ORACLE®

COMMERCE

Version 11.3

Platform-Guided Search Integration Guide

Platform-Guided Search Integration Guide

Product version: 11.3
Release date: 04-28-17
Document identifier: EndecaIntegrationGuide1704181210

Table of Contents

1 Introduction

The Oracle Core Commerce Platform - Guided Search integration enables customers of the Oracle Commerce Platform and Oracle Commerce Guided Search to index data from GSA repositories in Oracle Commerce MDEX Engines, where it can then be queried and the results can be displayed on commerce sites. This document describes how to configure Oracle Commerce Platform indexing and querying components to work with Guided Search.

This chapter provides an overview of installing and configuring a Guided Search integration environment, and provides a brief description of the Guided Search integration modules. It includes the following sections:

Note that Oracle Commerce Reference Store makes extensive use of the Guided Search integration to demonstrate the use of both the Oracle Commerce Platform and Oracle Commerce Guided Search on commerce sites, and in some cases extends the capability of the integration. See the Commerce Reference Store documentation for more information.

Installation Requirements

The Guided Search integration requires that the Oracle Commerce Platform and Oracle Commerce Guided Search (with or without Oracle Commerce Experience Manager) be installed in your environment. We also suggest that you initially install Oracle Commerce Reference Store, so that you have an application and data to work with as you familiarize yourself with the integration.

For information about installing the Oracle Commerce Platform software, see the *Platform Installation and Configuration Guide*. For information about installing Commerce Reference Store, see the *Commerce Reference Store Installation and Configuration Guide*. For information about installing Oracle Commerce Guided Search software, see the *Oracle Commerce Guided Search Getting Started Guide* and other related Guided Search installation documentation.

Creating the EAC Applications

To create a Guided Search EAC application to integrate with the Oracle Commerce Platform, use the CAS-based deployment template described in the *Oracle Commerce Guided Search Administrator's Guide*. (If your Oracle Commerce Platform environment is based on Oracle Commerce Reference Store, you can use the CAS-based deployment template that is included with it.) The deployment template includes a script that creates CAS (Content Acquisition System) record stores that the Guided Search integration submits records to. The naming convention for these record stores is:

```
applicationName-recordStoreType
```

For an application named `ATGen` that indexes GSA repository data, the record stores are:

- `ATGen-data` -- Holds data records representing repository items such as products and SKUs.
- `ATGen-dimvals` -- Holds dimension value records generated from the category hierarchy and from the hierarchy of repository item types.

The Guided Search integration includes classes and components that create records and write them to these records stores. In addition, the integration includes classes and components that create schema records, convert them to Configuration Import API objects, and submit these objects to the Endeca Configuration Repository.

Note that there is also an `ATGen-prules` record store, which is used to create Guided Search precedence rules. The integration does not provide a way to create precedence rules or write to this record store, but you can create precedence rules directly in Guided Search. See the Guided Search documentation for information about creating precedence rules.

Using an Older Deployment Template

This manual assumes you are using a CAS-based deployment template for your EAC applications, as mentioned above. If you are creating new EAC applications, it is highly recommended that you use this type of deployment template.

If you have EAC applications created in an earlier release, they may be using an older Forge-based deployment template such as the one described in the *Oracle Commerce Deployment Template Module for Product Catalog Integration Usage Guide*. This type of deployment template uses CAS for its record and schema storage, and Forge to generate configuration and transform records on import. (CAS-style deployment templates also use CAS for record storage, but store schema configuration in the Endeca Configuration Repository, and do not use Forge at all.) Forge-based applications are still supported, but require some differences in the configuration of Oracle Commerce Platform components, as they require record output in a somewhat different format from applications that use a CAS-based template.

If you do have existing EAC applications that use a Forge-based deployment template, you can recreate your EAC applications with a CAS-based deployment template. If you instead continue to use applications based on the older-style template, you will need to reconfigure several Oracle Commerce Platform components, as described in *Appendix A: Support for Older Deployment Templates* (page 175). In addition, you will need to follow the Oracle Commerce Guided Search migration procedure described in the *Oracle Commerce Guided Search Tools and Frameworks Migration Guide*.

