus();
      button.fadeOut();
    }
  } else {
    this.quantityFocus(data, event);
  }
```

```
  return true;
},
```

Product Details widget

Adding the product from Product Details widget will not trigger pricing. The added item updates in the cart view model first and after that, pricing triggers. The new methods are only called during nested view model logic when the large cart flag is turned on, just before system shunts the pricing. This ensures that the shopper has a view of the approximate prices of the items added to cart.

When calculating prices, sale and list prices are used. In the absence of pricing data, you can override `application.js` to update the specific item price using other special parameters such as bundled pricing, volume pricing, or tiered pricing from existing product data.

`updateItemPriceForLargeCart`

The following code is the method to update items during the “single item-add” operation of `updateItemPriceForLargeCart`.

```
CartViewModel.prototype.updateItemPriceForLargeCart = function(data,cartItem)
{
  var self =this;
  var price = data.childSKUs[0].salePrice ?
data.childSKUs[0].salePrice :data.childSKUs[0].listPrice;
  cartItem.itemTotal(cartItem.itemTotal()+price*data.orderQuantity);
};
```

`updateCartItemDataForLargeCart`

When calculating prices, sale and list prices are used. You can implement other scenarios based on your own requirements.

The following code shows the method to update the cart properties during the “single item-add” operation:

```
CartViewModel.prototype.updateCartItemDataForLargeCart = function(data){
  var self =this;
  var price = data.childSKUs[0].salePrice ?
data.childSKUs[0].salePrice :data.childSKUs[0].listPrice;
  self.numberOfItems(self.numberOfItems()+data.orderQuantity);
  self.events=[];
  self.shippingSurcharge(self.shippingSurcharge()+data.shippingSurcharge);
  self.subTotal(price*data.orderQuantity+self.subTotal());
  self.total(self.total()+price+data.shippingSurcharge);
  self.saveCartCookie();
};
```

You can extend this process by adding any other business related view model properties.

Adding multiple items at once (Quick order widget, Copy order, Purchase list)

Adding multiple items at once will not trigger pricing calls when large cart flag is true. Without modification, added items update in cart view model first and then, pricing triggers. The new methods are called during the nested view model logic when large cart flag is turned on, just

before system shunts the pricing. This ensures that the shopper has a view of the approximate prices of the item added to cart.

When calculating prices, sale and list prices are used. In the absence of pricing data, you can override these from `application.js` to update the item price taking into account such special parameters such as bundled pricing, volume pricing, or tiered pricing from existing product data.

updateCartItemsDataForLargeCart

The following code is a method to update the multiple item during “multiple item-add” operation:

```
    CartViewModel.prototype.updateCartItemsDataForLargeCart =
function(data){
  var self =this;

  //NOT REUSING updateCartItemDataForLargeCart to minimise mulple
notifications due to multiple update
  var
i,newQuantitiesAddedToCart=0,newAddedshippingSurcharge=0,newAddedsubTot
al=0;
  for(i=0;i<data.length;i++){
    var price = data[i].childSKUs[0].salePrice ?
data[i].childSKUs[0].salePrice :data[i].childSKUs[0].listPrice;
    newQuantitiesAddedToCart+=data[i].orderQuantity;
    newAddedshippingSurcharge+=data[i].shippingSurcharge;
    newAddedsubTotal+=price*data[i].orderQuantity;
  }
 self.numberOfItems(newQuantitiesAddedToCart+self.numberOfItems());
 self.events=[];
 self.shippingSurcharge(self.shippingSurcharge()
+newAddedshippingSurcharge);
 self.subTotal(newAddedsubTotal+self.subTotal());
 self.total(self.total()+newAddedsubTotal+newAddedshippingSurcharge);
 self.saveCartCookie();
 if (self.callbacks &&
self.callbacks.hasOwnProperty(ccConstants.ADD_ITEMS_SUCCESS_CB)
     && typeof self.callbacks[ccConstants.ADD_ITEMS_SUCCESS_CB] ===
'function') {
   self.callbacks[ccConstants.ADD_ITEMS_SUCCESS_CB](data);
 }
}
```

When calculating prices, sale and list prices are used. You can implement other scenarios based on your own requirements.

updateCartItemDataForLargeCart

The following code is a method to update the cart properties during a “multiple item-add” operation:

```
CartViewModel.prototype.updateCartItemDataForLargeCart = function(data)
{
      var self =this;
      var price = data.childSKUs[0].salePrice ?
```

