Developing Open Storefront Framework Applications for Oracle CX Commerce

F38425-01
April 2021

Developing Open Storefront Framework Applications for Oracle CX Commerce,

F38425-01

Contents

5 Create Custom Widgets

6 Design Storefront Pages

7 Develop for Performance

8 Build Payment Integrations

1

About This Guide

This guide is intended for developers who want to use the Open Storefront Framework to create Oracle CX Commerce storefront applications.

It describes how to set up the developer environment and create and deploy applications. It also explains how to create Open Storefront Framework widgets.

Note: For this release, product documentation relating to the Open Storefront Framework is contained only in this guide; the other books in the Commerce documentation set continue to apply to the original storefront framework, Storefront Classic. If you are developing Open Storefront Framework applications, assume that anything presented in this guide supersedes similar content in the other Commerce books.

Prerequisites

Before you start building applications with the Open Storefront Framework, it is highly recommended that you become familiar with the concepts and tools on which the framework is based.

Ensure you have a working knowledge of the following:

- JavaScript
- Yarn package manager
- Node.js and Node Package Manager (npm)
- React
- Redux
- Redux Saga

Open Storefront Framework accessibility

Oracle's goal is to make its products, services, and supporting documentation accessible to all users, including users with disabilities.

The Open Storefront Framework includes a command-line interface that supports accessibility. For example:

- The contrast ratio is set by the user, either at the system level or in the terminal program used to access the remote machine. The product's display of foreground and background text is a function of the settings in a GUI-based terminal application.
- The application does not change accessibility features provided by the underlying operating system.

The command-line interface is described throughout this guide. For additional information about Oracle CX Commerce accessibility, see Accessibility Tasks.

2

Understand the Open Storefront Framework

The Open Storefront Framework (OSF) is the next generation framework from Oracle CX Commerce for building commerce storefronts.

OSF runs exclusively on Oracle Cloud Infrastructure (OCI), Oracle's next generation purpose-built, best-in-class cloud platform for enterprise applications. OSF is an important leap forward in the evolution of Commerce storefront technology, focused on minimizing coding required, maintaining business level control, leveraging the latest technology frameworks, and providing robust tooling for the modern developer.

OSF has a number of benefits, including:

- Enabling front-end developers to deliver faster, more responsive, richer user experiences.
- Being built in React, but allowing flexibility to develop in any front-end library without risk of lock-in.
- Leveraging the fully-featured Oracle CX Commerce solution with drag-and-drop experience tools, context-aware preview, personalization, content, merchandising, search, catalog, inventory, reporting, and more.
- Providing a clean separation between the presentation layer and the state model to support local development and testing.
- Allowing micro (not macro) updates to simplify how experiences are assembled and delivered with reusable, granular components that minimize the need to write new code.

OSF is based on industry-standard JavaScript tools such as Yarn, Node.js, React, and Redux, and supports popular design patterns through an extensible architecture that uses modular building blocks. OSF enables a range of development strategies, from working with the complete framework to using only the REST API to build headless applications.

This chapter provides an overview of the Open Storefront Framework and how to develop storefront applications using it that run on mobile devices and desktops. For additional best practice and how-to material on OSF, visit the Customer Connect forum.

Understand the OSF architecture

A key part of OSF is the use of a Node.js server to perform server-side rendering of pages, make REST API calls to the storefront server, and communicate with the shopper's browser.

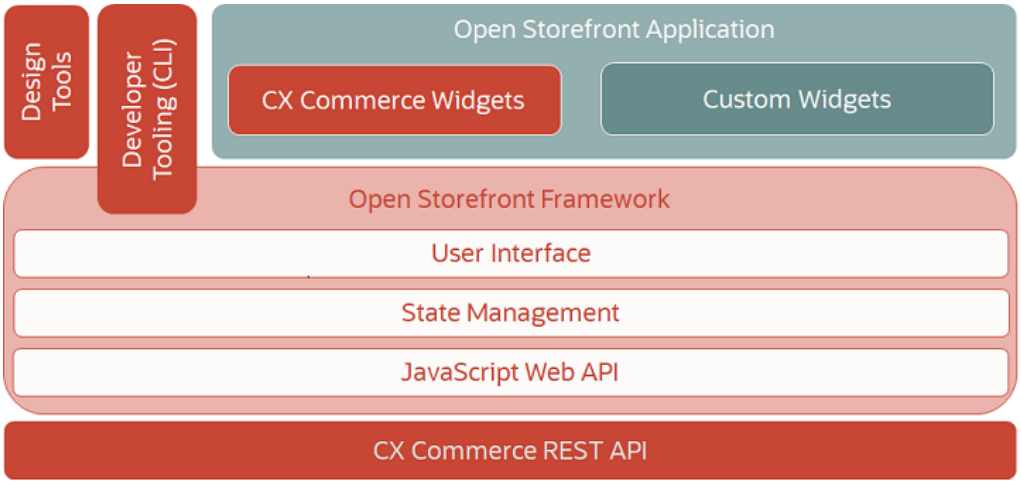
On a production system, the Node.js server runs in the same Oracle-hosted environment as the Commerce servers. When it renders a page, it sends it to the shopper's browser to be displayed.

For development purposes, Commerce supplies tools for setting up a workspace in your local environment. The workspace provides access to an npm (Node Package Manager) registry where you obtain OSF packages and their dependencies, as well as a Node.js server that runs JavaScript application code locally while communicating with Oracle-hosted Commerce servers. Storefront applications can be developed and run locally and then deployed to the Oracle environment for testing and production.

The Open Storefront Framework is designed to render pages quickly and to support rapid development. Its key features include:

- Optimized page loading – OSF supports single-page applications, in which pages are rendered on the Node.js server, and only the code needed for the specific features on a page is loaded. Pages load quickly and are SEO-friendly.
- Accelerated development – Development is done using industry-standard, developer-friendly tools, including Yarn and Node.js. The JavaScript APIs support best-practice design patterns.
- Layered, extensible architecture – The OSF architecture is based on modular building blocks, in which features are loaded as needed at runtime.
- Framework-neutral presentation layer – The storefront user interface is built using React by default, but you can substitute a different JavaScript framework.

The following diagram provides a high-level overview of the OSF architecture:



Understand OSF applications

You develop OSF storefront applications in a Yarn workspace on your local workstation, on which you are running a local Node.js instance.

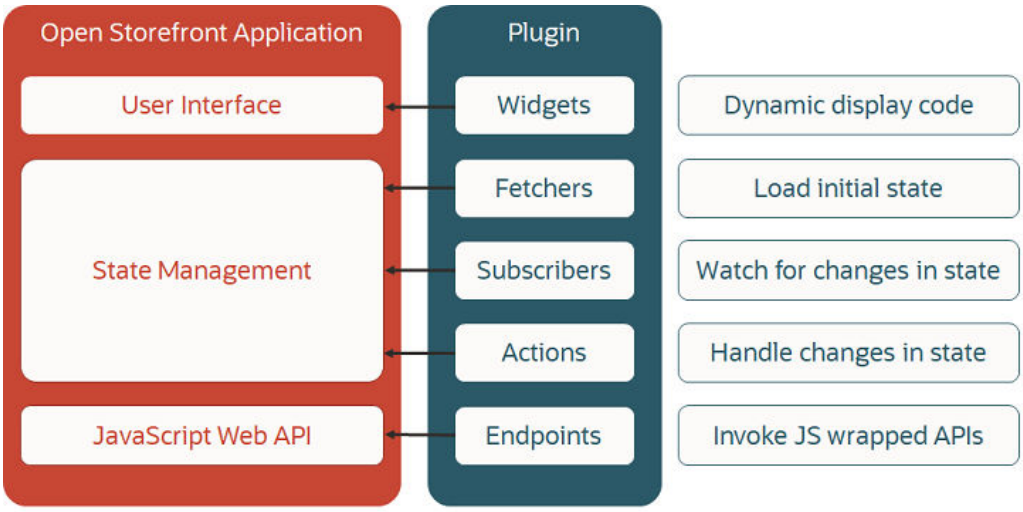
The packages in the workspace organize the JavaScript code plus the layouts and other assets used in the storefront presentation layer. The workspace includes native Yarn tools for managing the packages, as well as Oracle-supplied scripts and commands for configuring and building applications. Setting up the workspace is described in Set Up a Development Environment.

By default, the widgets for the storefront presentation are written as React components. Widgets are used in conjunction with other elements (plug-ins) to implement their functionality. These elements include:

- fetchers – Obtain data to generate the initial display of a widget.

- subscribers – Passively observe store events and trigger communication of these events to external services.
- actions – When invoked by widgets, send data that typically results in modification of the application's state.
- endpoints – Assemble the payloads for, and make calls to, Store REST API endpoints.

The following diagram illustrates the roles of the various types of OSF application elements:



See Create Custom Widgets for more information about these elements.

Once you have created your application, you can deploy it to one of the three server environments (test, staging, and production) that have been provided for you, as described in Develop and Deploy Applications.

Use a local workspace

You can create a storefront application and run the JavaScript components in a workspace on your local workstation.

A local workspace includes a Node.js server that runs JavaScript application code locally while communicating with Oracle-hosted Commerce servers. You can also use this workspace in local render mode, in which all design assets defined in the local workspace take precedence over those on the server. For example, if the local workspace has a home layout defined, it will be rendered instead of the home layout on the server. This approach provides you with the ability to do development in isolation, without your work being affected by code on the server or the work of other developers.

Use the workspaces on server environments

There are limitations to how much of the dynamic behavior that is provided on the server can be replicated when using local render mode.

For example, in local render mode, an individual developer can have only a single version of a layout, which is rendered in all cases. On the server, there can be multiple

versions of a given layout, with rules used at runtime to determine which one is rendered.

If you need access to this dynamic behavior, you can instead use a workspace in a server environment (typically the test environment) for rendering. Since this environment can be shared by multiple developers, you will need to coordinate with others to avoid conflicts. See Access local development applications on a Commerce server for information about OSF tools that support this.

Your organization is provided access to three server environments: test, staging, and production. Each environment includes administration, storefront, and Node.js servers. You use workspace with these environments to manage the lifecycle of the application code, and to access presentation content such as layouts that business users create using the Design tools in the administration interface.

You can interact with all three server environments in similar ways. You can deploy an application from a local workspace to a server environment's preview Node.js server, and publish a snapshot, including the workspace and design assets, to the live context. New development can then continue in the preview context and be published as desired.

Design assets

In an OSF local workspace, in addition to writing JavaScript code, you can create design assets such as layouts.

When you deploy an application to a server environment, the design assets are deployed along with the application code. As part of the deployment process, assets are extracted from the deployed workspace and used to update the assets in the server environment.

You also have the option of downloading the assets from the server environment's preview or live context to the local workspace for local development or committing to source control.

OSF registry

Oracle CX Commerce provides a read-only npm (Node Package Manager) registry that you can access to obtain OSF packages and their dependencies.

As part of creating a workspace, you configure Yarn so that when the `yarn install` command is executed, required packages are downloaded from the registry.

You can download packages and their dependencies at various points in the development cycle, such as when creating a local workspace and when deploying an application. You can also create new packages and declare dependencies in your local workspace, and these will be included when the application is deployed.

Command-line interface

Commerce provides a command-line interface (CLI) that you can use to manage your local workspace.

The CLI provides tools that you can use to create, configure, build, run, deploy, check deployment status and logs, and sync workspaces. Note that if you have multiple

workspaces, each workspace has its own matching version of the CLI (the version that was used to create the workspace).

3

Set Up a Development Environment

You can set up an environment for developing OSF applications on your local workstation.

This chapter describes how to set up an Oracle CX Commerce local storefront development environment using a provided accelerator template. It assumes that you have already read Understand the Open Storefront Framework and are familiar with the OSF architecture.

Once you have set up your environment, you can make a copy of the template and use it as a starting point for building your own application. This process is described in Develop and Deploy Applications.

Install required software

Before you can set up your development environment, you must install certain third-party software tools.

In addition to software supplied by Oracle, a storefront development environment requires the following:

- Node.js – Install the most recent LTS (long-term support) version of Node.js 12.x on your system. (Do not install a different major version number.) Node.js 12.x is available at:

  ```
  https://nodejs.org/dist/latest-v12.x
  ```

  The Node.js installation also includes npm.

- Yarn – Install the latest stable version of Yarn 1.x on your system. (Do not install Yarn 2.x) .Yarn 1.x is available at:

  ```
  https://legacy.yarnpkg.com/docs/install
  ```

Configure npm to access Commerce packages

The Commerce npm packages use the `@oracle-cx-commerce` scope and are hosted in an OSF registry that is separate from the primary npm registry.

To access this registry, enter the following command:

```
npm config set @oracle-cx-commerce:registry https://oracle-cx-commerce-repository.occa.ocs.oraclecloud.com
```

Set up a workspace

A workspace is a local directory hierarchy containing code and configuration for your storefront applications.

To create a workspace, you copy resources from one of the following:

- A prepackaged Commerce application that serves as an accelerator template.
- An existing deployed application.

This section describes how to create a workspace from an accelerator template. For information about creating a workspace from a deployed application, see Create a workspace from a deployed application.

Create a workspace from an accelerator template

You can use one of the following applications as a template for creating a workspace:

- `blank-store` – a minimal application that contains a single custom component
- `core-commerce-reference-store` – a more complete application that includes reference implementations of widgets and other components

To create the workspace, enter the following command:

```
npx @oracle-cx-commerce/cli-init create-workspace <workspace-dir> --
template <template-name>
```

<*workspace-dir*> is the absolute or relative path to the top-level directory of the workspace. This directory must currently not exist or else be empty.

For example:

```
npx @oracle-cx-commerce/cli-init create-workspace my-workspace --
template blank-store
```

This command uses the `cli-init` tool in the registry to run the Commerce `create-workspace` command. The command creates a subdirectory named `my-workspace` that contains the files and directories from the `blank-store` application.

The workspace top-level directory contains a number of files and directories, including the following:

- A `package.json` file that designates the workspace as a private package, specifies package dependencies, and defines several scripts.
- A `packages/apps` directory with a child directory containing the selected application (for example, `packages/apps/blank-store`).

You can also specify a different package name and Node.js module for your application when you create a workspace, by using the `--appPackageName` flag. For example:

```
npx @oracle-cx-commerce/cli-init create-workspace another-workspace --
template blank-store --appPackageName @my-module/my-storefront
```

This command creates a new workspace with an application named `my-storefront` in a module named `@my-module`.

Once you have created a workspace, you can begin creating your application by modifying or replacing the application code and configuration in the workspace.

Install package dependencies

Now install the package dependencies for the workspace. (These include the Commerce storefront framework and its dependencies.) To do this, go to the workspace directory and enter this command:

```
yarn install
```

This creates and populates the `node_modules` directory under your workspace directory. Notice that the `node_modules/@oracle-cx-commerce` subdirectory contains a symbolic link to the application directory (such as `blank-store`) in your workspace's `packages/apps` directory.

Note: When you run `yarn install`, you may see warnings similar to the following:

```
warning " > @oracle-cx-commerce/cli@1.0.0" has unmet peer dependency
"react@*".
```

You can safely ignore these warnings. For further information, see the following:

```
https://github.com/yarnpkg/yarn/issues/5347#issuecomment-463038189
```

Configure the workspace to access your server

Your local workspace requires access to a Commerce administration server environment to upload your application to and invoke REST API endpoints on. Typically, your organization is provided with three server environments (test, staging, and production). The process for configuring access to any of these servers is the same, differing only by which environment's URL you specify.

The first step is to generate an application key by registering the application on the specific server you want to access, as described in *Extending Oracle CX Commerce*. Then enter the following command to configure your workspace:

```
yarn occ configure --appKey --appServerAdmin <admin-server-URL>
```

where <*admin-server-URL*> is the URL of your Commerce administration interface. For example:

```
yarn occ configure --appKey --appServerAdmin https://myhost-admin.oc-
test.com
```

When you enter this command, you are prompted for an application key:

```
Please provide an appKey:
```

Enter the application key that was generated when you registered the application. (See Register applications.) When the command completes, it returns a listing of the various configuration settings for the workspace. Many of these settings are stored in the `config.js` file in the `.occ` directory of the workspace. You can update these settings with the `configure` command.

Build the application

To build the application in development mode, enter the following command:

```
yarn build
```

This invokes a script that runs Babel and Rollup to transpile all of the application's source code into code that is suitable for use in environments such as Node.js and web browsers.

Note: In development mode, Rollup remains active and occupies the shell window it is in, watching for changes in the source code to rebuild its output accordingly. If you want to build just once, you can safely use Ctrl-C to exit from Rollup once you see the "waiting for changes" message.

Upload the application to the administration server

To make your application's page layouts and components available in the administration interface, and to be able to preview and publish the application, you need to deploy the application to the administration server you configured access to. To deploy the application, enter this command:

```
yarn occ deploy
```

To move these assets from the preview context to the live context, log into the administration interface and publish the current change list.

Note that to see your deployed application's layouts and other assets you may need to change the active application to the one you uploaded. To do this, select the application from the dropdown list near the top right corner of the Design page.