Determining the Number of EAC Applications to Create

To integrate Guided Search with your Oracle Commerce Platform environment, you must create at least one EAC application and a corresponding MDEX. If you have data in multiple languages or multiple sites, the number

of EAC applications you have depends on your approach to indexing these languages and sites. To implement a specific approach, you need to configure a *routing strategy*, which controls the logic for directing data for indexing and querying to specific EAC applications. See the *Routing* (page 9)chapter for information about configuring routing strategies.

Provisioning the EAC Applications

You must provision each EAC application individually by running the `initialize_services.sh|bat` script found in the application's `/control` directory. Therefore, if you have three EAC applications, you must invoke the script three times. The `initialize_services.sh` script is found in the following location: `/endeca/EAC-application-directory/your-application/control/`.

Configuring the Oracle Commerce Platform Server Instances in CIM

You can configure your Oracle Commerce Platform server instances for a Guided Search integration environment using the Configuration and Installation Manager (CIM). The options you must configure are described below.

Product Selection

To configure your server instances to use the Guided Search integration, select the Guided Search integration and the Oracle Commerce Platform in the Product Selection menu. If your installation includes Oracle Commerce Reference Store, you can select Oracle Commerce Reference Store instead. Your server installations will automatically include the Oracle Commerce Platform and the Guided Search integration, because Commerce Reference Store requires them.

Oracle Commerce Platform Server Instance Creation

During your Oracle Commerce Platform server instance configuration, you must provide information about your Guided Search environment so that the Oracle Commerce Platform server instance can communicate with Guided Search. The required settings and their defaults are provided in the table below:

Setting	Default
CAS hostname	`localhost`
CAS port	8500
EAC hostname	`localhost`
EAC port	8888
EAC base application name	ATG

Setting	Default
Fully qualified Workbench host name, including domain	n/a
Workbench port	8006
Default MDEX host name	`localhost`
Default MDEX port number	15000

After your Oracle Commerce Platform server instances are configured in CIM, start them up in preparation for indexing.

Configuring the ApplicationConfiguration Component

The `atg.endeca.configuration.ApplicationConfiguration` class provides a central place for configuring various global settings, including language configuration options and application naming. The Guided Search integration includes a component of this class, `/atg/endeca/ApplicationConfiguration`. The following are key properties of this component:

locales

An array of the locales to generate records for. To generate records in multiple languages, you specify the locales using this property. For example:

```
locales=en_US,fr_FR
```

Note that only one set of records is generated for each language. So, for example, if you specify multiple locales where the language is French (for example, `ca_FR` and `fr_FR`), only one set of French records is generated.

defaultLanguageForApplications

The two-letter code of the default language for the applications. This should be null (the default) if you are using a separate EAC application for each language. See the *Routing* (page 9)chapter for more information about when this property should be set.

baseApplicationName

The base string used in constructing the EAC application names. The default setting is `ATG`. You can override the default when you use CIM to configure your Oracle Commerce Platform environment.

keyToApplicationName

A map of application keys to application names. You can use this property to override the default application naming convention. See the *Routing* (page 9)chapter for more information about the naming convention and when this property should be set.

defaultApplicationKey

The application key to use if the current application cannot otherwise be determined. An array of the keys is stored in the read-only `applicationKeys` property. If there is a separate application for each language, the first key listed in the `applicationKeys` property is the default, unless you change the default by explicitly setting the `defaultApplicationKey` property to a different key.

If there is only one EAC application, you do not need to set this property; it will automatically be set to `default`.

applicationKeyToMdexHostAndPort

A map where the keys identify each EAC application and the values specify the host names and port numbers for the MDEX engines associated with each application. See Connecting to the Workbench and MDEX (page 100) in the *Query Integration* (page 85) chapter for more information about this property.

applicationRoutingStrategy

A component of a class that implements the `atg.endeca.configuration.ApplicationRoutingStrategy` interface. The specific class determines the logic for directing records to EAC applications for indexing and for directing queries to those applications. See the *Routing* (page 9)chapter for more information.

workbenchHostName

The fully qualified host name, including the domain, of the machine running the Oracle Commerce Workbench. You can specify this setting when you use CIM to configure your Oracle Commerce Platform environment.

workbenchPort

The port number for accessing the Oracle Commerce Workbench. The default setting is `8006`. You can override this default when you use CIM to configure your Oracle Commerce Platform environment.