```
data.childSKUs[0].salePrice :data.childSKUs[0].listPrice;
      self.numberOfItems(self.numberOfItems()+data.orderQuantity);
      self.events=[];
      self.shippingSurcharge(self.shippingSurcharge()
+data.shippingSurcharge);
      self.subTotal(price*data.orderQuantity+self.subTotal());
      self.total(self.total()+price+data.shippingSurcharge);
      self.saveCartCookie();
    };
```

You can implement other scenarios based on your own requirements. In the Quick Order widget, instead of using the existing default widget batching logic, you can invoke view model methods to add multiple items. You will need to modify the Quick Order widget /Purchase List widget to accommodate large carts.

Add a page level spinner

You can add a page level spinner along with a curtain on page change that can stop unnecessary clicks while page load is under process.

This will also provide a definitive indicator for the user that a new page is loading. The curtain also keeps the user from clicking other links while the page is loading.

To include this behavior, set the `enableSpinnerOnPageLoad` flag to true in `cc-store-configuration-1.0.js`.

Enable prioritized loading of Storefront page content

You may be able to speed the loading of some Storefront pages by enabling the prioritized loading option.

With Prioritized Loading, every region in a page displays in an ordered top-down fashion, that is, rendering the header section, then the body of the page, and then the footer. Widgets that need to be rendered are maintained in a queue and are displayed in order and one widget will not display until the previous one in the queue has displayed. This works on all pages including cached pages.

To enable the feature, set `enablePrioritizedLoading` to true, in `cc-store-configuration-1.0.js`. If the flag is set to true, prioritized loading is enabled. If this flag is not set, the page is loaded in the usual way and there would be no impact on any of the existing functionalities.

Avoid synchronous AJAX calls

Avoid synchronous AJAX calls since they may block other JavaScript processing while waiting for a response from the server.

Do not use these calls with `pageLoad`, since they may block the loading of the page.

Avoid hiding elements with CSS styling

It is best to avoid using the CSS styles to hide DOM elements.

If you use `display:none` on a , the contained markup always remains in the DOM with its data-bind attributes applied, even if the is not displayed and any changes to the view model keep updating the children. This may result in performance degradation. Only use this approach if the visible condition changes frequently. This avoids reconstructing the DOM for every change. In all other situations, use the `ko-if` binding if the binding does not construct the DOM when the condition results to false. When the condition result is true, child nodes are reconstructed and binding of the view model occurs.

Remove unused UI elements completely from layouts

You may be able to improve performance by removing any unused UI elements completely from layouts and avoid hiding unused UI elements in the template.

Hidden elements will still be loaded which can add to browser processing time. For more information, see Understand widgets.

Use viewport specific layouts for mobile

You may be able to improve mobile performance by using viewport specific layouts for mobile and tablet presentations of the storefront.

This approach limits and optimizes the content that you want to display for each viewport. Restricting image size may also improve performance.

In general, do not force the browser to render your full desktop layout on mobile devices. This can overload the user with too much content, appear unpolished, and produce slower page load times. Also, avoid over using responsive layouts. They can be helpful, but if overused, can lead to performance eroding DOM node duplication.

For more information, see Create a Widget.

Keep contents of header and footer regions consistent

You may be able to improve performance by keeping the contents of header and footer regions consistent across page layouts to prevent unnecessary downloading and re-rendering of these regions during each user page navigation.

You should consider expanding this approach to other page layouts that have similar content. For example, when a page loads, if any region remains the same (if it contains exactly the same widget instances), from the previous page, it will not be reloaded. For more information, see Understand widgets.

Limit DOM node creation

To potentially improve performance, consider limiting the number of DOM nodes you create, since, in Knockout, the creation and removal of DOM nodes can have a performance impact.

To see the number of DOM nodes you have created, use http://www.webpagetest.org and work to reduce the number of nodes to be fewer than 1500. For more information, see Understand widgets.

Use `ccLink` binding for quicker page loading

If you use the standard href link syntax, for example, using `<a href="/aboutUs">About Us<a/>` within a widget, this will cause the entire page to load.

For better efficiency, instead use the `ccLink` custom binding. Note that this approach will only improve performance in internal page-to-page navigation and will not improve external web link performance. For more information, see the Use ccLink binding for quicker page loading.

Resize images using the `ccResizeImage` binding

You may be able to improve performance by using the custom `ccResizeImage` binding to provide scaled images for display on the UI.

Important: You should not resize images on the client.

The following is an example of the use of `ccResizeImage` binding:

```
<img data-bind="ccResizeImage: {
    source: '/file/v2/products/ST_AntiqueWoodChair_full.jpg',
    xsmall: '80,80',
    medium: '120,120',
    size:'50,50',
    alt: 'Antique Wood Chair',
    errorSrc:'images/noImage.png',
    errorAlt:'No Image Found'}"></img>
```

In the example, the `ccResizeImage` binding returns an image of size 80x80, and 120x120 for xsmall and medium viewports, respectively. For all other viewports, it returns an image of size 50x50. For more information, see Resize Images.