Run the application

Enter the following command to run the application in development mode:

```
yarn start
```

This invokes a script that uses the nodemon utility to start a local Express web server. The server hosts the application, and is restarted whenever the application is updated.

You can now access the application's home page at `http://localhost:80`.

Note: The default port for the web server is 80, which may conflict with other web servers you have running locally. You can use a different port for the Express web server if necessary. For example, to use port 3000, enter the command:

```
yarn occ configure --httpPort 3000
```

This setting is stored in the workspace's `.occ/config.js` file.

Create a workspace from a deployed application

Support personnel who need to troubleshoot issues can create a workspace by copying a deployed application.

Note: This option is intended for support purposes only, and is not intended as a way to set up new developers on a project. Code and configuration should be maintained in source control, and developers should create their workspaces from these files rather than by copying a deployed application.

To create a workspace from a deployed application, enter the following command:

```
npx @oracle-cx-commerce/cli-init create-workspace <workspace-dir> --
deployId <deploymentId> --appServerAdmin <admin-server-URL> --appKey
```

where <*deploymentId*> is the deployment ID of the deployed application, <*admin-server-URL*> is the URL of your Commerce administration interface, and <*workspace-dir*> is the absolute or relative path to the top-level directory of the workspace. This directory must currently not exist or else be empty. You can find the deployment ID of the application by using the `deploy-status` command, as described in Monitor deployed applications.

For example:

```
npx @oracle-cx-commerce/cli-init create-workspace my-workspace –-
deployId 200004 –-appServerAdmin https://myhost-admin.oc-test.com --
appKey
```

You will be prompted to specify an application key for downloading the deployed application. After you enter the application key, Commerce creates a new local workspace named `my-workspace` on your workstation by copying the specified workspace from the server.

As an alternative, you can specify the deployed application by indicating the environment it is running in and whether to use the live or preview version. For example:

```
npx @oracle-cx-commerce/cli-init create-workspace my-workspace –-app-
name my-storefront –-appServerAdmin https://myhost-admin.oc-test.com –-
live --appKey
```

Upgrade the version of OSF in a workspace

Oracle CX Commerce provides tools for upgrading your version of OSF to a newer version.

The OSF version in the Commerce registry is updated regularly. When you first create a workspace from the registry, the workspace has the latest version of OSF.

If you have been using a workspace that has an earlier version of OSF, you can upgrade the workspace using the `upgrade` command. This command updates the version of OSF packages used throughout the workspace. Typically it is used to

upgrade to a higher-numbered version, though in certain cases it can also be used to downgrade to a lower-numbered version. Note that it does not update the version of non-OSF packages.

The primary way to use the `upgrade` command is to update the version of OSF in a workspace to the latest version allowed by your Commerce server, provided that both versions have the same major version number. To do this, invoke the command without any flags:

```
yarn occ upgrade
```

So, for example, if the current OSF version is 2.2.3 and the latest version the server supports is 2.6.7, this command updates the workspace to version 2.6.7.

If you're uncertain of the effect an `upgrade` command will have on your workspace, you can invoke the command with the `--dryRun` flag. This returns a list of the changes the command would make but does not actually make them. For example:

```
yarn occ upgrade --dryRun
```

If the changes are what you expect, you can then run the command without the `--dryRun` flag.

If the current workspace version of OSF is newer than the latest version the Commerce server supports, or if the workspace has a different major version number from the server, the `upgrade` command produces a warning and exits. You can override this behavior by using one or both of these flags:

- `--acceptDowngrade` – If the current workspace version of OSF is newer than the latest version the Commerce server supports, the workspace is downgraded to the latest version that the server supports.
- `--latest` – The workspace is updated to the latest version the server supports, even if the workspace and the server have different major version numbers. Be aware that updating to a different major version may introduce breaking changes.

To upgrade to a specific version of OSF, specify the version number explicitly in the command. For example:

```
yarn occ upgrade 2.13.6
```

If the specified version number is lower than the current workspace version, include the `--acceptDowngrade` flag.

Special options

The following flags may be useful in certain circumstances, but you should use them with caution:

- `--no-verifyOcc` – The command does not check the server version when determining what version of OSF to upgrade to.
- `--force` – The command attempts to continue rather than exiting when encountering errors (version incompatibilities, inability to contact the registry or

Commerce servers, etc.). Can be used only if a specific version of OSF is specified. For example:

```
yarn occ upgrade 2.0.0 --force
```

If you use either of these flags, be sure you understand the implications. It is a good idea to try the command with the `--dryRun` flag first to see the changes it will make before running it normally.

Workspace commands and scripts

The OSF workspace includes commands and scripts that you use to build, deploy, and manage the lifecycle of storefront applications.

This section describes the OSF commands and scripts and how they are used.

Commands

The OSF commands are found in the `node_modules/.bin` subdirectory of the workspace. The commands all begin with `occ` and accept flags for specifying options. For example, the following command deploys the specified application without performing a build:

```
occ deploy my-storefront --no-build
```

Some of the flags take arguments. For example, the following command specifies the URL of the administration server to deploy the application to:

```
occ deploy my-storefront –-appServerAdmin https://myhost-admin.oc-test.com
```

The following table summarizes the available commands:

Command	Description
`build`	Build an OSF application
`configure`	Update the configuration of a workspace
`configure-app`	Update the configuration of an application
`create-action`	Create an action for an application
`create-endpoint`	Create an endpoint from a Swagger document or a URL
`create-fetcher`	Create a fetcher for an application
`create-template`	Create a workspace template archive containing the application
`create-widget`	Create a widget plug-in for an application
`delete`	Delete an OSF application from the specified server
`deploy`	Deploy an OSF application
`deploy-log`	Query the deployment logs for an application
`deploy-status`	Query the deployment status of an application

Command	Description
`download`	Download the current deployment of an application to the workspace
`download-assets`	Download design assets from a server to the workspace
`list-apps`	List the applications on a server
`list-endpoints`	List all of the endpoints in a Swagger document
`output`	Generate the deployment files for the application
`redeploy`	Resend a deployment to a cluster
`serve`	Start a presentation server
`set-logging-options`	Set the logging options for a cluster
`upgrade`	Update the versions of OSF packages in a workspace
`upload-custom-typeahead-keywords`	Upload custom keywords for typeahead search
`upload-search-config`	Upload the search configuration for an application

You can also see a complete list of the `occ` commands by entering the following command:

```
occ –-help
```

To see a help page about the syntax of an individual command and the options it supports, enter the command with the `--help` flag. For example:

```
occ list-apps –-help
```

The response is similar to the following:

```
Usage: cli list-apps [options]

List the applications that are on the server.

Options:
  --json                  Output the results as raw JSON instead of
formatted text.
  -V, --version           output the version number
  --verbose               Provides verbose logging where available
  --no-verbose            Disables verbose logging
  --appKey [key]          With this option you will be prompted for an
application key (OAuth access token)
  --appServer <url>       Application server URL
  --appServerAdmin <url>  Application admin server URL
  --serverEnv <env>       Cloud Commerce server environment to use
  -h, --help              display help for command
```

Access the commands

Each workspace has its own version of the CLI. If you have multiple workspaces, you must use the version of the CLI associated with the workspace you are currently working in. To ensure that you access the correct version of the CLI, preface the commands with `yarn` and invoke the command from within the workspace.

For example, suppose you want to invoke the following command:

```
occ deploy my-storefront --no-build
```

Instead, enter the following from anywhere in the workspace you are working in:

```
yarn occ deploy my-storefront --no-build
```

When you call an `occ` command this way, Yarn uses Node.js module resolution to ensure that the correct version is executed.

Scripts

In addition to the `occ` commands, the workspace provides several scripts that can also be run from the command line. These scripts are defined in the `scripts` object of the `package.json` file in the top-level directory of the workspace. For example:

```
"scripts": {
    "stylelint": "stylelint **/*.css --ignore-disables",
    "eslint": "eslint .",
    "eslint:fix": "eslint --fix .",
    "prettier:fix": "prettier --config .prettierrc.js --write
\"{packages,qa}/**/*.js\"",
    "lint": "yarn eslint && yarn stylelint",
    "build:prod": "occ build --production",
    "build": "occ build --watch",
    "test:int": "jest -c jest.config.int.js",
    "test:int:debug": "node --inspect-brk node_modules/jest/bin/jest.js
-i -c jest.config.int.js",
    "test": "jest -c jest.config.js",
    "test:debug": "node --inspect-brk node_modules/jest/bin/jest.js -i
-c jest.config.js",
    "test:api": "jest -c jest.config.api.js",
    "test:api:debug": "node --inspect-brk node_modules/jest/bin/jest.js
-i -c jest.config.api.js",
    "perf": "jest -i -c jest.config.perf.js",
    "perf:lighthouse": "jest -i -c jest.config.perf.lighthouse.js",
    "perf:wpt": "jest -i -c jest.config.perf.wpt.js",
    "perf:debug": "node --inspect-brk node_modules/jest/bin/jest.js -i
-c jest.config.perf.js",
    "deploy": "occ deploy",
    "delete": "occ delete",
    "download": "occ download",
    "output": "occ output",
    "seed": "yarn deploy --reset --publish",
    "start:prod": "occ serve",
    "start": "nodemon --inspect node_modules/@oracle-cx-commerce/cli/
cli.js serve",
    "configure": "occ configure",
```

```
    "deploy-status": "occ deploy-status",
    "deploy-log": "occ deploy-log",
    "redeploy": "occ redeploy",
    "download-assets": "occ download-assets",
    "upload-search-config": "occ upload-search-config",
    "upload-custom-typeahead-keywords": "occ upload-custom-typeahead-
keywords",
    "list-apps": "occ list-apps"
}
```

Each entry in the `scripts` object consists of a Yarn command and the value it maps to. So, for example, suppose you enter this command:

```
yarn build:prod
```

This is equivalent to the following:

```
yarn occ build --production
```

As you can see, many of the scripts are equivalent to `occ` commands.

You can also display a complete list of the scripts, and select one to run, by entering the following command:

```
yarn run
```

The Yarn scripts use Node.js module resolution to locate the commands they call. This ensures that as long as a script is called from within a workspace, it will find the version of the command that is appropriate for that workspace.

4

Develop and Deploy Applications

Once you have set up an OSF development environment on your local workstation, you can build applications and deploy them to server environments.

This chapter provides an overview of working in OSF workspaces to develop and deploy storefront applications. More detailed information about creating storefront components is available in other chapters of this manual.

Create a new application by copying a template

You can create the basic directory structure for your application by copying an existing application directory from your workspace's `packages/apps` directory into a new subdirectory of `packages/apps`.

For example, if you have downloaded the `blank-store` template and you want your new application to be named `my-app`, go to your workspace's `packages/apps` directory and copy the entire `blank-store` directory and its subdirectories into a new directory named `my-app`. Then enter the following command:

```
yarn occ configure --appName my-app
```

This modifies the `config.js` file in the workspace's `.occ` directory to make `my-app` the default application for the workspace. Now the workspace commands and scripts such as `build`, `deploy`, and `start` will operate on this application instead of the template application.

At this point, the `my-app` application is identical to `blank-store`. You can now modify the `my-app` application to implement your own application.

Modify the application's `package.json` file

The top-level directory of an application (for example, `my-workspace/packages/apps/my-app`) contains an application-specific `package.json` file. This file is distinct from the `package.json` file in the top level of the workspace.

After creating a new application, you need to modify this `package.json` file. Change the values of the following settings:

- `name` – The name for your application (for example, `@my-apps/my-app`).
- `version` – The version number for your application (for example, `0.0.1`). You can use whatever numbering scheme you want, but it is a good idea to track the version using this field.
- `description` – The value of this setting is used to identify your application in the Design page application selector dropdown.

For example:

```
"name": "@my-apps/my-app",
"version": "0.0.1",
"private": true,
"description": "My storefront application",
```

In addition, change these settings in the file's `occ` object:

- `locales` – An array of the locales for which you are including translation strings for your application. You may want to set this to a single locale while you're getting started and update the setting later.
- `whiteListedUrls` – An array of the domains that the application calls out to. These domains are added to the list maintained by your Oracle CX Commerce environment. Calls to domains that are not on this list are blocked.

For example:

```
"occ": {
  "namespace": "occ.react",
  "locales": [
    "en",
    "de"
  ],
 "whiteListedUrls": [
    "http://www.example.com",
    "http://www.example2.com"
  ]
}
```

After you have updated the settings in this `package.json` file, run `yarn install` to create the symbolic link to your application directory.

At this point, if you build, upload, and start the application, you will see your application in your browser when you access the local web server. You can now proceed to make functional changes to the application by creating your own pages and components.

For information about how to build storefront components and include them in pages, see Create Custom Widgets and Design Storefront Pages.

Create components using command-line tools

The OSF command-line interface includes several commands that you can use to simplify the process of creating a widget and supporting components.

You can use these commands to create templates for building the following types of components:

- widgets
- actions
- fetchers
- endpoints

The commands create the files and folders that make up the structure of the components. Many of these files include comments that supply guidance about writing the code and supplying the configuration for the component. The guidance assumes you are writing a sample currency selector widget and the supporting components, but you can supply logic for any type of widget you want.

Create a widget

To create a widget, you use the `create-widget` command. For example:

```
yarn occ create-widget --name MyWidget

[cli] info: Creating Widget: MyWidget
[cli] info: Widget path: packages\apps\blank-
store\src\plugins\components
[cli] info: App locals - en,de
[cli] info: Creating new widget folder packages\apps\blank-
store\src\plugins\components\my-widget
[cli] info: Writing template files to packages\apps\blank-
store\src\plugins\components\my-widget
[cli] info:      config\index.js
[cli] info:      config\locales\en.js
[cli] info:      config\locales\de.js
[cli] info:      index.js
[cli] info:      locales\en.js
[cli] info:      locales\de.js
[cli] info:      meta.js
[cli] info:      README.md
[cli] info:      styles.css
[cli] info:      __test__\my-widget-widget.spec.js
[cli] info:      __test__\my-widget.spec.js
[cli] info: Updating exports for files:
[cli] info:     packages\apps\blank-
store\src\plugins\components\index.js
[cli] info:     packages\apps\blank-store\src\plugins\components\meta.js
```

This command creates a widget named `MyWidget` in the default application in the workspace. For example, if the default application is named `my-app`, the widget files are created in `packages/apps/blank-store/src/plugins/components/my-widget`.

To create a widget in an application other than the default application, you can specify the application name in the command. For example:

```
yarn occ create-widget my-other-app --name AnotherWidget
```

Create an action

To create an action, you use the `create-action` command. For example:

```
yarn occ create-action --name myAction

[cli] info: Creating action: myAction
[cli] info:      Action exported as _myAction
[cli] info:      Action's Path: src/plugins/actions
```

```
[cli] info: Creating new action folder packages\apps\blank-
store\src\plugins\actions\my-action
[cli] info: Writing template files to packages\apps\blank-
store\src\plugins\actions\my-action
[cli] info:      index.js
[cli] info:      meta.js
[cli] info:      schema\input.json
[cli] info:      __test__\index.spec.js
[cli] info: Updating exports for files:
[cli] info:     packages\apps\blank-store\src\plugins\actions\index.js
[cli] info:     packages\apps\blank-store\src\plugins\actions\meta.js
```

When you create an action, you can also create a reducer that the action invokes:

```
yarn occ create-action --name myAction --reducer
```

The reducer is created in the action's `index.js` file.

You can also specify an endpoint that the action invokes:

```
yarn occ create-action --name myAction --endpoint myEndPoint
```

Note that this command doesn't create the endpoint, it just specifies the name of the endpoint required by the action. You can either specify an endpoint that has already been created, or write one afterward with the name you specified. (If you use `--endpoint` but omit the endpoint name, the name defaults to the name of the action.)

Create a fetcher

To create a fetcher, you use the `create-fetcher` command. For example, the following command create a fetcher for the `HelloWorld` widget:

```
yarn occ create-fetcher --name myFetcher --endpoint getOrder --selector
mySelector --forComponent HelloWorld
```

Note that the specified widget must already exist. The specified endpoint and selector can already exist, or you can write ones afterward with the specified names.

You can create a global fetcher by using the `--global` flag rather than the `--forComponent` flag. For example:

```
yarn occ create-fetcher --name myFetcher --endpoint getSite --selector
mySelector --global
```

Create an endpoint

To create an endpoint, you use the `create-endpoint` command. The version of the command that you use depends on whether your organization maintains a Swagger-based document containing a JSON representation of the REST API endpoints that can be called from widget code.

To create an endpoint from a Swagger document, you use the `--swaggerUrl` flag. For example:

```
yarn occ create-endpoint --directoryName swagger-endpoints --swaggerUrl
http://www.example.com/catalogApi --endpoints getOrder,putOrder
```

The `--directoryName` flag specifies the subdirectory (relative to the application's `\src\plugins\endpoints` directory) to create the endpoint in. The command creates the specified directory; it cannot already exist.

The `--endpoints` flag is used to specify a comma-separated list of the IDs of the REST endpoints in the JSON document. There are two ways to specify these IDs:

- If the JSON document includes an `operationId` value for each endpoint, use these values to specify the endpoints. The example above illustrates using these values.
- If the JSON document does not include `operationId` values, use the zero-based index of the ordering of the REST endpoints in the document. For example, to specify the first and fifth endpoints in the file, you would use `--endpoints 0,4`.