credentialStoreManager

The component that manages the credential store where login credentials for the Oracle Commerce Workbench are stored. By default, this property is set to:

```
credentialStoreManager=\
  /atg/dynamo/security/opss/csf/CredentialStoreManager
```

The credential store is implemented using the Credential Security Framework (CSF) of Oracle Platform Security Services (OPSS). You can create credentials using CIM, and add or delete credentials using the page for the `CredentialStoreManager` component in the Dynamo Server Admin. For more information about using CSF with Oracle Commerce, see the *Platform Programming Guide*.

workbenchCredentialStoreMapName

The name of the credential store map used to store workbench login credentials. By default, this is set to:

```
workbenchCredentialStoreMapName=endecaToolsAndFrameworks
```

workbenchCredentialStoreKeyName

The name of the key used to retrieve workbench login credentials from the credential store map. By default, this is set to:

```
workbenchCredentialStoreKeyName=ifcr
```

recordIdName

The output name used in records for the `$docId` property. Set by default to `record.id`. The value of this property is used as the unique identifier for a record. See Automatically Included Properties (page 50) in the *Configuring EndecaIndexingOutputConfig Definition Files* (page 49) chapter for more information about this property.

recordSourceName

The output name used in records for the `$repository.repositoryName` property. Set by default to `record.source`. The value of this property identifies the name of the source repository. See Automatically Included Properties (page 50) in the *Configuring EndecaIndexingOutputConfig Definition Files* (page 49) chapter for more information about this property.

recordTypeName

The output name used in records for the `$itemDescriptor.itemDescriptorName` property. Set by default to `record.type`. The value of this property identifies the repository item type used to generate the record. See Automatically Included Properties (page 50) in the *Configuring EndecaIndexingOutputConfig Definition Files* (page 49) chapter for more information about this property.

Configuring Sites in a Multisite Environment

In multisite environments that use the Guided Search integration, site configuration exists in both Site Administration and in the EAC application. These configurations have to match each other; for example, if you have three sites configured in Site Administration, you should have three corresponding sites configured for your EAC application.

To create a mapping between the sites, you use the Guided Search Site ID property in Site Administration. This property is located on the Site tab when you view a site's details in Site Administration. For each site defined in Site Administration, enter the ID of the corresponding site as defined in the EAC application.

Transaction Timeout and Datasource Connection Pool Settings

Depending on your application server, you may need to increase the transaction timeout and datasource connection pool settings in order for indexing to run successfully.

Increasing the Transaction Timeout

All supported application servers roll back transactions that do not complete in a specified number of seconds. These transaction rollbacks can cause indexing jobs to fail. If your indexing process fails, try increasing the

transaction timeout setting to 300 seconds or more. For details on changing your transaction timeout, see *Setting the Transaction Timeout on WebLogic*, *Setting the Transaction Timeout on JBoss*, or *Setting the Transaction Timeout on WebSphere* in the *Platform Installation and Configuration Guide*.

Increasing the Datasource Connection Pool

Oracle recommends setting the data source connection pool maximum capacity to 30 or greater for all of your data sources. For information on setting the data source connection pool maximum capacity, refer to your application server's documentation.

Oracle Commerce Platform Modules

The main Guided Search integration modules are:

Module	Description
`DAF.Endeca.Index`	Includes the necessary classes for exporting data to CAS record stores and triggering indexing, along with associated configuration.
`DAF.Endeca.Index.Versioned`	Adds configuration for running on an ATG Content Administration instance. This module adds basic record generation configuration for ATG Content Administration servers, including a deployment listener.
`DAF.Endeca.Assembler`	Contains classes and configuration for creating an Assembler instance that has access to the data in your application's MDEX engines. Also provides classes for querying the Assembler for data and managing the content returned.
`DCS.Endeca.Index`	Configures components for creating CAS data records from products in the catalog repository and dimension-value records from the category hierarchy.
`DCS.Endeca.Index.SKUIndexing`	Modifies configuration so that CAS data records are generated based on SKUs rather than products.
`DCS.Endeca.Index.Versioned`	Adds Commerce-specific configuration for running on an ATG Content Administration instance.
`DCS.Endeca.Assembler`	Contains Commerce-specific configuration for query-related components.

Additional modules for indexing Web content in the content management repository are described in the *Indexing the Content Management Repository* (page 73) chapter.

Note that when you assemble an application that includes any of the modules listed in the table above, other modules they have dependencies on, such as `DAF.Endeca.Base` or `DAF.Search.Index`, are automatically

included in the EAR file as well. In addition, some of the Guided Search-specific modules pull in classes from other search modules (without including the modules in their entirety) through the `ATG-Class-Path` entries in their manifest files.

2 Routing

Routing is the process of directing records for indexing to specific EAC applications and their corresponding MDEX instances, and ensuring that queries (for example, search terms or dimension selections) are directed to the correct EAC applications as well.

The Guided Search integration supports a variety of routing options. These differ by how data for indexing is divided up among EAC applications, depending on criteria such as language or site.

This chapter describes the various routing options available in the integration and how to configure them. It includes the following sections:

Overview of Routing

Routing involves mapping languages or sites (or both) to specific EAC applications, based on criteria that you specify. For example, if you have a site that is in French and English, you may want to direct the French records to one EAC application for indexing, and direct the English records to a different EAC application. After indexing, when a user enters a search term or selects a dimension, the same routing logic directs the query to the correct EAC application (for example, French queries to the EAC application that indexes the French records).

Different routing options are supported by different classes that implement the `atg.endeca.configuration.ApplicationRoutingStrategy` interface. The class you use determines the logic for directing records to EAC applications for indexing and for directing queries to those applications.

In addition to `ApplicationRoutingStrategy`, there is another key routing interface, `atg.endeca.configuration.RoutingObjectAdapter`. Classes that implement this interface are responsible for determining the current site or language so the `ApplicationRoutingStrategy` class can then direct data to the correct application. There are two subinterfaces of `RoutingObjectAdapter`:

- `atg.endeca.index.configuration.ContextRoutingObjectAdapter` – Classes that implement this interface are used during indexing to obtain the current indexing context from the `atg.repository.search.indexing.Context` object.
- `atg.endeca.assembler.configuration.RequestRoutingObjectAdapter` – Classes that implement this interface are used during querying to obtain the current querying context from the `DynamoHttpServletRequest` object and other objects that hold request-specific state, such as `SiteContext` objects.

The `ApplicationRoutingStrategy` and `RoutingObjectAdapter` classes are discussed below. To specify the routing classes you want to use, see Configuring Routing (page 11).

ApplicationRoutingStrategy

Different routing options are supported by different classes that implement the `atg.endeca.configuration.ApplicationRoutingStrategy` interface. There are three main classes in the `atg.endeca.configuration` package that implement this interface:

- `SingleApplicationRoutingStrategy`
- `SiteApplicationRoutingStrategy`
- `GroupingApplicationRoutingStrategy`

The various routing options and the `ApplicationRoutingStrategy` classes that support them are summarized in the following table:

Option	Description	Routing Strategy
One EAC application	All records are directed to a single EAC application, regardless of the language or site.	`SingleApplicationRoutingStrategy`
One language per EAC application	Each language's records are directed to a separate EAC application. If there are multiple sites, all records for an individual language are directed to the language's EAC application, regardless of the site.	`SingleApplicationRoutingStrategy`
One site per EAC application	Each site's records are directed to a separate EAC application. If there are multiple languages, all records for an individual site are directed to the site's EAC application, regardless of the language.	`SiteApplicationRoutingStrategy`
One combination of site and language per EAC application	Records for each combination of site and language are directed to a separate EAC application. For example, if there are five sites and three languages, there may be as many as 15 EAC applications. (There may be fewer if not all of the sites include all three languages.)	`SiteApplicationRoutingStrategy`
Arbitrary grouping of sites per EAC application	For example, there are five sites, with records for two sites directed to one EAC application, and records for the other three sites directed to a second EAC application. If a site has multiple languages, all records for the site are directed to the site's EAC application, regardless of the language.	`GroupingApplicationRoutingStrategy`

RoutingObjectAdapter

The `RoutingObjectAdapter` interface is responsible for obtaining context information that the `ApplicationRoutingStrategy` uses to route data for indexing and querying. This interface has two subinterfaces, `ContextRoutingObjectAdapter` (for indexing) and `RequestRoutingObjectAdapter` (for querying). These subinterfaces differ mainly in terms of where they obtain the context information.