Note that you can display a list of all the endpoints in a Swagger document by using the `list-endpoints` command. For example:

```
yarn occ list-endpoints --swaggerUrl http://www.example.com/catalogApi
```

To create an endpoint without a Swagger document, you use the `--url` flag to specify the URL of an individual REST endpoint, and the `--verb` flag to specify the HTTP verb. For example:

```
yarn occ create-endpoint --directoryName other-endpoints --url http://
www.example.com/orders --verb GET
```

Deploy the application

In OSF, much of the work of serving the presentation layer is delegated to a Node.js server. Widget code, for example, is part of the Node.js application.

In a development environment, the Node.js application runs in your local workspace, and communicates with Oracle-hosted Commerce servers. The JavaScript application can then be uploaded to a Node.js instance on a Commerce server environment, where it runs alongside administration and storefront instances. Each environment can run separate preview and live versions of the application.

Typically, developers have access to three server environments: test, staging, and production (see below). To deploy application changes to a server environment, you will need an application key specifically for that environment. You should be assigned a unique application key for each environment you have access to. Other developers may require access to different environments. For security purposes, you should avoid sharing your application keys with other developers or checking them into your source-control system.

When an application is deployed, the workspace itself is uploaded to the administration server. You can use the CLI to prepare the workspace for deployment,

upload it, and track its status. See Workspace commands and scripts for information about the command-line tools available for managing storefront applications.

Because the entire workspace is deployed, it is possible for other developers to create their own workspaces from the deployed application, although it is recommended that they create their workspaces from source control. See Create a workspace from a deployed application.

Server environments

This section describes the test, staging, and production server environments.

Test environment

The test environment is intended to be used when creating an application. Developers create the initial set of components and layouts in their local workspaces and deploy them to the test environment as needed.

Staging environment

The staging environment is intended to be used as a pre-production environment for running the developed application. The application running in this environment should be similar to the production version in terms of business data and integrations. Developers and business users can test and refine the application in a simulated production environment.

Production environment

The production environment is where the public-facing version of the application runs. Major changes should not be rolled out to the production environment without first being tested in the test or staging environment. Access to the production environment should be strictly controlled to allow only a limited group of developers to update the production version of the application.

Deployment status

OSF provides commands for accessing deployment logs and monitoring the deployment status of applications. See Monitor deployed applications for more information.

Promote the application to the live context

After you deploy an application to the preview context on a Commerce server environment, its assets appear in the publishing list along with all the other assets that have pending updates.

To complete an application deployment and update the live instance, you perform a publishing operation in the administration interface. You should view the publishing list carefully and explicitly specify whether the preview application, the design assets, or both should be published by checking or unchecking the corresponding items in the list.

Sync assets on a Commerce server with a local workspace

Oracle CX Commerce provides tools for keeping assets on server environments in sync with local workspaces.

Your development team should maintain application code and configuration in a source control system. Developers can access the source control system on their workstations, build and modify applications in their local workspaces, and then deploy the applications to Commerce server environments. Working this way helps ensure that the assets on the server match the ones in the source control system.

However, not all assets are necessarily created by developers in local workspaces. The Design page in the administration interface provides tools that designers can use to create assets such as layouts and containers. In order to keep these assets up to date in the source control system, the OSF command-line interface includes options for syncing assets between a server environment and a local workspace. This section describes how to use these commands. The assets they affect include layouts (pages), widgets and widget instances, containers (widgets that can contain other widgets), text snippets, and slot instances.

Note that you should be careful when you use these options to ensure that you get the results you expect. Depending on which commands you use and the flags you specify, the logic associated with deleting or overwriting existing assets on the target system may differ. Be sure you understand what the effect of a given command will be before you execute it, to ensure that you do not lose any changes. To help avoid conflicts, it is a good idea for each developer working on an application to create a local development version, as described in Access local development applications on a Commerce server.

Upload assets from a local workspace to a server

To copy assets from a local workspace to a server environment, you use the `deploy` command. For example:

```
yarn occ deploy blank-store
```

When you run this command, Commerce creates only new assets on the server. Any asset that already exists on the server will not be overwritten by an asset with the same name from the local workspace.

Run the `deploy` command with the `--reset` flag to make the workspace on the administrative server match the local workspace. The server will contain just the assets that were uploaded from the local workspace. For example:

```
yarn occ deploy blank-store --reset
```

Note that with this option, any changes that have been made to assets on the server that are not also on the local workspace will be lost. To avoid losing these changes, you should download these assets to the workspace first, and then deploy the application.

Download assets from a server to a local workspace

The CLI includes a `download-assets` command for copying assets from a server environment to a local workspace. For example:

```
yarn occ download-assets blank-store
```

Any asset in the workspace whose name matches an asset downloaded from the server is overwritten; other assets in the workspace remain as is. To avoid losing

changes that you have made to assets in your workspace, you should first commit these changes to your source control system before downloading assets from the server, and then resolve any conflicts afterward.

Run the `download-assets` command with the `--reset` flag to make the local workspace match the administration server. The local workspace will contain just the assets that were downloaded from the server. For example:

```
yarn occ download-assets blank-store --reset
```

Note that with this option, any changes that have been made to assets in the workspace that are not also on the server will be lost. To avoid losing changes that you have made in your workspace, you should first commit these changes to your source control system before downloading assets from the server, and then resolve any conflicts afterward. Or you can first deploy the changes to the server, and then download the assets from there.

Access local development applications on a Commerce server

OSF supports isolating different versions of assets on a Commerce server instance, allowing developers to work on local versions of an application independently while taking advantage of the features provided by remote rendering.

The key to sharing a Commerce server instance is for each developer to supply a local development application name for his or her version of the application. The local development application name should be different for each developer, and each local development name should be different from the hosted application name (which is the same for every developer). For example, the name for the hosted application might be `storefront-app`, while the local development names could be `fsmith-app`, `jjones-app`, and so on.

The location of the source code for the application is the same in each local workspace, and reflects the hosted application name (for example, `/packages/apps/storefront-app`).

Use the local development application name

You can use the `--localDevAppName` flag to specify the local development application name to use when deploying and accessing the assets that are specific to the local workspace. For example, you could enter:

```
yarn occ deploy storefront-app --localDevAppName fsmith-app
```

This command deploys the local version of the application to the Commerce server as `fsmith-app`.

The following commands all accept the `--localDevAppName` flag:

```
create-template, delete, deploy, deploy-log, deploy-status, download,
download-assets, serve, upload-demo-search-keywords, upload-search-
config
```

A local development name can also be configured persistently for an application so that it does not have to be included explicitly in commands. For example:

```
yarn occ configure-app storefront-app --localDevAppName fsmith-app
```

The setting is stored in the application configuration file (`/packages/apps/storefront-app/.occ/config.js`). For example:

```
module.exports = {
  "localDevAppName": "fsmith-app"
};
```

Override the local development application name

A developer can use the `--no-localDevAppName` flag to remove the local development name from the application configuration. For example, the following command deletes the setting from the application's `.occ/config.js` file:

```
yarn occ configure-app blank-store --no-localDevAppName
```

The `--no-localDevAppName` flag can also be used with the commands that accept `--localDevAppName` to override the local development application name without removing the name from the application configuration. For example:

```
yarn occ deploy storefront-app --no-localDevAppName
```

This command deploys the application to the server as `storefront-app`, but leaves the local development application name setting in the application configuration. For example, a developer might use this to check the deployment status of the hosted application in the preview or live context.

Render a local development application in a local workspace

Local development applications deployed to a Commerce server cannot access the Node.js instances on the server, and therefore cannot be hosted in the server's preview or live context. Only the hosted version of the application can be installed in these environments.

To view and test a local application while accessing the application-specific asset versions, a local workspace's Node.js instance must be used to render the design assets that are on the Commerce server. So, for example, the following commands might be used to specify the local application name, deploy the application to the Commerce server, and then run the application locally while accessing assets on the server:

```
yarn occ configure-app blank-store --localDevAppName fsmith-app
yarn occ deploy blank-store
yarn occ serve blank-store --dsAssetMode remote
```

Access local application assets in the administration interface

Although a local development application that is deployed to a Commerce server cannot be rendered on the server, its assets can be accessed in the administration

interface. Multiple developers may have their own local development versions of the same application, and to avoid conflicts, each version has a separate entry in the application selector dropdown on the Design page. To distinguish the different versions, the description that appears for each application in the dropdown is prefixed by the local development name of the application.

Update a hosted application

Each local application can have a separate version of a specific asset. Even when a local asset is deployed to a Commerce server and published, each version of the application can have a different version of that asset.

This means that developers must ensure that they do not overwrite each other's changes when they commit them to version control. Code must be merged carefully to resolve conflicts. Keep in mind that in addition to changes made in local workspaces, developers and designers may also makes changes on the server that must be managed as well, as discussed in Sync assets on a Commerce server with a local workspace.

To minimize risks, it is a good idea to use a continuous integration server that runs the code that has been committed to version control, and deploy changes to the hosted application from there.

Deploy a local development application as the hosted application

As discussed above, a local development application's assets can be deployed to a Commerce server, but for the application to run it must be rendered in a local workspace.

In some cases, a developer may want to deploy a local version of an application to the server, and have it be rendered there so other developers can access the running application. Before doing this, make sure that all of the assets from the "true" hosted version have been checked into source control so that the version can easily be re-created later; there should be no pending changes. Then deploy the local application using the `--no-localDevAppName` flag, so that it becomes the hosted application. For example:

```
yarn occ configure-app blank-store --reset --no-localDevAppName
```

Note that the `--reset` flag forces all of the assets from the local application to be deployed to the server, overwriting the assets from the hosted version.

Monitor deployed applications

OSF provides tools that you can use for monitoring deployed applications by querying the deployment controllers running on the storefront server.

A controller instance is a Node.js application that manages the availability of a new deployment. The controller receives the deployment from the administration server, installs it, and then spawns application instances from it. The controller then switches incoming HTTP traffic to the new instances and takes the previous deployment offline. Each deployment is assigned a unique ID that can be used to identify it for monitoring.

The controller produces logs related to the process of accepting, installing, starting, and switching deployments. These logs are particularly helpful in cases where the application does not install successfully. For example, if the deployment status

indicates that a deployment is not active on a controller, the logs for the controller will contain information that can help you determine why.

A cluster is a group of preview and live controllers. A cluster has one or more controllers configured for the preview context and one or more controllers configured for the live context.

When a deployment is uploaded, the administration server pushes it to the preview controllers in the cluster. When each preview controller receives the new application, it makes the previous preview application inactive. The controller deletes the inactive application's files, sets up the new application, starts it, and then makes it active by beginning to route traffic to it. Similarly, when a preview application is published, it is pushed from the preview context to the live context controllers, which inactivate the previous live application and activate the newly published application.

View deployment status

You can use the `deploy-status` command to return status information about a deployed application. The command supports a variety of options for querying about the status of deployments. These include options that return:

- The status of the preview and live deployments for a specified application.
- The status of a deployment specified by ID.
- The status of the deployments in a cluster.

Different information is returned for each of these queries. The responses are described below.

Query by application

To see the status of the preview and live deployments for an application, specify the name of the application in the command like this:

```
yarn occ deploy-status <app-name>
```

If you do not supply the application name, it defaults to the value of the `appName` setting in the workspace's `.occ/config.js` file. You can change this setting using the `configure` command or edit the setting in the file directly.

The `deploy-status` command returns information about the preview and live deployments that have been received by the administration server and their status on their respective controllers. For example:

```
yarn occ deploy-status my-app

[cli] info: Getting status for application my-app
[cli] info: Deployment 300322 found in preview.
[cli] info: - Deployment: 300322 -
[cli] info:   application id: my-app
[cli] info:   creation date:  2020-06-10T03:50:27.087Z
[cli] info:   tags:           username:fsmith
[cli] info: Deployment 300322 is fully installed in preview.
[cli] info: Deployment 300322 found in live.
[cli] info: - Deployment: 300322 -
[cli] info:   application id: my-app
[cli] info:   creation date:  2020-06-10T03:50:27.087Z
```

```
[cli] info:   tags:           username:fsmith
[cli] info: Deployment 300322 is fully installed in live.
```

The response includes the deployment ID for each deployment. In this example, the same deployment (ID = 300322) is present in both the preview and live contexts, but that may not always be the case.

If the response indicates that a deployment is fully installed in the preview or live context, this means it is currently running on that context's controllers. However, if you see a message like the following, the deployment was received by the administration server but has failed to install and run successfully:

```
[cli] info: Deployment 300322 found in preview.
. . .
[cli] info: Deployment 300322 is not fully installed in preview.
```

See Redeploy an application for information for information about what to do if this occurs.

Query by deployment ID

To see the status of a single deployment specified by deployment ID, use the `--deployment` option:

```
yarn occ deploy-status --deployment <deployID>
```

The deployment can be one currently running in the preview or live context, or any earlier deployment for which Commerce still has a record. If there is a record of the specified deployment on the administration server, information like the following is returned:

```
[cli] info: Getting status for deployment 300322
[cli] info: - Deployment: 300322 -
[cli] info:   application id: my-app
[cli] info:   creation date:  2020-06-10T03:50:27.087Z
[cli] info:   tags:           username:fsmith
```

If the specified deployment is the current preview or live deployment, information like the following is also returned:

```
[cli] info: Deployment 300322 found in preview.
[cli] info: Deployment 300322 is fully installed in preview.
[cli] info: Deployment 300322 found in live.
[cli] info: Deployment 300322 is fully installed in live.
```

Query by cluster

You can query deployment status by cluster to return information about the deployments running on the controllers in the cluster, and about earlier deployments that are now inactive. The information returned reflects only deployments that have been successfully installed, not deployments that were received by the administration server but were not installed successfully.

To return the status of the deployments in a cluster, use the `--cluster` option:

```
yarn occ deploy-status --cluster <clusterID>
```

If <*clusterID*> is omitted, its value defaults to `storefront`. This is the only value currently supported.

The response is returned in JSON format, as in the following example:

```
[cli] info: Getting status for cluster storefront
[cli] info: {
  "preview": [
    {
      "registry": "https://myhost-npmregistry.oc-test.com",
      "host": "https://myhost-previewsf2.oc-test.com/gcommerce-
osfcontroller1",
      "containerId": "b593ddef66b6",
      "controllerVersion": "1.8.0",
      "status": [
        {
          "appCapacity": 0,
          "instanceFailures": 0,
          "appName": "my-app",
          "state": "STOPPED",
          "deployId": "300321"
        },
        {
          "appCapacity": 1,
          "instanceFailures": 0,
          "appName": "my-app",
          "state": "ACTIVE",
          "deployId": "300322"
        }
      ]
    },
    {
      "registry": "https://myhost-npmregistry.oc-test.com",
      "host": "https://myhost-previewsf2.oc-test.com/gcommerce-
osfcontroller2",
      "containerId": "50c631159649",
      "controllerVersion": "1.8.0",
      "status": [
        {
          "appCapacity": 0,
          "instanceFailures": 0,
          "appName": "my-app",
          "state": "STOPPED",
          "deployId": "300321"
        },
        {
          "appCapacity": 1,
          "instanceFailures": 0,
          "appName": "my-app",
          "state": "ACTIVE",
```

```
          "deployId": "300322"
        }
      ]
    }
  ],
  "deployments": [
    {
      "metadata": "{\"appName\":\"my-app\",\"appDir\":\"packages/apps/
my-app\",\"tags\":[\"username:fsmith\"]}",
      "repositoryId": "300322",
      "clusterId": "storefront",
      "id": "300322",
      "applicationId": "my-app",
      "creationDate": "2020-06-10T03:50:27.087Z"
    },
    {
      "metadata": "{\"appName\":\"my-app\",\"appDir\":\"packages/apps/
my-app\",\"tags\":[\"username:fsmith\"]}",
      "repositoryId": "300321",
      "clusterId": "storefront",
      "id": "300321",
      "applicationId": "my-app",
      "creationDate": "2020-06-10T03:48:29.049Z"
    }
  ],
  "links": [
    {
      "rel": "self",
      "href": "https://myhost-admin.oc-test.com/ccadmin/v1/
deploymentEnvironment/storefront/status"
    }
  ],
  "live": [
    {
      "registry": "https://myhost-npmregistry.oc-test.com",
      "host": "https://myhost-livesf2.oc-test.com/gcommerce-
osfcontroller1",
      "containerId": "eb262e366c7f",
      "controllerVersion": "1.8.0",
      "status": [
        {
          "appCapacity": 0,
          "instanceFailures": 0,
          "appName": "my-app",
          "state": "STOPPED",
          "deployId": "300321"
        },
        {
          "appCapacity": 1,
          "instanceFailures": 0,
          "appName": "my-app",
          "state": "ACTIVE",
          "deployId": "300322"
        }
      ]
```

```
    }
  ]
}
```

View deployment logs

You can use the `deploy-log` command to return controller logs for a deployed application. The command supports a variety of ways to specify a cluster to return logs for. These include supplying one of the following:

- The name of the application deployed in the cluster.
- The cluster ID.
- The ID of the deployment that the cluster is associated with.

You can specify the name of the application in the command like this:

```
yarn occ deploy-log <app-name>
```

If you do not supply the application name, it defaults to the value of the `appName` setting in the workspace's `.occ/config.js` file. You can change this setting using the `configure` command or edit the setting in the file directly.

The command determines the current preview deployment for the specified application, determines what cluster that deployment was installed on, and then queries that cluster for its logs related to that deployment. This enables you to retrieve the logs even if you do not know the deployment ID.

You can specify the cluster by using the `--cluster` option:

```
yarn occ deploy-log --cluster <clusterID>
```

Specifying the cluster ID overrides the application name. Note that if you include the `--cluster` option but omit the cluster ID, the cluster ID defaults to `storefront`. This is the only value currently supported.

You can specify the deployment by using the `--deployment` option:

```
yarn occ deploy-log --deployment <deployID>
```

The deployment ID is required. This option overrides the application name and the `--cluster` option if they are specified.

Return the system log

To see the system log for the cluster rather than the application log, use the `--system` option. For example:

```
yarn occ deploy-log --system

[cli] info: Connecting to https://myhost-admin.oc-test.com
[cli] info: Getting logs for application my-app
[cli] info: logs for http://10.74.98.250:4080
2020-05-19 17:30:15 - [storefront-60e98a0a1f6e-controller-104] info:
[1/5] Validating package.json...
```

```
2020-05-19 17:30:15 - [storefront-60e98a0a1f6e-controller-104] info:
[2/5] Resolving packages...
2020-05-19 17:30:15 - [storefront-60e98a0a1f6e-controller-104] info:
[3/5] Fetching packages...
2020-05-19 17:30:15 - [storefront-60e98a0a1f6e-controller-104] info:
[4/5] Linking dependencies...
2020-05-19 17:30:15 - [storefront-60e98a0a1f6e-controller-104] warn:
warning " > @oracle-cx-commerce/cli@1.8.0" has unmet peer dependency
"@oracle-cx-commerce/rollup-config@*".
2020-05-19 17:30:15 - [storefront-60e98a0a1f6e-controller-104] warn:
warning " > @oracle-cx-commerce/cli@1.8.0" has unmet peer dependency
"builtin-modules@*".
...
2020-05-19 17:30:17 - [storefront-60e98a0a1f6e-controller-104] info:
[5/5] Building fresh packages...
...
```

Redeploy an application

If you detect issues with an existing deployment or if a new deployment fails, you can use the `redeploy` command to resend the deployment to the cluster. You can use this command to retry a failed installation or to synchronize the controllers on the same deployment. The deployment is determined based on the application that was last deployed to the cluster.

You can use command options to specify the server environment and context. For example, use the `--live` option to specify the live context:

```
yarn occ redeploy --live
```

By default, the deployment is pushed only to the controllers that are not already running it. To force controllers to reinstall the deployment regardless of whether they are currently running it already, use the `--live` option. For example:

```
yarn occ redeploy --force
```

Tag deployments

To help you to track your deployments, the deploy command includes a `--tag` option that you can use to add custom metadata to deployments to distinguish them from one another. This option enables you to label your deployments in a way that is meaningful within your organization. For example, the following command deploys an application and adds tags that specify the build number (using whatever convention your team prefers) as well as a feature in the build:

```
yarn deploy my-app --tag build1.2.3 --tag featureX
```

The following example illustrates how the tags are returned by the `deploy-status` command:

```
yarn deploy-status --deployment 2300001

[cli] info: Getting status for deployment 2300001
```

```
[cli] info: - Deployment: 2300001 -
[cli] info:   application id: my-app
[cli] info:   creation date:  2020-05-07T22:32:26.754Z
[cli] info:   tags:           build1.2.3, featureX, username:fsmith
[cli] info: Deployment 2300001 found in preview.
[cli] info: Deployment 2300001 found in live.
```

Note that the `username` tag is added automatically. It reflects the username on the machine where the deployment was created.

5

Create Custom Widgets

Widgets are added to the layouts that provide the user interface for your storefront.

Note that the widgets used in the Open Storefront Framework (OSF) are implemented differently from the widgets used in previous Oracle CX Commerce versions. This section describes OSF-based widgets. For information on widgets used in Oracle CX Commerce in the 19.x storefront framework, refer to Create a Widget.

Understand widgets

Widgets, which work together with plug-ins and other components, help you create your website user interface.

Before you can create custom widgets, ensure that your development environment is configured correctly. For detailed information on creating your environment, refer to Set Up a Development Environment.

Widgets are user interface elements of your application that work in tandem with *plug-ins*. Plug-ins are elements that have been configured or created to support your application. Each widget, which is also a plug-in, is an individual piece of code that performs specific tasks that interact with other plug-ins, such as actions, and can be re-used as needed. Widgets are the only component that is displayed on the Design page in the administration interface, and must be exported to be visible.

Widgets in OSF are state driven and are composed on the OSF server. Widgets be constructed from a broad spectrum of JavaScript technologies and are typically more concentrated and lightweight. OSF widgets can be grouped together into a *container* for easy placement on your layouts, or they can be referenced independently.

Design features with widgets

Use widgets when you want to design features that are available in different contexts on multiple pages. Widgets are also useful if you want to place an element in several locations or target specific audiences.

OSF provides several default widgets that give you the ability to interact with your customers, update and modify profile information, customize product information, and display payment and shipping methods. However, you may want to create a new widget or customize an existing widget.

For example, you might want to create a new widget that presents your shoppers with upsell information or customize an existing widget that allows your shoppers to set up a default favorite store. Or you may want to create a page where your shoppers can select a specific currency. Widgets allow you to create these interfaces.

Prerequisites for widgets

Before you create or customize widgets, you should be familiar with the technologies described in the Prerequisites section. You should also be familiar with creating and using the following:

- Redux Store – (Information can be found at https://redux.js.org/api/store/)
- Actions – (Information can be found at https://redux.js.org/basics/actions)
- Sagas– (Information can be found at https://redux-saga.js.org/docs/introduction/BeginnerTutorial.html)
- Reducers – (Information can be found at https://redux.js.org/basics/reducers/)
- Promises – (Information can be found at https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Promise)

You must also create a workspace in which your widgets will reside. You do this using the `create-workspace` tool, which is described in Set up a workspace.

Understand how widgets work

Widgets are React function components that do the following:

- Export a function that accesses a `props` object that defines its properties.
- Return a React element that is specified with the JSX syntax.

The following is an example of a very basic widget. This example shows a simple component that takes `props` and then returns a `<div>` with a text string:

```
import React from 'react';
const MyWidget = props => {
  return (
    <div>
      This is my widget!
    </div>
  );
 };
 export default  MyWidget;
```

Widgets typically display information from the Redux store. OSF provides a higher-order `connect` component that allows a widget to update its display when specific properties change within the Redux store. Widgets can also invoke *actions*, which are payloads of information that perform specific tasks and can result in updates to the Redux store.

Widgets are modules that export a function. Functions accept a `props` object that contains the properties of the component. This allows you to include that component in another component or page. The function returns a React element that is added to the DOM. React uses a JSX syntax that allows you to create composite elements that are expressed in an HTML-like manner.

The following is a high-level overview of how a typical widget works:

1. A widget's initial display is generated on the server using information that has been applied to the Redux store by a data *fetcher*. Fetchers determine actions that the component takes when generating the initial state and typically invoke an Oracle CX Commerce REST API endpoint and apply the response to the Redux store. Fetchers run on the server if the widget is rendered due to a full page request. However, if the widget is rendered while dynamically creating a layout as a result of navigation within a single-page application, the fetcher runs on the client.

2. The widget invokes an *action*, for example, in response to a user gesture. An action is a function in the store's API that accepts an action type and an optional payload containing additional information.
3. Using information from the action, the Redux store invokes a Saga effect, resulting in an asynchronous call to an endpoint of the Oracle CX Commerce REST API.
4. The endpoint response is applied to a specific area of the Redux store.
5. The widget displays the updates that reflect that were made in the Redux store.

Before you can use a widget, you must export the widget information to your server's framework. When you call the client, meta, or server start functions, using the `client`, `meta`, or `server.js` files located in your application's `/src` directory, the system registers all of the exported sub-modules from the `/plugins` directory as parameters to those functions. This tells the framework of their availability. The `root` widget, which is a required component, is rendered as the top-level React component. There are additional components that are required by applications, including the `container` and the `connect` components. These required components are available to you by default.

Identify types of widgets

There are three different types of widgets, which are set using the `type` attribute in the component-level `meta.js` file. You should declare the type of widget you are creating based upon how you intend to use it:

- Container – Containers are place holders on a layout. You can drag other widgets into a container, thereby grouping a number of related widgets together on the layout. You can also use containers to duplicate a set of widgets across a layout. Container widgets can be nested within other containers up to five levels. Containers identify regions on the layout that allow you to add components. Refer to Understand containers for information on containers.
- Stacks – Stacks are like containers. The following stacks types are available: Accordion, Tab, Popup, and the Checkout Layout Progress Tracker.
- Individual – Individual widgets are singular widgets that can be added to layouts, as well as to container or stack widgets. This is the default setting. If you do not specify a type, the widget you create is an individual widget.

Review widget file structure

Widgets have a specific directory structure. The path structure uses several files that communicate with one another to share data. Note that there are application-level files, plug-in level files, and component-specific files.

As described in Develop and Deploy Applications, when you create an application, you define specific files and file structures. In addition to creating assets, you create a `/src` directory, which contains a `/plugins` directory. This directory contains all of the components needed to work with widgets.

The `/src` directory should also contain the following application-level files:

- `client.js` - This file provides client-side JavaScript entry points to your application. The client-side application code registers components and other entities by passing them in to a `createClient` function, which initializes a Redux store.

- `server.js` - This file initiates the `createServer` function, which creates a server factory. Components and other metadata is available to your application when you provide information to the server factory.
- `meta.js` - This file creates an object that contains your application's metadata, which is then passed to the `createServer` function in the `server.js` file.

The `/src/plugins` directory contains the code used by your widget. As an example, the `currency-selector` widget, located in the `/my_app/src/` directory, has the following data structure and files:

```
/plugins
  /actions
     index.js
     meta.js
     /get-currency
       index.js
       meta.js
  /components
     index.js
     meta.js
     /currency-selector
       index.js
       meta.js
       styles.css
       /locales
           de.js
           en.js
  /endpoints
     index.js
     meta.js
     /get-currency
        index.js
        meta.js
```

Understand widget composition

All widgets are composed of some combination of the following files. These files are laid out in the file structure specified in the description:

File name	Description
`index.js`	Each plug-in type has its own `index.js` file. For example, the `/package/plugins/actions/index.js` file contains a dynamic import reference to the module-based actions. Each widget has its own component-level `index.js` file. This `/packages/plugins/component/component-name/index.js` file performs a dynamic import of the helper components, classes, and plug-in-specific objects.

File name	Description
`meta.js`	The `meta.js` file stores metadata of the component, such as resources, configurations, actions, endpoints, fetchers and the type of component. It is best to limit `meta.js` file dependencies to contain only fetchers, locales, configurations, type and `packageId`. Each plug-in has its own root `meta.js` file. This `meta.js` file makes a dynamic reference to the module-based plug-in directory. Each widget has its own component-level `meta.js` file. This `/packages/plugins/component/component-name/meta.js` file identifies fetchers and actions, as well as performs a dynamic import of dependencies required for the component.
`selectors.js`	This file contains helper methods that select the required data from the store state, inject that data into a `props` object, and share the data with the widget. Selectors are written at the component-level or the application-level. (Application-level selectors can be shared among multiple widgets.)
`styles.css`	This component-level file defines widget-specific CSS styles.
`config.js`	This component-level file provides the widget settings that are imported in the `meta.js` file. It is recommended that you write a `config.js` file that contains the necessary configuration settings and refer to the `config.js` file from the `meta.js` file.

Understand containers

You can use containers to nest components within other components. Containers, which are regular components with associated layouts, can be nested up to five levels deep. Container components are React components that provide a list of regions as part of their state model. Containers can also pass context down to child components.

The component is responsible for rendering the layout. The following is an example of a container widget's `/plugins/components/container/index.js` file:

```
import React from 'react';
 import Region from '@oracle-cx-commerce/react-components/region';
 import Styled from '@oracle-cx-commerce/react-components/styled';
 import css from './styles.css';

 const Container = props => {
  const {regions = [], configuration = {}} = props;
  const {className = ''} = configuration || {};

  return (
    <Styled id="Container" css={css}>
```

```
      {/* render each child region */}
      <section className={`Container__Section ${className}`}>
        {regions.map(regionId => (
          <Region key={regionId} regionId={regionId} />
        ))}
      </section>
    </Styled>
  );
 };
 export default Container;
```

This example shows how the container defines regions and the container class name.

The following is an example of a container widget's `/plugins/components/container/meta.js` file. The metadata for a container widget might contain information such as the following:

```
import config from '../../config/general/config.json';
 import de from '../../config/general/locales/de.json';
 import en from '../../config/general/locales/en.json';

 export const Container = {
  type: 'container',
  config: {
    ...config,
    locales: {
      en,
      de
    }
  }
};
```

Containers can also be preconfigured and uploaded with an application. This allows you to associate a predefined layout with a default container component. For example, you could upload a custom header that is comprised of nested components. Predefined containers that use predefined layouts are stored in your `/assets/containers` directory. Once you have uploaded a predefined container, it is available in the component drawer in the administrative interface and can be dragged onto a layout. Once it has been dragged onto a layout, the administrative interface creates a component instance that uses the associated predefined layout.

When you create a predefined container, you create the necessary JSON files in the `/container` directory in the root level of your application.

For example, if you were to create a `/src/components/container/header.json` file, it might contain the following:

```
{
  "title": "Header",
  "component": "Container",
  "layout": [
    {
      "width": 2,
      "components": [
```

```
        "MenuMobile"
      ]
    },
    {
      "width": 8,
      "components": [
        "Logo"
      ]
    },
    {
      "width": 1,
      "components": [
        "SearchMobile"
      ]
    },
    {
      "width": 1,
      "components": [
        "CartStatusMobile"
      ]
    }
  ]
}
```

The metadata of this container contains the `title`, base `component` and the `layout`.

Once you have created the container, you can reference it from the default page layouts, located in the `/pages/home.json` file:

```
{
  "title": "Homepage",
  "address": "/home",
  "shortName": "home",
  "layout": [{
      "type": "header",
      "width": 12,
      "components": [
        "Header"
     ]
    },
    {
      "type": "main",
      "width": 12,
      "components": [
        "Placeholder"
      ]
    },
    {
      "type": "footer",
      "width": 12,
      "components": [
        "Footer"
      ]
    }
```

```
  ]
}
```

Understand plug-ins

Widgets are used in conjunction with the specific elements, known as plug-ins. Plug-ins enable a widget to invoke API functions, retrieve state information, send and retrieve data from your application and watch for registered actions.

Plug-ins can be used independently, but are usually combined to create functionality:

- Endpoints – Endpoints allow you to create an adapter to Oracle CX Commerce REST API functions. For detailed information on creating and working with endpoints, refer to Create an endpoint.
- Actions – Actions, which are invoked by widgets, send data from your application to the state store, as well as identify any behavior that should occur in response to a change of state. For detailed information on creating and working with actions, refer to Write an action.
- Fetchers – Before they can be rendered, widgets require that data from many sources be uploaded to the state model. Fetchers, which allow you to identify this data, are functions that use the current store and widget instance configuration values to return a promise. The promise is resolved once the fetcher has completed loading data. Fetchers load data into the state using the `store.endpoint` method. For detailed information on creating and working with fetchers, refer to Write a fetcher.
- Subscribers – Subscribers passively watch for registered actions that occur outside of the component. Once an action is recognized, sagas dispatch them. Subscribers do not invoke Redux store actions. They can, however, invoke endpoints independent of the Redux store. Note that subscribers run only on the client. For information on subscribers, refer to Create a subscriber.
- Helper components – Helper components are any framework module that can be individually referenced. For information on components, refer to Understand helper components.

Understand helper components

Widgets access and use built-in plug-ins that perform various functions. These functions, or helper components, allow you to further customize your widgets by enabling formats, reading forms, and performing other functions.

Widgets can reference helper components exposed by the application, such as those provided in the `/react-components` directory, or custom components that are defined explicitly within your application's `/plugin` directory. Helper components exist under your application and can be referenced in other components or widgets. These helper components include formats for adding icons, images, links, styles, and others. For example, the `Form` component allows you to display HTML in a form.

Helper components are abstract, reusable lower-level components or utilities. Although not exposed in the Design page of the administration interface, they can be used when coding widgets and containers. Note that these components are not connected to the state model and get their state information from widgets.

Important: Do not modify the generic helper components, as you may affect the functionality and performance of your application.