ContextRoutingObjectAdapter

Classes that implement the `atg.endeca.index.configuration.ContextRoutingObjectAdapter` interface use the `atg.repository.search.indexing.Context` object to determine the current indexing context. The `Context` object provides a central place to store data and state information, such as the current site and locale, during an indexing job.

There are three `ContextRoutingObjectAdapter` classes in the `atg.endeca.index.configuration` package, which correspond to the three `ApplicationRoutingStrategy` classes discussed above:

- `SingleContextRoutingObjectAdapter`
- `SiteContextRoutingObjectAdapter`
- `GroupingContextRoutingObjectAdapter`

So, for example, if you are using `SiteApplicationRoutingStrategy` as your routing strategy, you should use `SiteContextRoutingObjectAdapter` for determining the indexing context.

RequestRoutingObjectAdapter

Classes that implement the `atg.endeca.assembler.configuration.RequestRoutingObjectAdapter` interface use the `atg.servlet.DynamoHttpServletRequest` object and other objects that hold request-specific state, such as `SiteContext` objects, to determine the current context for querying.

There are three `RequestRoutingObjectAdapter` classes in the `atg.endeca.assembler.configuration` package, which correspond to the three `ApplicationRoutingStrategy` classes discussed above:

- `SingleRequestRoutingObjectAdapter`
- `SiteRequestRoutingObjectAdapter`
- `GroupingRequestRoutingObjectAdapter`

So, for example, if you are using `SiteApplicationRoutingStrategy` as your routing strategy (and `SiteContextRoutingObjectAdapter` for determining the indexing context), you should use `SiteRequestRoutingObjectAdapter` for determining the querying context.

Configuring Routing

Once you have determined the routing strategy you want to use (see Overview of Routing (page 9)), you specify it by setting the `applicationRoutingStrategy` property of `/atg/endeca/ApplicationConfiguration` to a component of one of the `ApplicationRoutingStrategy` classes mentioned above. Other configuration depends on which routing strategy class you specify.

This section describes how you configure different routing strategies and associated components. For more information about configuring the `IndexingApplicationConfiguration` component,

see the IndexingApplicationConfiguration Component (page 27) section of the *Configuring the Indexing Components* (page 27) chapter. For more information about configuring the `AssemblerApplicationConfiguration` component, see the AssemblerApplicationConfiguration Component (page 100) section of the *Query Integration* (page 85) chapter.

SingleApplicationRoutingStrategy

Use the `SingleApplicationRoutingStrategy` class if you have a single EAC application, or if you have a separate EAC application for each language. In either case, routing is not affected by sites; for a given language, records for all sites are directed to the EAC application associated with that language.

To use `SingleApplicationRoutingStrategy`, set the `ApplicationConfiguration.applicationRoutingStrategy` property to null. This property is null by default, so you can leave it unset or set it to null explicitly. If `applicationRoutingStrategy` is null, an instance of the `SingleApplicationRoutingStrategy` class is created automatically.

Similarly, set the `IndexingApplicationConfiguration.routingObjectAdapter` and `AssemblerApplicationConfiguration.routingObjectAdapter` properties to null to automatically create instances of the `SingleContextRoutingObjectAdapter` and `SingleRequestRoutingObjectAdapter` classes.

Additional configuration differs depending on whether you have a single EAC application for all languages or a separate EAC application for each language.

Single EAC Application

If all languages are being handled by the same EAC application, set the `ApplicationConfiguration` component's `defaultLanguageForApplications` property to the two-letter language code for the default language. For example:

```
defaultLanguageForApplications=en
```

Note that you should set this property even if your data is in only one language.

By default, the name of the application is assumed to be formed by concatenating the value of the `ApplicationConfiguration` component's `baseApplicationName` property and the value of the `defaultLanguageForApplications` property. For example, if `baseApplicationName` is `ATG` and `defaultLanguageForApplications` is en, the Oracle Commerce Platform assumes the name of the EAC application is `ATGen`. If your application has a different name from the default, specify the name by setting the `ApplicationConfiguration` component's `keyToApplicationName` property:

```
keyToApplicationName=\
  default=application-name
```

For example, if the name of the EAC application is `MyApp`:

```
keyToApplicationName=\
  default=MyApp
```

Note that the key is `default` only when there is a single EAC application.