The following are some of the helper components that are available:

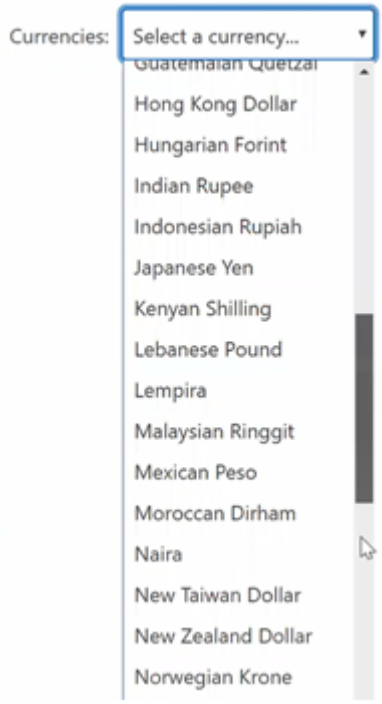
- `Form` - The `Form` component displays HTML forms. It supports callback methods to invoke actions, for submissions and errors. This component has default form field validation and uses the `validationMessage` class. You can add custom validation into the Form component. When `Form` validation occurs, it is best to ensure that it is using HTML5 validation. Note that any form validation should be displayed above the form.
- `Img` - This component displays images. It takes callback methods and fallback images when working with failed images.
- `Icons` - Icons should be created as a separate component in your application. They must be defined with an SVG paths defined in the component.
- `Styled` - This component reads CSS and appends it to the HTML header. The `Styled` component should be wrapped around all components that have a CSS style.
- `Link` - This component should be used on all page navigation that call a URL, as opposed to using an anchor tag with `href` links. Using an anchor tab with an `href` link causes the entire page to refresh every time one of the links is selected.

Build a widget

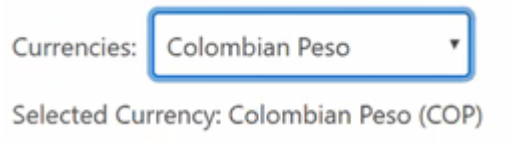
Widgets allow you to create graphic elements for your storefront. There are several steps to creating a widget, including planning, associating layouts, creating components and registering.

This section takes the concepts discussed earlier and walks you through the building of a widget. The following is an example of a widget that allows a shopper to see a currency symbol for a specific currency, or the `CurrencySelector` widget. The widget example shows you how to get data from a REST API call for your initial display and then get additional data from another API function that you can use to update your display.

This example application has a single currency selector widget that presents a dropdown list of all of the available currencies as returned by the `list-currencies` JavaScript endpoint.



This example has an event listener that listens for selections from this list. When a selection occurs, the ID of the selected currency is passed to the `getCurrency` REST endpoint to get specific details. When that information is received, it is displayed in an auxiliary panel, showing you the currency symbols for the available currencies. For example:



Steps to create a widget

Creating new widgets allows you to customize your website's interface, as well as provide additional functionality to your shoppers. Before you begin to develop new widgets, it is best to plan out your requirements.

Note: Before you create a widget, ensure that you have correctly configured your OSF workspace. Refer to Set Up a Development Environment for information on setting up and configuring your OSF workspace.

It is important that you have identified the task that your widget is going to perform. Additionally, identify if your widget needs data to be added to the Redux store, and if that data needs to be available during server-side rendering (SSR).

Ideally, a component should only perform one task. When creating a widget, try and make your components smaller, which will make your code reusable across different pages. It is also recommended that you create functional components, which consume less than a class-based component.

Once you have determined what and how your widget will work, perform the following tasks:

1. Create the plug-in directory structure needed for your new widget, as described in Review widget file structure.
2. Create the initial widget index and metadata JavaScript files, as described in . These files will be modified as you continue to add other plug-in components.
3. Add the widget to your application as described later in this section.
4. Add your widget to your home page layout as described in Identify the layout.
5. Create and generate the CSS files that your widget will use as described in Set the widget style sheet.
6. Display localized strings, if necessary, which is described in Create locale information.
7. If you have determined that your data needs to be available during SSR, add a new selector to the widget to access the data, as described in Create a selector.
8. To obtain the necessary data for the widget to run, create fetchers, as described in Write a fetcher.
9. To access an endpoint, write an action, as described in Write an action.
10. Create an endpoint, as described in Create an endpoint.
11. If you need to deploy a global widget that has no user interface, create a subscriber as described in Create a subscriber.

Create a widget directory structure

Your widget needs to follow a specific file structure. Create the file structure for you widget similar to that described in Review widget file structure. When you create a new widget, you will have to create these directories. If you clone and then modify an existing widget, these directories may already exist.

For our example, create the widget in the `/plugins/components/currency-selector/` directory. This directory will contain the widget you create for the currency selection.

Create widget files

As described in Understand widget composition, all widgets contain, at the very least, both an `index.js` file and a `meta.js` file. Note that these are different than the `index.js` and `meta.js` files that reside in the `/plugins` directory.

Using the currency selector example, when you write your widget, you must add it to the components directory in the `/plugins/components/currency-selector` directory that you just created. For example, the following `/plugins/components/currency-selector/index.js` file defines the widget:

```
import React from 'react';
const CurrencySelector = () => {
  const currencies = [
    {repositoryId: 'c1', displayName: 'Franc', currencyCode: 'FRA'},
    {repositoryId: 'c2', displayName: 'Pound', currencyCode: 'GBP'},
    {repositoryId: 'c3', displayName: 'Yen', currencyCode: 'JPY'}
  ];
  const selectedCurrency = currencies[0];

  return (
```

```
    <div>
      {currencies && (
        <div>
         <Dropdown
            id="CurrencySelector-list"
            name="CurrencySelector-list"
            label="Currencies:"
            aria-label="Currencies:"
            value={selectedCurrency && selectedCurrency.repositoryId}
            onChange={event => {
              console.log(`selected ${event.target.value}`);
            }}
            onBlur={() => {}}
          >
            <option value="">Select a currency...</option>
            {currencies.map(currency => (
              <option key={currency.repositoryId}
value={currency.repositoryId}>
                {currency.displayName}
              </option>
            ))}
          </Dropdown>
         </div>
       )}
      {selectedCurrency && (
        <div>
          Selected currency: {selectedCurrency.displayName}
({selectedCurrency.currencyCode})
        </div>
       )}
    </div>
  );
};
export default CurrencySelector;
```

Widget code has associated metadata, so whenever you create a module that implements your widget, you must also create a parallel metadata object. The following is an example of an undefined `/plugins/components/currency-selector/meta.js` file:

```
/**
  * Metadata for the currencySelector component.
  */
 export default {};
```

Add your widget to your application

OSF provides a number of default widgets that you can use when creating your own widget. To add a default widget and make it available to the system, edit the `index.js` and the `meta.js` files in the component directory, for example, `/plugins/component/index.js`. This `index.js` file exports a number of functions whose names correspond to the name of your application's widgets. Each function returns a dynamic import that resolves to a module whose default export is the React component that initiates the widget.

By default, when the system generates an application, it exports of all of the default widgets, which are available to you should you choose. Your application may not require all of the default widgets.

For example, to create the `currency-selector` widget, you would add the following to the `/plugins/components/index.js` file:

```
 // Export a reference to our _CurrencySelector widget.
export const _CurrencySelector = () => import('./currency-selector');
```

Whenever you create a new widget, you must add a reference to it.

Add the following to the `/plugins/components/meta.js` file:

```
export {default as _CurrencySelector} from './currency-selector/meta';
```

Note: Preceding the names of your new components with an underscore is the recommended naming convention for custom widgets, as it will prevent naming collisions with default widgets. For example, `_CurrencySelector`.

Identify the layout

Before creating the widget, set the layout page that the widget will use by setting the *`my_app`*`/assets/pages/home.json` file. Set the layout page as follows:

```
{
   "title": Home Page",
   "address": "/home",
   "shortName": "home",
   "defaultPage": true,
   "layout": [
     {
       "type": "main",
       "width": 12,
       "components": [
         "_CurrencySelector"
       ]
     }
   ]
 }
```

Note that the `_CurrencySelector` has been added as a component.

You initially create your layouts locally, and then upload them to the Design page of the administration interface.

Set the widget style sheet

Our example, the `CurrencySelector` widget, uses a default CSS style sheet that creates a CSS class called `CurrencySelector`. This file, which is defined in the `/`

`plugins/components/currency-selector/styles.css` file, contains standard CSS styling:

```
/**
 * CurrencySelector CSS definitions.
 */

 .CurrencySelector {
   padding: 10px;
 }
 .CurrencySelector__Label {
   padding-right: 10px;
 }
 .CurrencySelector__SelectedCurrencyInfo {
   padding-top: 10px;
 }
```

If you create or use any CSS that is specific to a user interface component, that CSS should be stored within the widget's `styles.css` file. Note that UI components should be plain CSS, which will enable you to apply styles on your UI components. It is also recommended that you use flex or grid CSS layouts.

Once you have created the CSS, you must update the Currency Selector widget to use the CSS:

1. Import the contents of the `styles.css` file into a string constant named `css`.
2. Wrap the widget's React element in a `Styled` component, with an `id` property named after the widget (in this case, `CurrencySelector`) and a `css` property that contains the imported CSS. This inserts the CSS into an HTML `style` element before rendering the widget.
3. Use class names from the CSS by setting the React `className` property on individual React elements.

Note: Always import or export individual constants, classes or functions separately and not by using wildcards. The following example shows how to take the `<div>` of the `className` and put it inside the `Styled` component. It gives it a unique ID, and then pulls in the CSS file as a plain string named CSS. This lets you set it as a property on the `Styled` component. That causes the CSS from that style sheet to be added to the DOM and makes it available for you to use. Then set the class name of the `<div>` to the `CurrencySelector` class that was identified in the style sheet. This is a portion of the example `/plugins/components/currency-selector/index.js` file:

```
import React from 'react';
import Styled from '@oracle-cx-commerce/react-components/styled';
import css from './styles.css';

const CurrencySelector = () => {
  const currencies = [
    {repositoryId: 'c1', displayName: 'Franc', currencyCode: 'FRA'},
    {repositoryId: 'c2', displayName: 'Pound', currencyCode: 'GBP'},
    {repositoryId: 'c3', displayName: 'Yen', currencyCode: 'JPY'}
  ];
  const selectedCurrency = currencies[0];
```

```
  return (
    <Styled id="CurrencySelector" css={css}>
      <div className="CurrencySelector">
        {currencies && (
          <div>
            <span className="CurrencySelector__Label">Currencies:</span>

            <select
              value={selectedCurrency && selectedCurrency.repositoryId}
              onChange={event => {
                console.log(`selected ${event.target.value}`);
              }}
              onBlur={() => {}}
            >
              <option value="">Select a currency...</option>

              {currencies.map(currency => (
                <option key={currency.repositoryId} value={currency.repositoryId}>
                  {currency.displayName}
                </option>
              ))}
            </select>
          </div>
        )}

        {selectedCurrency && (
          <div className="CurrencySelector__SelectedCurrencyInfo">
            Selected currency: {selectedCurrency.displayName}
({selectedCurrency.currencyCode})
          </div>
        )}
      </div>
    </Styled>
  );
};
export default CurrencySelector;
```

Note that the `/plugins/components/currency-selector/meta.js` file should contain the following:

```
import * as de from './locales/de.json';
import * as en from './locales/en.json';

export default {
  resources: {
    de,
    en
  }
};
```

For additional information on CSS performance, refer to Remove unnecessary CSS.

Create locale information

You can add styling and localized strings to your widget. The widget obtains localized text from a resource bundle that identifies the language of the shopper's browser.

Your application can store the necessary code in the `/locales` directory under your widget's component directory. For example, `/plugins/components/currency-selector/locales`. Note that all shared locale strings should be put into this common resource directory, however, component-specific locales should never be added to the common directory. CSS classes will also identify how to display your widget based upon the device used to view the site. For example, you may want to create mobile-specific CSS and desktop-specific CSS. These CSS classes can be differentiated using a `@media` query to identify different resolution and size images.

This directory should contain resource bundles for all of the supported locales. These resource bundles contain strings. For example, the English locale, or the `en.json` file, contains the following for the `CurrencySelector` widget:

```
{
  "labelCurrencies": "Currencies:",
  "labelSelectACurrency": "Select a currency...",
  "labelSelectedCurrency": "Selected Currency:"
}
```

Since resources are passed into widgets as normal props, it is best to use a naming convention that differentiates resource props from other props. It is recommended that you reuse labels, and avoid duplication.

And the German locale, or the `/plugins/components/currency-selector/locales/de.json` file, contains the following:

```
{
  "labelCurrencies": "[de]Currencies:[de]",
  "labelSelectACurrency": "[de]Select a currency...[de]",
  "labelSelectedCurrency": "[de]Selected Currency:[de]"
}
```

So that resources can be passed in as component props, replace your widget's `meta.js` file to be similar to the following:

```
import de from './locales/de.json';
import en from './locales/en.json';

/**
 * Metadata for the CurrencySelector component.
 */
export default {
  // Include references to all of our resource strings in all supported
locales.
  // This will enable the component to access any resource string via its
props,
  // using the locale that is currently in effect.
  resources: {
    de,
```

```
    en
  }
};
```

Replace the hard-coded strings in your widget with resource prop values:

```
import React from 'react';
import Styled from '@oracle-cx-commerce/react-components/styled';
import css from './styles.css';

const CurrencySelector = ({labelCurrencies, labelSelectACurrency,
labelSelectedCurrency}) => {
  const currencies = [
    {repositoryId: 'c1', displayName: 'Franc', currencyCode: 'FRA'},
    {repositoryId: 'c2', displayName: 'Pound', currencyCode: 'GBP'},
    {repositoryId: 'c3', displayName: 'Yen', currencyCode: 'JPY'}
  ];
  const selectedCurrency = currencies[0];
   return (
    <Styled id="CurrencySelector" css={css}>
      <div className="CurrencySelector">
        {currencies && (
          <div>
            <span className="CurrencySelector__Label">{labelCurrencies}</span>
             <select
              value={selectedCurrency && selectedCurrency.repositoryId}
              onChange={event => {
                console.log(`selected ${event.target.value}`);
              }}
              onBlur={() => {}}
            >
              <option value="">{labelSelectACurrency}</option>

              {currencies.map(currency => (
                <option key={currency.repositoryId} value={currency.repositoryId}>
                  {currency.displayName}
                </option>
              ))}
            </select>
          </div>
        )}
        {selectedCurrency && (
          <div className="CurrencySelector__SelectedCurrencyInfo">
            {labelSelectedCurrency} {selectedCurrency.displayName}
({selectedCurrency.currencyCode})
          </div>
        )}
      </div>
    </Styled>
  );
};
export default CurrencySelector;
```

Note that you must redeploy your application for localization to take effect as the localization mechanism requires that Design Studio text snippets have been created to contain the translated strings.

When you create shared locale resource strings, they should go into a common resource directory. However, ensure that you do not add component-specific locales into a common resource directory.

Now that you have created your widget, you can configure it to communicates with REST API endpoints.

Create a subscriber

Subscriber plug-ins are passive observers that allow you to deploy a global widget that has no user interface.

Subscribers allow you to add connections to other independent endpoints. For example, if you want to add Google Analytics to your site, use a subscribers plug-in. Subscribers do not invoke store actions, and as such, they receive read-only versions of the Redux store's API. You can, however, use subscribers to invoke endpoints independent of the Redux store. Note that subscribers are run only on the client-side.

Use a subscriber for any feature where you want to passively observe an event. If your feature needs to update the application state, then you should create an action. Creating a subscriber is similar to creating an action or an initializer. The following example shows how to create a subscriber named `my-subscriber`:

1. Create a `/plugins/subscribers/my-subscriber` directory to store your subscriber `index` and `meta.js` files.
2. Create the `/plugins/subscribers/my-subscriber/index.js` file. For example:

```
export default ({endpoint, getState, subscribeDispatch}) => {
 return subscribeDispatch(action => {
 const {type} = action;
 if (type === 'getState') {
 // Track getState action
 }
 else if (type === 'myOtherAction') {
 // Track myOtherAction action
 }
 });
};
```

3. Create the `/plugins/subscribers/my-subscribers/meta.js` file. For example:

```
export default {};
```

4. Export your subscriber by creating a `/plugins/subscriber/index.js` file that contains the following:

```
export const mySubscriber = () => import('./my-subscriber');
```

5. Then export your subscriber's metadata by creating a `/plugins/subscribers/meta.js` file that contains the following:

```
export (default as mySubscriber) from './my-subscriber/meta';
```

6. You must add the subscribers to your application's `client.js` file. For example:

```
import * as actions from './plugins/actions';
 import * as components from './plugins/components';
 import * as endpoints from './plugins/endpoints';
 import * as subscribers from './plugins/subscribers';
 import {createClient} from '@oracle-commerce/react-app/client';
 createClient({
   actions,
   components,
   endpoints,
   subscribers
}). then(({start}) => start ());
```

7. Finally, you must add the subscriber information to your application's `meta.js` file. For example:

```
import * as actions from './plugins/actions/meta';
 import * as components from './plugins/components/meta';
 import * as endpoints from  './plugins/endpoints/meta';
 import * as initializers from './plugins/initializers/meta';
 import * as subscribers from './plugins/subscribers/meta';
 import {createMeta} from '@oracle-commerce/react-app/meta';
 export default createMeta({
   actions,
   components,
   endpoints,
   initializers,
   subscribers
});
```

Configure a widget to use REST endpoints

Once you have created a custom widget, you can update your application to communicate with the REST API.

To continue creating the widget code that displays a list of currencies, you must obtain the location from which the data should be retrieved. For this example, assume that you get the data from the Redux store. First, you must declare the component. Do this by exporting a `connect` component. It acts as a wrapper around your component and lets you connect your component to a specific part of the state located in the Redux store. This allows properties to be passed in as `props` to your component. Then you pass a selector that points to a specific part of the Redux store into the constructor for that component. Whenever those properties change, your component will re-render. The `connect` component binds the display of your component to the values of your properties.

This is how your component obtains data. You can also use the `connect` component to perform specific tasks, which might result in updates to the data in the Redux store. This would result in re-rendering certain parts of your application.

This is an example of the widget's `/plugin/components/currency-selector/index.js` file where the wrapped component is identified:

```
/**
 * Wrap the component with a "connect" object that supplies the state's
 * currency information as props and redisplays the component when any
of those
 * props change.
 */
 export default connect(getCurrencyInfo)(CurrencySelector);
```

When you create a widget that may have significant differences between user interfaces, it is recommended that you create separate widgets.

In the following code, the HTML looks up the currencies array. Then it takes the given currency and adds options using the `displayName` of the given currency. This is defined in the `/plugin/components/currency-selector/index.js` file.

Once you have configured your widget to use endpoints, this is what the `index.js` file for the currency selector should resemble:

```
import React, {useContext} from 'react';
import {StoreContext} from '@oracle-cx-commerce/react-ui/contexts';
import Styled from '@oracle-cx-commerce/react-components/styled';
import {connect} from '@oracle-cx-commerce/react-components/provider';
import css from './styles.css';
import {getCurrencyInfo} from '../../selectors';
import {listCurrenciesFetcher} from '../../fetchers';
import {useListCurrenciesFetcher} from '../../fetchers/hooks';

// The server-side rendering framework checks each component for a
"fetchers" array
// to determine what actions to take in order to populate the initial
state. For this
// component, we need the currency list.
export const fetchers = [listCurrenciesFetcher];

/**
 * The CurrencySelector component.
 *
 * @param {object} props.currencies Array of info about available
currencies
 * @param {object} props.selectedCurrency Info about the selected
currency
 * @param {string} props.label* Resource strings
 */
const CurrencySelector = ({
  currencies,
  selectedCurrency,
  labelCurrencies,
  labelSelectACurrency,
```

```
  labelSelectedCurrency
}) => {
  // Make sure we have the latest currency list during client-side
rendering.
  const store = useContext(StoreContext);
  useListCurrenciesFetcher(store);

  // Invoked when the user selects a currency from the
  // select element. Invokes the '_getCurrency' action to
  // update the selected currency with info from the server.
  const onCurrencyChange = event => {
    const repositoryId = event.target.value;
    if (repositoryId) {
      store.action('_getCurrency', {repositoryId});
    }
  };

  // Display a panel with:
  // * a select element with a placeholder and all available currencies
  // * a panel with information about the selected currency
  return (
    <Styled id="CurrencySelector" css={css}>
      <div className="CurrencySelector">
        {currencies && (
          <div>
            <span className="CurrencySelector__Label">{labelCurrencies}</span>

            <select
              value={selectedCurrency && selectedCurrency.repositoryId}
              onChange={onCurrencyChange}
              onBlur={onCurrencyChange}
            >
              <option value="">{labelSelectACurrency}</option>

              {currencies.map(currency => (
                <option key={currency.repositoryId} value={currency.repositoryId}>
                  {currency.displayName}
                </option>
              ))}
            </select>
          </div>
        )}

        {selectedCurrency && (
          <div className="CurrencySelector__SelectedCurrencyInfo">
            {labelSelectedCurrency} {selectedCurrency.displayName}
({selectedCurrency.currencyCode})
          </div>
        )}
      </div>
    </Styled>
  );
};

/**
```

```
 * Wrap the component with a "connect" object that supplies the state's
 * currency info as props and redisplays the component when any of those
 * props change.
 */
export default connect(getCurrencyInfo)(CurrencySelector);
```

The `select` attribute has an `onChange` handler, and a function that is defined named `onCurrencyChange`. The `value={currency.repositoryId}` uses the ID of the currency as the value of the option. If the ID is defined, the code calls an action, which allows the widget to react to changes made in the currency list.

Widgets share detailed information using context. For example, the `StoreContext` is a store object method used to invoke actions. The store object has an action method that uses the action name and parameters to invoke actions on the store. Each action should have a corresponding saga or reducer to update the store with the new state.

Context provides a framework for component communication. The `useContext` React hook allows actions to read and access default context to the widget. Use the `useContext` hook to obtain the context of an application. There are several context that can be used within an application. The contexts used by each widget indicates what methods and utilities will be available. Context invokes actions and returns action methods.

To reference the action in the example, use the React `useContext` hook. The `storeContext` allows you to get the value of a global variable, or a reference to the applications state. Once you have referenced the action, use the action function to dispatch actions. The action this example dispatches is the `_getCurrency` action. This example passes in the payload, which is the repository ID of the currency requested. This is described in the `/plugin/components/currency-selector/index.js` file. Note that this is a portion of the code example referenced above:

```
import React, {useContext} from 'react';
 import {StoreContext} from '@oracle-cx-commerce/react-ui/contexts';
...
  // Make sure we have the latest currency list during client-side
rendering.
  const store = useContext(StoreContext);
  useListCurrenciesFetcher(store);

  // Invoked when the user selects a currency from the
  // select element. Invokes the '_getCurrency' action to
  // update the selected currency with info from the server.
  const onCurrencyChange = event => {
    const repositoryId = event.target.value;
    if (repositoryId) {
      store.action('_getCurrency', {repositoryId});
    }
  };
```

When this action is invoked, the `getCurrency` endpoint is called with the given ID and a part of the state identified as `myRepository.currencyInfo.selectedCurrency` is updated. Because the code is listening for changes in `CurrencyInfo`, it re-renders when there are changes. The following example uses a `<div>` that is rendered when there is a selected currency. This `<div>` contains a source string, a `displayName` and a

`currencyCode`. This is added to the `/plugin/components/currency-selector/index.js` file. Note that this is a portion of the code example referenced above:

```
{selectedCurrency && (
  <div className="CurrencySelector__SelectedCurrencyInfo">
    {labelSelectedCurrency} {selectedCurrency.displayName}
       ({selectedCurrency.currencyCode})
  </div>
)}
```

When you access store data on your UI components, use selectors and pass them using the `connect` method. It is important that you use the correct selector method to fetch the precise information required for rendering the UI component. There are a number of default selectors that enable you to retrieve store state, however, if you cannot find a selector that meets your needs, you will have to create a custom selector. You can also use the `useSelector` hook to obtain part of the state information.

When you use the `StoreContext` hook to get the store object, you must us an `action` method that uses the action's name and parameters to invoke actions on the Redux store. Each action must have a corresponding saga or reducer that updates the store with the new state. Note that the action dispatch call should happen only when using the `useEffect` hook. It is best not to invoke any action calls in a React function component `render` method or `useMemo` hook.

Create a selector

Use a selector to access state models.

Selectors return specified data from the data store. If the data has not been updated, it will not be selected.

In actuality, you are exporting a wrapper around the component. The code arranges for the properties in this part of the Redux store to be passed into the component as `props`, and will redisplay the components if any of the properties change.

In this example, the selector named `getCurrencyInfo` is added within your widget's `index.js` file. However, you can define selectors in other modules and reference them in your widget. For this example, this widget looks for the currency information in the state and looks in a sub-property of the state called `myRepository`. Within that repository there is a sub-property called `currencyInfo`. This is the convention for referring to parts of the state, which allows you to create individual functions that return the values of specific parts of the state. If you have not defined a function, it returns an empty object.

The following example uses `props` that are a combination of resource strings and `props` that are in the `currencyInfo` object. The currencies property populates the drop-down list.

Before you can create a selector, you must create a `/selectors` sub-directory in the `/src/plugins` directory. This is an example of the selector added to the `/src/plugin/selectors/index.js` file:

```
/**
 * Selector to extract myRepository from a state object.
 */
export const getMyRepository = state => state.myRepository || {};

/**
 * Selector to extract myRepository.currencyInfo from a state object.
 */
export const getCurrencyInfo = state =>
getMyRepository(state).currencyInfo || {};

/**
 * Selector to extract myRepository.currencyInfo.currencies from a state
object.
 */
export const getCurrencies = state => getCurrencyInfo(state).currencies
|| {};
```

Write a fetcher

Typically, a widget renders your page on the server as the initial display for the widget is generated. The data needed to do this is obtained using data fetchers.

There are many default data fetchers, which you can reference from the `/oracle-cx-commerce/fetchers` directory, but you can also create your own.

Data fetchers are plug-ins that can be included in your application to identify what your widget needs before it can be rendered. When you use a template to create a widget, the template pulls in all default fetchers. Fetchers that you create to use with your widget are stored in your application's `/plugins/fetchers` directory.

There are two types of fetchers. The first is a *global* fetcher that is independent of the widget. This means that the widget does not use the `widgetConfig` parameter. Global fetchers can be used by any widget, or by multiple widgets simultaneously. This is a generic fetcher that loads the current site:

```
const fetchSite = store => store.endpoint('getSite');
```

The other type of fetcher is a *component-specific* fetcher, or one that is dependent on the widget configuration. These fetchers create requests that are specific to the widget instance and may return unique results per widget instance.

The following is a generic global fetcher that loads the category menu data:

```
const fetchMenuCategories = (store, widgetConfig) => {
  // Get the necessary values from widget config
  const {numberOfChildCategoriesToDisplay = 3, hideCategoryIfOutOfStock =
false} = widgetConfig;

  // Pass the widget config values to the endpoint call
```

```
  return store.endpoint('getCollectionMenu', {
    categoryId: 'rootCategory',
    filterKey: 'categoryNavData',
    expandChildren: true,
    maxLevel: numberOfChildCategoriesToDisplay,
    disableActiveProdCheck: hideCategoryIfOutOfStock
  });
};
```

Fetcher loads data into the state using the `store.endpoint` method. For example:

```
interface Fetcher {
  (store: Store, [widgetConfig: Object]): Promise<*>
}
```

Use the following steps to create an fetcher:

1. Create the appropriate applications `/plugin` directory.
2. Once you have the application directory, create a directory named `/plugins/fetchers`. This directory should contain an `index.js` that contains the following:

   ```
   /**
    * References to the application's custom data fetchers.
    */
   export {listCurrenciesFetcher} from './list-currencies';
   ```

3. In the `/plugins/fetchers` directory, create a `hooks.js` file that contains the following:

   ```
   /**
    * References to the application's custom data fetcher hooks.
    */
   export {useListCurrenciesFetcher} from './list-currencies/hook';
   ```

4. Now create a sub-directory within the `/plugins/fetchers` directory, for example `/plugins/fetchers/list-currencies/`. This directory will also contain an `index.js` and a `hook.js` file.
5. Create the fetcher `index.js` file. For example, create the `/plugins/fetchers/list-currencies/index.js` file:

   ```
   /**
    * Fetcher that requests the currency list using an endpoint.
    */
   export const listCurrenciesFetcher = store =>
   store.endpoint('_listCurrencies');
   ```

6. Create the fetcher `hook.js` file. The hook file allows fetchers to be run inside React components. Fetcher hooks are based on the `useEffect` hook and run only in the browser. The `useEffect` hook allows you to invoke the fetcher only after the

page has been rendered. For example, create the `/plugins/fetchers/list-currencies/hook.js` file:

```
import {useEffect} from 'react';
import {isEmptyObject} from '@oracle-cx-commerce/utils/generic';
import {getCurrencies} from '../../selectors';
import {listCurrenciesFetcher} from '..';

/**
 * This hook will invoke the listCurrenciesFetcher if the currency
 * list is not already available in the application state.
 */
export const useListCurrenciesFetcher = store =>
  useEffect(() => {
    if (isEmptyObject(getCurrencies(store.getState()))) {
      listCurrenciesFetcher(store);
    }
  }, [store]);
```

It is a good practice to conditionally invoke the fetcher in the `hook.js` file to ensure that the fetcher is not already executed on the server-side. The above example conditionally invokes the fetcher after the `isEmptyObject` validation.

For performance information, refer to Prevent useEffect() from executing unnecessarily.

Create an endpoint

You can use an OSF function called `createEndpoint` to create an adapter to Commerce REST API functions.

Endpoints handle information provided to the REST URL and return information on how to implement the response.

Endpoints use the `processInput` and `processOutput` functions to provide and return information.

When you create a new endpoint, import the `createOccRestRequest` function to make calls to the REST endpoints. This function contains all the necessary information to ensure that correct headers are included when you create a REST request.

The following is an example of a `/plugins/endpoint/list-currencies/index.js` file. In this example, the adapter takes a payload sent to the action and converts it into whatever form is required by the REST endpoint:

```
import {createEndpoint, getBodyAsJson} from '@oracle-cx-commerce/wapi/
endpoint';
import {populateError} from '@oracle-cx-commerce/wapi/utils';

/**
 * Adapter for the _listCurrencies endpoint.
 */
export default createEndpoint('listCurrencies', {
  /**
   * Perform any necessary extra processing on the payload object that is
```

```
   * included when dispatching an action that invokes this endpoint.
   *
   * @param {object} payload The action payload
   *
   * @return {object} Object with extra info to be included with endpoint
request
   */
  processInput() {
    // Include a query parameter that limits the set of returned property
values.
    return {
      query: {fields: 'repositoryId,displayName'}
    };
  },

  /**
   * Convert the response from the endpoint invocation into an object
   * to be merged into the application state.
   *
   * @param {object} response The response from the endpoint call
   *
   * @return {object} An object to be merged into the application state
   */
  async processOutput(response) {
    const json = await getBodyAsJson(response);
    // Store the response "items" array under
myRepository.currencyInfo.currencies.

    return response.ok
      ? {
          myRepository: {
            currencyInfo: {
              currencies: json.items
            }
          }
        }
      : populateError(response, json);
  }
});
```

The `processOutput` function takes the response of the API call and stores it within the appropriate place in the state. This endpoint identifies an endpoint ID, which is how you associate this endpoint with the `getCurrency` Commerce REST API. The `index.js` file take the repository ID from the payload and adds it to the path of the URL. For example, if you call `/ccstore/v1/store/currencies` endpoint, your response will contain currency. It returns this because the repository ID is added to the path. Because this example only uses the `currencyCode` and `displayName` fields a query parameter limits the set of properties returned.

The code also shows you how to return an object that augments the URL for the endpoint. This utility function gets an object out of the returned response and takes that object and merges it into the correct part of the Redux state. This is what changes the panel that is displayed to the shopper. The `get-currency` endpoint was invoked for this currency, and the display name and the currency code were returned.

Create a corresponding `/plugins/endpoint/list-currencies/meta.js` file that indicates the name of the associated endpoint by doing the following:

```
/**
 * Metadata for the _listCurrencies endpoint.
 */
export const _listCurrencies = { }
```

You must export the endpoint so that your application can recognize it. The `/plugins/endpoint/index.js` file contains a set of functions with names that correspond to the names of your application's endpoints. Edit the `/plugins/endpoint/index.js` file to contain the following:

```
/**
 * This module exports references that enable the application's
 * endpoints to be accessed using dynamic imports.
 */

// By default, all available endpoints are exported.
export * from '@oracle-cx-commerce/endpoints';

// Export a reference to our _getCurrency endpoint.
export const _getCurrency = () => import('./list-currencies');
```

You must also edit the `/plugins/endpoint/meta.js` file:

```
/**
 * This module exports references to metadata for the application's
endpoints.
 */
export * from '@oracle-cx-commerce/endpoints/meta';
export {_getCurrency} from './list-currencies/meta';
```

Write an action

Actions are payloads of information that identify the state of your application. Actions, which are JavaScript objects, use a `type` property that indicates the type of action it performs, such as a subscription, get state, or a process input.

When your widget is displaying its initial data you may want to configure a button that invokes an action that generates data. When you invoke an action, you specify the type of action so that the system knows what to do. You may also choose to include additional information. The system takes that action and invokes an endpoint in the REST API. It does this using Redux-Saga, which enables asynchronous calls. Once the endpoint is invoked, the system takes its response and places it into a specific part of the Redux store. Selectors that listen to that part of the Redux store (using the `connect` elements described earlier) will be updated to reflect the new data.

Actions return *promises*, which represent the eventual completion of an operation and its resulting value. Promises are either resolved, where no exception occurs, or rejected, where an exception occurs. Resolved actions are identified by the OK status of the response. Once an action is either resolved or rejected, it is completed.

Actions do not describe how the application state changes, however they do use *reducers*, which specify how an application's state changes in response to actions that are sent to the state tree. A reducer is a pure function that takes the previous state and an action and returns the next state. OSF provides several default actions, which are found in the `@oracle-cx-commerce/actions/<action_type>` directory, that contain information that sends data to an application. Information is also sent to and from a state tree for your application. You can use default actions to perform such tasks as displaying product information or profile addresses, retrieving inventory and adding or removing shipping group information.