One EAC Application per Language

If each language is being handled by a separate EAC application, set the `defaultLanguageForApplications` property to null. This property is null by default, but if it has been subsequently set to a non-null value, you must explicitly set it back to null:

```
defaultLanguageForApplications^=/Constants.null
```

The two-letter language codes are used as the application keys for routing. By default, the name of each application is assumed to be the value of the `ApplicationConfiguration` component's `baseApplicationName` property plus the two-letter language code. For example, if `baseApplicationName` is `ATG` and you have records in English, German, and Spanish, the Oracle Commerce Platform assumes the names of the EAC applications are `ATGen`, `ATGde`, and `ATGes`. If your applications are named differently, use the `ApplicationConfiguration` component's `keyToApplicationName` property to explicitly map the language codes to your application names. For example:

```
keyToApplicationName=\
  en=MyEnglishApp,\
  es=MySpanishApp,\
  de=MyGermanApp
```

SiteApplicationRoutingStrategy

Use the `SiteApplicationRoutingStrategy` class if you have a separate EAC application for each site (with all languages in a given site being handled by that site's EAC application), or if you have a separate EAC application for each combination of site and language.

To use `SiteApplicationRoutingStrategy`, set the `ApplicationConfiguration` component's `applicationRoutingStrategy` property as follows:

```
applicationRoutingStrategy=\
  /atg/endeca/configuration/SiteApplicationRoutingStrategy
```

In addition, to ensure that separate records are created for each site, you need to add the `UniqueSiteVariantProducer` to the `variantProducers` property of each `EndecaIndexingOutputConfig` component. For example

```
variantProducers+=/atg/search/repository/UniqueSiteVariantProducer
```

`EndecaIndexingOutputConfig` components are discussed in the next several chapters of this manual. For information about variant producers, including `UniqueSiteVariantProducer`, see the Using Variant Producers (page 63)section of the *Customizing the Output Records* (page 61) chapter.

Set the `routingObjectAdapter` property of the `/atg/endeca/index/IndexingApplicationConfiguration` component to specify the `ContextRoutingObjectAdapter` component to use:

```
routingObjectAdapter=\
```

```
  /atg/endeca/index/configuration/SiteContextRoutingObjectAdapter
```

Set the `routingObjectAdapter` property of the `/atg/endeca/assembler/AssemblerApplicationConfiguration` component to specify the `RequestRoutingObjectAdapter` component to use:

```
routingObjectAdapter=\
  /atg/endeca/index/configuration/SiteRequestRoutingObjectAdapter
```

Additional configuration differs depending on whether you have a single EAC application for each site or a separate EAC application for each combination of site and language.

One EAC Application per Site

If you have one EAC application per site (with all languages for each individual site being handled by that site's EAC application), set the `ApplicationConfiguration` component's `defaultLanguageForApplications` property to the two-letter language code for the default language. For example:

```
defaultLanguageForApplications=it
```

Note that you should set this property even if your data is in only one language.

The site IDs are used as the application keys for routing. The value of the `SiteApplicationRoutingStrategy` component's `applicationNameFormatString` property specifies the default naming scheme for the EAC applications. The value of this property is a format string in which 0 is the value of the `baseApplicationName` property, 1 is the site ID, and 2 is the two-letter language code. In this case, since routing does not take language into account, the 2 should be omitted, and the property should be set to:

```
applicationNameFormatString={0}{1}
```

Suppose, for example, that `baseApplicationName` is `ATG` and you have three sites whose IDs are `storeIT`, `storeDE`, and `storeFR`. The default names of the EAC applications are `ATGstoreIT`, `ATGstoreDE`, and `ATGstoreFR`. If your applications are named differently, use the `ApplicationConfiguration` component's `keyToApplicationName` property to explicitly map the site IDs to your application names. For example:

```
keyToApplicationName=\
  storeIT=MyItalyStore,\
  storeDE=MyGermanyStore,\
  storeFR=MyFranceStore
```

Separate EAC Application for each Combination of Site and Language

If you have a separate EAC application for each combination of site and language, set the `defaultLanguageForApplications` property to null. This property is null by default, but if it has been subsequently set to a non-null value, you must explicitly set it back to null:

```
defaultLanguageForApplications^=/Constants.null
```

Since in this case routing must take into account both site and language, each application key is formed by concatenating the site ID with the language code, separated by the underscore character

(_). Similarly, the application names need to reflect the sites and languages, so the value of the SiteApplicationRoutingStrategy component's applicationNameFormatString property should be set to:

```
applicationNameFormatString={0}{1}{2}
```

Suppose, for example, that baseApplicationName is ATG and you have Canada site whose ID is storeCA. If the site has two languages, French and English, the default names for the corresponding EAC applications would be ATGstoreCAfr and ATGstoreCAen, and the keys for these applications would be storeCA_fr and storeCA_en.