When you create your own action, it should be stored in your `/plugins/actions/<action_type>` directory.

The following example of a `/plugin/actions/get-currency/meta.js` file creates the `get-currency` action. The metadata for this action specifies an `endpoint` property, which is required to associate this action with an endpoint:

```
/**
 * Metadata for the _getCurrency action.
 */
 export const _getCurrency = {

   // This action uses a Saga to invoke the _getCurrency endpoint.
   endpoints: ['_getCurrency']
};
```

The action `/plugin/actions/get-currency/index.js` file contains a saga:

```
import {endpointSaga, takeLatest} from '@oracle-cx-commerce/store/
utils';
 /**
 * The getCurrency action.
 *
 * This exports an object with a generator function named "saga", whose
presence
 * signals OSF to pass the generator function to Redux-Saga's
middleware.run API
 * the first time the action is dispatched via the store API.
 *
 * The generator function results in an asynchronous endpoint invocation
 * when the action is dispatched.
 */

 export default {
   *saga() {
     yield takeLatest('_getCurrency', endpointSaga);
   }
 };
```

When you create an action that exports an object with a generator function, whose name is `saga`, the system adds it to the saga middleware to execute an endpoint. This means that when the action is dispatched, the `get-currency` endpoint will be asynchronously executed.

Sagas use the `takeEvery` and `takeLatest` functions. The `takeEvery` function allows multiple instances to be started concurrently. The `takeLatest` function allows only the latest request fired (displaying the latest version of data). It only allows one task to run at any moment.

Once you have created your actions, you must add them to the `/plugins/actions/index.js` and `meta.js` files. The following is an example of an `index.js` file:

```
/**
 * This module exports references that enable the application's
 * actions to be accessed using dynamic imports.
 */
 // By default, all available actions are exported.
 export * from '@oracle-cx-commerce/actions';
 // Export a reference to our getCurrency action.
 export const getCurrency = () => import('./get-currency');
```

The following is an example of the `/plugins/actions/meta.js` file:

```
/**
 * This module exports references to metadata for the application's
actions.
 */
 export * from '@oracle-cx-commerce/actions/meta';
 export {getCurrency} from './get-currency/meta';
```

To invoke an endpoint response from the `_getCurrency` action, you must create an endpoint adapter similar to the one you created in the CREATE AN ENDPOINT section.

To do this, create a `/get-currency` subdirectory in your application's `/src/plugins/endpoints` directory. Within this directory, create an `index.js` file similar to the following:

```
import {createEndpoint, getBodyAsJson} from '@oracle-cx-commerce/wapi/
endpoint';
import {populateError} from '@oracle-cx-commerce/wapi/utils';

/**
 * Adapter for the _getCurrency endpoint.
 */
export default createEndpoint('getCurrency', {
  /**
   * Perform any necessary extra processing on the payload object that is
   * included when dispatching an action that invokes this endpoint.
   *
   * @param {object} payload The action payload
   *
   * @return {object} Object with extra info to be included with endpoint
request
   */
  processInput({repositoryId}) {
    // Append to the REST resource URL the repository id of the requested
    // currency object, and include a query parameter that limits the set
```

```
    // of returned property values.
    return {
      params: [repositoryId],
      query: {fields: 'currencyCode,displayName'}
    };
  },

  /**
   * Convert the response from the endpoint invocation into an object
   * to be merged into the application state.
   *
   * @param {object} response The response from the endpoint call
   *
   * @return {object} An object to be merged into the application state
   */
  async processOutput(response) {
    const json = await getBodyAsJson(response);
    // Store the returned currency info under
myRepository.currencyInfo.selectedCurrency.

    return response.ok
      ? {
          myRepository: {
            currencyInfo: {
              selectedCurrency: json
            }
          }
        }
      : populateError(response, json);
  }
});
```

Once you have created the `index.js` file, create the `meta.js` file:

```
/**
 * Metadata for the _getCurrency endpoint.
 */
export const _getCurrency = {};
```

Once you have created these files, you must add an endpoint adapter to your application. To do this, edit the `src/plugins/endpoints/index.js` and `meta.js` files. The following is an example of the `index.js` file:

```
/**
 * This module exports references that enable the application's
 * endpoints to be accessed using dynamic imports.
 */

// By default, all available endpoints are exported.
export * from '@oracle-cx-commerce/endpoints';

// Export references to our own endpoints.
```

```
export const _getCurrency = () => import('./get-currency');
export const _listCurrencies = () => import('./list-currencies');
```

The following is an example of the `meta.js` file:

```
/**
 * This module exports references to metadata for the application's
endpoints.
 */
export * from '@oracle-cx-commerce/endpoints/meta';
export {_getCurrency} from './get-currency/meta';
export {_listCurrencies} from './list-currencies/meta';
```

6
Design Storefront Pages

This chapter describes how designers and business users can work with Open Storefront Framework (OSF) design assets in the Oracle CX Commerce administration interface. It assumes you have some experience working with layouts and widgets in the administration interface.

Understand Default Open Storefront Framework Widgets

The Open Storefront Framework (OSF) includes a number of default layouts and widgets that you can customize as you build your store.

This release of OSF includes a template that acts as a core reference store with a library of reference components and core shopping flows, leveraging best practice UX, to help accelerate new deployments and maintain a best practice implementation. The application is optimized for mobile storefronts.

Layouts represent different page types, such as product details, checkout, and order confirmation. The default layouts act as templates, and you use them to design, adjust, and preview your pages before publishing live. Each layout has access to a group of layout-specific widgets that are individual pieces of code that perform specific tasks.

Note: OSF includes layouts and widgets for the storefront only. The Agent interface continues to be managed with the classic framework.

This section describes each of the default layouts in OSF, including the default widgets available for each layout. Detailed information about each widget's code, including actions, selectors, fetchers, and endpoints, is available in the JS API documentation that you can download from Oracle Customer Connect.

The reference application includes the following shopper flows:

- Registered & Anonymous Shoppers Visits
- Browse/Search
- Add to Cart
- Shopping Cart
- Checkout
- Split Shipping
- Login
- Register
- Managing Shopper Profile

Home page widgets

The reference template includes the following widgets for the store's home page.

- Progress Bar

- Cart Link
- Country Store/Language/Currency Picker - Mobile
- Image Widget (similar to a promotion banner)
- Footer Container, including Footer Link, Copyright WebContent, and Social Links
- Header Container Widget - Mobile
- Profile Registration, Login, Forgot Password Update Password, Update Expired Password

Product Listing and Search widgets

The reference template includes the following widgets for the store's product listing pages, including search containers.

- Back to Top Link
- Faceted Navigation - Mobile
- Product Listing Container, including the listing-summary and load-more-button - Mobile
- Typeahead Search - Mobile
- Dynamic Breadcrumbs
- Result List Table - Mobile

Product Details page widgets

The reference template includes the following widgets for a product's details page. These widgets are all included in the Product Details Container widget.

- Product Add To Cart Button
- Product Inventory Status
- Product Details Container
- Product Variant Options
- Product Long Description
- Widget Product Summary (Short Description)
- Product Image Viewer
- Product Name
- Product Price
- Product Quantity
- Shipping Surcharge
- Social Sharing Widget

Shopper Profile widgets

The reference template includes the following widgets for the registered shopper profile.

- Profile Order Actions

- Profile Email Marketing Preferences Summary, Profile Email Marketing Preferences
- Profile Recent Orders
- Profile Welcome
- Navigation Breadcrumbs
- Profile Return Details
- Profile Account Details and Update Password
- Profile Order Information, Profile Order Details Container
- Profile Order History
- Profile Submit Order Cancellation
- Profile Address Book Widget (summary, address book list, create and edit address)
- Profile Return Items Request Container

Shopping Cart widgets

The reference template includes the following widgets for the cart.

- PromotionsDisplay
- PromotionCodeEntry and Promotion Container
- CartOrderSummary, ProfileOrderSummary
- Add Gift With Purchase Support to CartItemDetails
- CheckoutButton with Validate Cart
- CartItemDetails Widget

Checkout widgets

The reference template includes the following widgets for the cart.

- PromotionsDisplay
- PromotionCodeEntry and Promotion Container
- CartOrderSummary, ProfileOrderSummary
- Add Gift With Purchase Support for CartItemDetails
- CheckoutButton with Cart Validation
- CartItemDetails Widget

Payment widgets

The reference template includes the following widgets that handle order payments. For details about configuring payments in an OSF storefront, see Supported payment methods and transaction types.

- Checkout Payment Methods Container
- Profile Saved Cards Widget
- Profile Saved Cards - Add Credit Card
- Profile Saved Card Summary

- Checkout Gift Card Status Widget - Show Gift Card Applied status
- Checkout Gift Card Entry Widget - Apply Gift Card

Understand how to use the Design page with OSF applications

If you are experienced using the Design page in the administration interface to work with classic framework design assets, you will find working with OSF design assets to be a nearly identical experience.

Not all Design page features you are used to using with the classic framework are available for OSF applications. This section describes differences you may encounter when working with OSF design assets in the administration interface.

- OSF widgets and layouts are associated with applications that have been deployed to the Commerce server. In order to view and access an application's widgets and layouts on the Design page, you must first select the application. See Select an application for more information.
- The Design Code Utility (on the Design page's Developer tab) is still available to download and use with classic framework applications, but it is not available to use with OSF applications. OSF application code is managed in a developer's local workspace. See Develop and Deploy Applications for more information.
- In the classic framework, global text snippets enable you to directly access your web store's customizable global text, which can include messages, labels and help tips. In OSF applications, shared locale strings are the equivalent of global text snippets. You cannot use the administration interface to edit an application's shared locale strings. These can be edited only in a local workspace. See Configure widgets in the administration interface for more information.
- In the classic framework, a theme is a set of styling configurations, including colors, fonts, and other design elements, that defines the overall look and feel of the online store. OSF applications do not use themes and so the Theme tab is not available for OSF applications. Design elements controlled by themes in classic storefront are controlled by CSS styling in OSF applications. CSS files must be configured by a developer in a local workspace. See Build a Widget for an example of creating styles CSS files in a local workspace.
- On a widget's details page, the Extend JavaScript, Go to Widget Code, and Download Source buttons are not available. This is because OSF widget code is created and edited in a developer's local workspace. See Create Custom Widgets for more information.

Select an application

Once a developer has built an OSF application and deployed it to the Commerce server environment, designers and other business users can work with it on the Design page in the administration interface.

Before you can work with the design assets in a deployed application, you must select the application on the Design page.

1. Log into Commerce and click the **Design** icon in the left pane. In the top-right corner of the Design page, you should see a dropdown menu where you can select a deployed OSF application.
   If the dropdown menu does not appear on the Design page, then no OSF applications have been deployed to the Commerce server you are logged into. See Deploy the application for information about deploying applications to a Commerce server.
2. Select an OSF application from the dropdown menu.
   You can now use the Design page tabs to view and work with the widgets, containers, and layouts that are part of the application you selected.

You will notice that the Design page looks different when an OSF application is loaded than it does when you work with classic design assets. Some tools and tabs you are used to seeing are not available because they cannot be used with OSF applications. For example:

- The Theme tab is not available for OSF applications because the design elements controlled by themes in classic storefront are controlled by CSS styling in OSF applications. (See Build a Widget for an example of creating styles CSS files in a local workspace.)
- On a widget's details page, the **Extend JavaScript**, **Go to Widget Code**, and **Download Source** buttons are not available. This is because OSF widget code is created and edited in a developer's workspace. See Create Custom Widgets for more information.

Configure widgets in the administration interface

Developers create and configure default settings for OSF widgets in a local workspace. Once an application is downloaded to a server environment, designers can update a number of settings for the application's widgets on the administration interface Design page.

You can customize the following widget settings in the administration interface:

- Configuration options for the widget. These options are included when a developer creates a widget and are specific to how you want that particular widget to behave on the storefront. For example,
- Text snippets let you customize the default text contained within an individual widget.
- The widget's display name and internal notes. See Design Your Store Layout for details about working with widgets in the administration interface.

Edit widget configuration options

Keep in mind that not every widget has configuration options.

Edit text snippets

Text snippets let you customize the component-specific text contained within an individual widget. You must also use text snippets to translate widget text into the different languages your store supports.

Note that you must redeploy your application for localization to take effect as the localization mechanism requires that Design Studio text snippets have been created to contain the translated strings.

Note: In this release, you cannot use the administration interface to edit an application's shared locale strings. These can be edited only in a local workspace. See Create locale information to learn more about shared strings.

Configure and display color swatches

Swatches let shoppers see images of a product in all its available colors without leaving the product details page.

When a shopper clicks a swatch, the main product image changes to display the product in the selected color. If applicable, other product information, such as price, description, and sizes, is also updated along with the product image.

To display swatches on a product details page, you must configure catalog, widget, and image settings.

Configure catalog settings

First, you must create a custom property on `productType` and set both its name and ID to `x_swatchMapping`. Make sure you select **Allow property to be searched** for this property in order to enable color swatches to appear on the product listing page.

You can create the custom property on either the Base product type or any custom product type that requires swatches. See Create and edit product types for more information about creating custom product-type properties.

Once you have created the custom property, modify `packages/tools/cli/config/search/index.js` to include `product.x_swatchMapping` under attributes and `sku.<colorProperty>` (for example `sku.color`) under `childRecordAttributes`, and run the command `yarn occ upload-search-config`. For more information, see Develop and Deploy Applications.

Finally, populate the product property (`x_swatchMapping`) on a product with mapping information. For example:

```
{"Gray":"gry","Red":"red","Pale Green/New Green":"pgn"}
```

The key must match the color values configured on the product. The value can be any short string.

To see the colors configured on the product, open the Commerce administration interface. Edit the product and click its SKUs tab. Under Configure Storefront Display, look for colors available for that product.

Configure the Product Variant Options widget

The Product Variant Options widget displays a product's variant options, such as color or size. To display color swatches, you configure the widget to display the options as swatch images. Before you configure the widget, make sure you have configured catalog settings and uploaded the images you want to use.

You can configure the widget on the Design page in the Commerce administration interface. Edit the following options:

- Color Swatch Variant Option (Variant Option To Be Displayed As Swatch Image): Leave the setting at the default value, which is Color.

- Product Property for Color Swatch Mapping (Product Property that contains swatches mapping information) The default value is `x_swatchMapping`. If you change the name from `x_swatchMapping`, an additional step must be taken to ensure this value is returned in search results. First, edit the file `packages/tools/cli/config/search/index.js` to change the `x_swatchMapping` in `product.x_swatchMapping` to your new property name, then run yarn `occ upload-search-config` to upload this new configuration.
- Color Swatch URL placeholder. (The URL template used for product color swatch images). You can modify this URL if you want to fetch JPG images instead of PNG images, or if you are looking for images in a different path. The default value is:

  ```
  /ccstore/v1/images/?source=/file/products/
  __productId__.__swatchKey__.png&outputFormat=JPEG&quality=0.8&height
  =__height__&width=__width__
  ```

  When the widget renders a product, it replaces the `__productId__` and `__swatchKey__` and finally resolves to a valid image URL. For example, the `__swatchKey__` is replaced with gry, red or pgn values from the following mapping on the product: `{"Gray":"gry","Red":"red","Pale Green":"pgn"}`.
- Overlay Image for Unavailable Color Swatch (The image used to overlay on a swatch image, if the swatch is invalid or unavailable) The default value is `/file/general/outofstock_overlay.png`.

Configure site-level settings

Upload the swatch images on the administration interface Media page, under Products. For example, the gray swatch image in media could be named `prod10004.gry.png`.

Now when the widget replaces `__productId__` and `__swatchKey__` parts of `swatchUrlPlaceholder` to a valid URL, it finds the image in the media library and renders it on the UI. The final URL might look like this:

```
/ccstore/v1/images/?source=/file/products/
prod10004.gry.png&outputFormat=JPEG&quality=0.8&height=40&width=40
```

where `__productId__ = prod` and `10004 __swatchKey__ = gry`

Work with Cart REST API Endpoints

The Store REST API endpoints for cart tasks have been enhanced to optimize cart and checkout flows for OSF widgets.

Cart REST API endpoints have been enhanced for use with OSF widgets. allow for more granular cart functions. The core business logic has been shifted to the server-side, and they support server-side shopping carts for anonymous shoppers. Note that with the exception of these Cart API updates, all Commerce REST APIs are backward-compatible and support both Storefront Classic and OSF storefronts.

Granular control is possible because the endpoints break an order down into individual, parts. Each part can be updated separately with the HTTP `PATCH` method. `PATCH` requests modify the specified resource by importing new or modified attributes into it. This allows for changes to individual parts of the order.

Order endpoints

This section describes the endpoints you can use to individually manage parts of an order. All requests for these endpoints require the `X-CCVisitorId` header parameter. This parameter specifies the ID provided by the Oracle Commerce Visit Service to uniquely identify the current visitor. It is required for every request to enable full endpoint capabilities for all shopper types.

The following endpoints manage the current order for both logged-in shoppers and guests.