Note that if you have multiple sites and multiple languages, but not all of the sites support all of the languages, SiteApplicationRoutingStrategy does not create keys for the missing combinations. Records are generated only for valid combinations of sites and languages.

If your applications do not use the default naming scheme, use the ApplicationConfiguration component's keyToApplicationName property to explicitly map the keys to your application names. For example:

```
keyToApplicationName=\
  storeCA_fr=MyCanadaStoreFrench,\
  storeCA_en=MyCanadaStoreEnglish,\
  storeDE_de=MyGermanyStoreGerman,\
  storeDE_en=MyGermanyStoreEnglish,\
  storeFR_fr=MyFranceStoreFrench
```

GroupingApplicationRoutingStrategy

The GroupingApplicationRoutingStrategy class allows more flexible groupings of sites than SiteApplicationRoutingStrategy does. For example, with GroupingApplicationRoutingStrategy, you can have three sites handled by one EAC application and two other sites handled by a second EAC application. If a site has multiple languages, all records for the site are directed to the site's EAC application, regardless of the language.

To use GroupingApplicationRoutingStrategy, set the ApplicationConfiguration component's applicationRoutingStrategy property as follows:

```
applicationRoutingStrategy=\
  /atg/endeca/configuration/GroupingApplicationRoutingStrategy
```

Since this strategy may involve having multiple languages in a single EAC application, you need to set the ApplicationConfiguration component's defaultLanguageForApplications property to the two-letter language code for the default language. For example:

```
defaultLanguageForApplications=fr
```

Note that you should set this property even if your data is in only one language.

The mapping of EAC applications to sites is done through the GroupingApplicationRoutingStrategy component itself, rather than the ApplicationConfiguration component. Therefore, you must set the keyToApplicationName properties of the ApplicationConfiguration component to null:

```
keyToApplicationName^=/Constants.null
```

Mapping of applications to sites is done through the `applicationGroupingMap` property of the `GroupingApplicationRoutingStrategy` component. This property is a Map where each key is the name of an EAC application and the corresponding value is a list of the site IDs of the sites to be routed to that application. The list is in the form of a string with the pipe character (`|`) used as the separator between site IDs. For example:

```
applicationGroupingMap=\
  footwearStores=shoeSiteUS|shoeSiteCanada,\
  apparelStores=clothesSiteUS|clothesSiteUK|clothesSiteCanada
```

Set the `routingObjectAdapter` property of the `/atg/endeca/index/IndexingApplicationConfiguration` component to specify the `ContextRoutingObjectAdapter` component to use:

```
routingObjectAdapter=\
  /atg/endeca/index/configuration/GroupingContextRoutingObjectAdapter
```

Set the `routingObjectAdapter` property of the `/atg/endeca/assembler/AssemblerApplicationConfiguration` component to specify the `RequestRoutingObjectAdapter` component to use:

```
routingObjectAdapter=\
  /atg/endeca/index/configuration/GroupingRequestRoutingObjectAdapter
```

To ensure that separate records are created for each EAC application, you need to add the `MultipleSiteVariantProducer` to the `variantProducers` property of each `EndecaIndexingOutputConfig` component. For example:

```
variantProducers+=/atg/search/repository/MultipleSiteVariantProducer
```

`EndecaIndexingOutputConfig` components are discussed in the next several chapters of this manual. For information about variant producers, including `MultipleSiteVariantProducer`, see the Using Variant Producers (page 63)section of the *Customizing the Output Records* (page 61) chapter.

3 Overview of Indexing

To make data in GSA repositories available for searching, the Oracle Commerce Platform must transform the data into the appropriate format, and then submit this data to Oracle Commerce for indexing.