Endpoint	HTTP Method	Path
`getIncompleteOrder` Returns incomplete order of the logged- in user	GET	/ccstore/v1/orders/current
`removeCurrentProfileIncompleteOrder` Removes the persisted order for the logged in user	DELETE	/ccstore/ v1/orders/current
`submitCurrentOrder` Submits current order; captures and authorizes payment before submitting order	POST	/ccstore/ v1/orders/current/ submit
`updateCurrentProfileOrder` Updates the persisted order for the logged in user	POST	/ccstore/ v1/orders/current
`updateIncompleteOrder` Update the current incomplete order	PATCH	/ccstore/ v1/orders/current

The following endpoints manage commerce items within the context of the current order.

Endpoint	HTTP Method	Path
`addCartItems` Add one or more items to the cart	POST	/ccstore/v1/orders/current/ items/add
`deleteCartItem` Delete the item in the cart for the given id	DELETE	/ccstore/v1/orders/current/ items/{id}
`getCartItems` Get all the items in the cart	GET	/ccstore/v1/orders/current/ items
`makeGiftWithPurchaseSelection` Enables selection of a gift item, based on qualified Gift with Purchase promotion	POST	/ccstore/v1/orders/current/ items/makeGWPSelection

`removeAllSelectableQuantity` Remove all selectable gift quantity.	POST	/ccstore/v1/orders/current/items/removeAllSelectableQuantity
`removeSelectableQuantity` Remove selectable gift quantity	POST	/ccstore/v1/orders/current/items/removeSelectableQuantity
`updateCartItem` Update the item in the cart specified by the item's ID.	PATCH	/ccstore/v1/orders/current/items/{id}

7

Develop for Performance

When you develop a storefront application, it is important to ensure that it is fast and responsive.

To help avoid performance issues, this chapter discusses potential problems to look out for when you code your pages and components, and how to fix these issues when you find them.

For more information about Oracle CX Commerce performance, see Extending Oracle CX Commerce. For more information about using React efficiently, go to `reactjs.org`.

Prevent `useEffect()` from executing unnecessarily

If a component includes the React `useEffect()` hook, it runs immediately after the component is rendered, and then each time any of its declared dependencies change. To avoid executing `useEffect()` unnecessarily, you should construct your code so that `useEffect()` runs only when it is actually needed.

In the following example, `useEffect()` is invoked after the component is first rendered, but the conditional statement ensures that the method's logic is executed only if any of the variant options change. The logic is skipped if no option is selected:

```
useEffect(() => {
  // Fire action only if variantOptions exist and an option is changed
    if (variantOptions && optionChanged) {
      // Get the latest selected item from the options
      const currentSelectedItem =
getCurrentSelectedItem(variantOptions);
      const variantOptionsLite = {};
      // Remaining code
    }
}, [variantOptions,variantOptionPermutations, optionChanged, widgetId,
action,variantToSkuLookup]);
```

Avoid unnecessary rerendering

As you write your code, be aware of situations where React components may be rerendered unnecessarily.

Common situations to avoid include:

- Rendering a component multiple times when a page is first loaded.
- Rerendering a component even though the props have not changed.

Avoid rerendering widgets when a page is initially loaded

When debugging a widget's `render()` method, check how many times the widget is rendered when a page is loaded. Ideally, a widget should be rendered only once when

the page is first loaded. If a widget is rendered more than once, try to identify why and rewrite your code to rectify this. For example, you might be modifying a state variable or props in the `useEffect()` hook, resulting in the widget rendering again.

In the following example, a state variable is declared and then its value is set in `useEffect()`:

```
const [quantity, setQuantity] = useState(null);

useEffect(() => {
  ...
  setQuantity(1);
  ...
}, []);
```

This results in the component being rendered twice. After the first render, `useEffect()` is executed and the quantity is changed from null to 1. The rerendering can be avoided by setting the value when the variable is declared. For example:

```
const [quantity, setQuantity] = useState(1);
```

Avoid rerendering with the same props

Although React updates only the DOM nodes that have changed, rerendering still takes some time. If your widget is rerendered frequently with the same set of props, you can use the `React.memo()` higher-order component to reduce unnecessary rendering.

In the following example, the `Child()` function would be forced to render by the `Parent()` function, but `React.memo()` is used to prevent this:

```
function Parent() {
 const currentDate = 10/10/2020;
 const dynValue = "Test";
  return(
    <Child dynValue ={dynValue} currentValue = {currentDate }/>
  );
}

function Child({ someFn, currentDate }) { ... };

export default React.memo(Child);
```

Note: If the component is rerendered due to changes in a Context object, using `React.memo()` may not improve performance.

Use `useCallback()` and `useMemo()` efficiently

Functions defined inside function components are recreated each time the component is rendered, resulting in referential inequality. This causes these components to rerender, in some cases unnecessarily.

To prevent rerendering components, you can use the `useCallback()` and `useMemo()` hooks. The `useCallback()` hook returns a memoized callback to maintain referential equality between renders of functions, and the `useMemo()` hook returns a memoized value to maintain referential equality between renders of values.

Note that `useCallback()` and `useMemo()` can result in more memory being allocated, so they must be used appropriately or they may actually reduce performance.

When to use useCallback()

This section describes situations where it is desirable to use `useCallback()`:

- Avoiding rerendering a child component when a function is recreated.
- Preventing `useEffect()` from creating an infinite loop.

Avoid rerendering a child component when a function is recreated

If you have a parent component that passes a callback function to a child component that uses `React.memo()`, rerendering the parent component recreates the function, which forces the child component to rerender, despite it using `React.memo()`. To avoid rerendering the child component, wrap the function with `useCallback()`. For example:

```
function Parent() {
 const currentDate = 10/10/2020;
 const someFn = () => {
     ...
  };

  return(
    <Child someFn={someFn} currentValue = {currentDate }/>
  );
}

function Child({ someFn, currentDate }) { ... };

 // Wrap function in useCallback() to prevent rerendering the child
component
 const someFn = useCallback(() => {
     ...

  }, [dependencies]);

export default React.memo(Child);
```

Prevent useEffect() from creating an infinite loop

If you invoke a function inside the `useEffect()` hook, `useEffect()` expects the function to be declared as a dependency. So each time the function changes, `useEffect()` runs. But running `useEffect()` causes the component to recreate the function, which then causes `useEffect()` to run again, resulting in an infinite loop. For example, the code below can cause an infinite loop:

```
const getItemFromCart = skuId => {
  const cartItems = Object.values(commerceItems);
  return cartItems.find(cartItem => cartItem.catRefId === skuId);
```

```
}

useEffect(() => {
  const selectedCartItem = getItemFromCart(skuId);
  setSomeState(someNewState);
   ...
}, [getItemFromCart]);
```

To avoid an infinite loop, wrap functions that run inside `useEffect()` with `useCallback()`. Wrapping the function in `useCallback()` ensures the function is not recreated, so `useEffect()` is not triggered to run again.

When to use useMemo()

If a function includes a complex calculation, you can avoid recomputing it by using the `useMemo()` hook. This hook returns a memoized value that is not recomputed unless the dependencies change.

In the following example, code that displays images is wrapped in `useMemo()` to prevent the images from being rerendered whenever the state changes:

```
<div
  className="ProductImageSlider__Wrapper"
  style={{transform: `translateX(${translateValue}px)`}}
  ref={slideWrapperEl}
  onTouchStart={handleTouchStart}
  onTouchMove={handleTouchMove}
  onClick={() => setPortalRenderedCallback(!portalRendered)}
>
  {useMemo(() => {
    return images.map(image => <Slide key={image} image={image} />);
  }, [images])}
</div>
```

Use `React.lazy()` to render components conditionally

You can use the `React.lazy()` function conditionally to ensure a component is rendered only if it is needed.

OSF makes extensive use of server-side rendering to speed up loading of pages. In some cases, however, a component will have code that should not be rendered in advance, such as a user interface element that is displayed conditionally. For example, a popup that is displayed when an item is added to the shopping cart should be rendered only when the shopper clicks the **Add to Cart** button.

In the following example, `LazyComponent` is rendered only if the shopper clicks the **Show Component** button:

```
const LazyComponent = React.lazy(() => import('./component'));

const [showComponent, setShowComponent] = useState(false);

const ParentComponent = () => {
   return (
```

```
      <div>
        <button onClick={() => setShowComponent(!showComponent)}> Show
Component
        </button>
        {showComponent && <LazyComponent />}
      </div>
     );
};
```

This approach can be used for CSS as well. Styles that apply only to the lazy-loaded component can be placed in the component-specific CSS file, so they will also be loaded only when needed.

Identify and optimize inefficient code

React uses the browser's User Timing API to mark events and measure the time between them.

For example, in the Chrome browser, the Developer tools Performance tab has a Timings section that uses the User Timing API. You can use the Timings section to identify tasks that are CPU-intensive, so you can target these tasks for potential improvements.

It is a good idea to run your browser in private browsing mode (for example, an incognito window in Chrome) when running performance timings. This ensures that results are not affected by cached content and that most extensions are disabled.

In addition to your browser's native developer tools, you can also install the React DevTools profiler and use it to identify widgets and other components that take a long time to run or are rendered multiple times.

Remove unnecessary CSS

It is a good idea to eliminate unused CSS, as it can make your pages load more slowly.

When a page loads, the browser must construct the CSSOM before it can start rendering content. If there is unused CSS, the time it takes to load delays rendering.

To help find unused CSS, you can use the Coverage tab in Chrome developer tools. Or you can also use one of the many browser extensions or plugins available for finding unused CSS, and in some cases, removing it automatically.

Note: Be careful not to remove CSS that is loaded and used conditionally. Styles that are loaded as part of a lazy-loaded component may be reported as unused if the component is not present on the page, but they will be needed when the component actually loads.

You can also trim the size of the CSS by omitting values for properties whose defaults you want to accept. Consider the following example:

```
.ABC__XYZ form {
  display: flex;
  flex-direction: row;
}
```

The default value of the `flex-direction` property is `row`, so you can omit it, and just specify the following instead:

```
.ABC__XYZ form {
  display: flex;
}
```

8

Build Payment Integrations

You can use tools that Commerce provides to create custom integrations with payment gateways.

Commerce includes built-in integrations with CyberSource, PayPal, and Chase Paymentech gateways. These integrations are described in Using Oracle CX Commerce. Note, however, that the built-in PayPal and CyberSource integrations support storefronts built using the Storefront Classic framework only; they do not work with storefronts built using OSF.

You can build custom integrations using the Generic Payment webhook, the Store REST API endpoints, and storefront components you create in OSF. This chapter describes how to build payment gateway integrations that work with OSF storefronts.

Understand payment integrations

You can use tools that Commerce provides to create custom integrations with payment gateways.

Oracle CX Commerce provides a framework for building custom integrations with payment gateways. The framework supports integrating with a variety of different payment types, including:

- credit cards
- gift cards
- invoices and purchase orders
- cash
- store credit
- loyalty points

A key aspect of these integrations is the use of payment webhooks. When a shopper places an order, the `POST /ccstore/v1/orders/current/submit` endpoint triggers a call by a webhook. The webhook sends a request to a specified URL and includes the payment-related data it received from the endpoint. The target system processes the payment data and then sends a response that indicates success or failure and other information about the transaction.

Except for loyalty points, the payment types listed above are all supported through the Generic Payment webhook. Loyalty point payments are supported through the Custom Currency webhook.

In addition to supporting integrations that can handle one or more of these payment types directly, the Generic Payment webhook can also be used for implementations in which Commerce delegates the payment handling to an external web payment system such as Amazon Payments and PayPal.

Creating a custom payment integration involves a number of steps. These include:

- Creating a gateway extension to configure access to the gateway. The extension defines configuration options that merchants can set on the **Payment Gateways** tab of the **Payment Processing** page in the administration interface.
- Developing storefront checkout pages that work with the integration. You may need to customize your payment widgets to supply specific information to the payment or order endpoints.
- Configuring the Generic Payment webhook to send payment data to the gateway.

Depending on the gateway and payment providers, there may be additional steps involved in creating an integration. You need to configure settings on the gateway itself, and if your integration handles credit cards, you must ensure it is PCI-compliant. Also, you will typically need to create a custom payment integration service for routing transaction data from the Generic Payment webhook to the gateway and mapping the webhook data properties to fields used by the gateway.

One way to create a payment integration service is to write a server-side extension (SSE) that functions as a translation and routing layer. For example, the sample Worldpay integration available on the Cloud Customer Connect site is designed to work like this:

1. When a shopper or agent submits or modifies an order, a payment transaction is initiated (an authorization, void, or refund, depending on what the shopper or agent is trying to do).
2. The Generic Payment webhook sends a POST request to a REST endpoint that a server-side extension implements.
3. The server-side extension's JavaScript code transforms the JSON from the Generic Payment webhook into the XML format that Worldpay supports, and sends it to Worldpay.
4. Worldpay processes the transaction, and sends an XML response to the server-side extension.
5. The server-side extension's JavaScript code transforms the XML response to the JSON format that Commerce supports, and sends the JSON to the Commerce server as the Generic Payment webhook response.
6. The Commerce server returns the results of the transaction to the storefront.

Supported payment methods and transaction types

The Generic Payment webhook supports a wide range of options for integrating with gateways that handle a variety of different 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.

Method	Supported Transaction Types
`card` – credit card payment	`authorization` – approve payment for an order `void` – cancel an order or a payment `refund` – issue a credit to the shopper after a return

Method	Supported Transaction Types
`cash` – cash payment	`initiate` – create an order to be completed later `cancel` – cancel an order or a payment
`physicalGiftCard` – pay with gift card	`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` – generate an invoice to bill the buyer	`authorization` – approve payment for an order
`storeCredit` – pay with store credit	`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
`generic` – supports more complex payment logic, such as is needed for integrating with web checkout systems	`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

Create an extension for a gateway integration

You use payment gateway extensions to specify settings for a gateway integration, and to create controls in your administration interface for configuring those settings.

These settings appear on 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 CX 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.

`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": "2021-01-01",
  "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`.
- `processors` – Including the `card3ds` processor adds support for 3D-Secure credit card payments.

In the following example, the Demo Payment Provider is configured to support credit card `authorization`, `void`, and `refund` transactions, and cash `initiate` and `cancel` transactions:

```
{
  "provider": "Demo Payment Provider",
  "paymentMethodTypes": ["card", "cash"],
  "transactionTypes": {
    "card": ["authorization", "void", "refund"],
    "cash": ["initiate", "cancel"]
  },
  "processors": {
    "card": "card3ds"
  }
}
```

`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": "paymentMethodTypes",
      "type": "multiSelectOptionType",
      "name": "paymentMethodTypes",
      "required": true,
      "helpTextResourceId": "paymentMethodsHelpText",
      "labelResourceId": "paymentMethodsLabel",
      "defaultValue": "card",
      "displayAsCheckboxes": true,
      "public": true,
      "options": [
         {
           "id": "cash",
           "value": "cash",
           "labelResourceId": "cashPayLabel"
         },
         {
           "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 `type` attribute of the `paymentMethodTypes` property is set to `multiSelectOptionType`, which means that multiple methods can be selected (for example, `card` 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.
- 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",
    "cashPayLabel: "Cash In-Store Payment",
    "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.
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, there should be checkboxes for enabling payment options under Preview Configuration, Agent Configuration, and Storefront Configuration.

If you enable a credit card payment gateway for a site, make sure that other credit card gateways are disabled on that site. Only one credit card gateway integration should be enabled for an individual site.

Configure the Generic Payment webhook

When you create an integration for a payment gateway, the integration uses the Generic 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 **Generic 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.

Send transaction data to a payment gateway

When payment transaction data is sent to a payment provider by the Generic Payment webhook, the provider processes the payment and returns data 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. The set of properties differs depending on the payment method and the type of transaction. For example, the following is a sample payload for 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",
   }
}
```

The request and response payload properties for the payment methods Commerce supports are described in detail in Extending Oracle CX Commerce.

Send custom properties

Depending on the payment method and the provider you integrate with, there may be additional payment data that you need to send. If so, you can include this data in the `POST /ccstore/v1/orders/current/submit` endpoint 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"
  }
],
...
```

The custom properties from the endpoint 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 also 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. In addition, the webhook can return a `customPaymentProperties` object that specifies a list of the properties in the `additionalProperties` object that should be returned to the storefront in the endpoint response. 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",
...
```

Look up payment configurations

Commerce provides a REST endpoint that your OSF applications can use to return configuration information about your sites' payment gateways.

You can use the `getPaymentConfigurations` endpoint in the Store API to return information about the payment gateways for a site. For example:

```
GET /ccstore/v1/merchant/paymentConfigurations  HTTP/1.1
x-ccsite: 100002
```

The response is similar to the following:


```
{
    "settings": [
      {
        "paymentMethods": "card",
        "enabledForScheduledOrder": false,
        "enabledForApproval": false
      }
    ],
    "paymentMethods": [
       "physicalGiftCard",
       "card"
    ],
    "links": [
        {
          "rel": "self",
          "href": "http://myserver.example.com:7003/ccstore/v1/merchant/
paymentConfigurations"
        }
    ]
}
```


Note that the endpoint returns configuration information only for gateway integrations that are enabled on the site. If a site is not specified with the `x-ccsite` header, the endpoint returns the gateway configuration for the default site. The endpoint returns only those properties whose `public` attribute is set to `true` in the gateway extension's `config.json` file.