The process of indexing GSA repository data in Guided Search works like this:

1. Oracle Commerce Platform components transform the repository data into Guided Search records that represent Guided Search properties, dimensions, and schema:

   - Properties of GSA repository items are used to create Guided Search properties and non-hierarchical dimensions.

   - The hierarchy of repository item types is used to create a hierarchical dimension in Guided Search. If you index the product catalog, the category hierarchy is used to create a hierarchical category dimension.

   - A Guided Search schema is created by examining the set of repository item properties to be indexed.

2. The generated data and dimension value records are submitted to CAS record stores. The schema records are converted to the format used by the Endeca Configuration Repository and this data is submitted to it using the Configuration Import API.

3. The EAC application is invoked, which processes the data and invokes indexing.

This chapter gives an overview of the classes and components that perform these steps, and the user interface provided for managing the process. It focuses on the product catalog repository, but the process described here applies to indexing any GSA repository.

This chapter includes the following sections:

Other chapters of this book provide more detail about configuring and using these and other classes and components to work with the data in your Oracle Commerce Platform environment.

Indexable Classes

The Oracle Commerce Platform includes an interface, `atg.endeca.index.Indexable`, that is implemented by the classes involved in creating Guided Search records. Key classes that implement this interface include:

- `atg.endeca.index.EndecaIndexingOutputConfig`
- `atg.commerce.endeca.index.dimension.CategoryTreeService`
- `atg.endeca.index.dimension.RepositoryTypeHierarchyExporter`
- `atg.endeca.index.schema.SchemaExporter`

These classes are discussed below.

EndecaIndexingOutputConfig Class

The main class used to specify how to transform repository items into records is `atg.endeca.index.EndecaIndexingOutputConfig`. The Guided Search integration includes two components of this class for transforming data in the product catalog:

- `/atg/commerce/search/ProductCatalogOutputConfig`
- `/atg/commerce/endeca/index/CategoryToDimensionOutputConfig`

Each `EndecaIndexingOutputConfig` component has a number of properties, as well as an XML definition file, for configuring how repository data should be transformed to create Guided Search records. The configuration of these components is discussed in detail in EndecaIndexingOutputConfig Components (page 28).

ProductCatalogOutputConfig Component

The `ProductCatalogOutputConfig` component specifies how to create Guided Search data records that represent items in the product catalog. Each record represents either one product or one SKU (depending on whether you use product-based or SKU-based indexing), and contains the values of the properties to be included in the index.

In addition, each record includes properties of parent and child items. For example, a record that represents a product includes information about its parent category's properties, as well as information about the properties of its child SKUs. This makes it possible to search category and SKU properties as well as product properties when searching for products in the catalog.

The names of the output properties include information about the item types they are associated with. For example, a record generated from a product may have a `product.description` property that holds the value of the `description` property of the `product` item, and a `sku.color` property that holds the value of the `color` properties of the product's child SKUs.

Multi-value properties are given names without array subscripts. For example, a `product` repository item might have multiple child `sku` items, each with a different value for the `color` property. In the output record there will be multiple entries for `sku.color`.

The following is an XML representation of a portion of a Commerce Reference Store data record. Note that the actual records submitted to the CAS data record store are in a binary object format, not XML.

```
<RECORD>
  <PROP NAME="product.baseUrl">
    <PVAL>atgrep:/ProductCatalog/clothing-sku/xsku1013</PVAL>
```

```
  </PROP>
  <PROP NAME="product.repositoryId">
    <PVAL>xprod1003</PVAL>
  </PROP>
  <PROP NAME="product.brand">
    <PVAL>CricketClub</PVAL>
  </PROP>
  <PROP NAME="product.language">
    <PVAL>English</PVAL>
  </PROP>
  <PROP NAME="product.priceListPair">
    <PVAL>plist3080003_plist3080002</PVAL>
  </PROP>
   <PROP NAME="product.description">
    <PVAL>Genuine English leather wallet</PVAL>
  </PROP>
  <PROP NAME="product.displayName">
    <PVAL>Organized Wallet</PVAL>
  </PROP>
  <PROP NAME="sku.activePrice">
    <PVAL>24.49</PVAL>
  </PROP>
  <PROP NAME="clothing-sku.color">
    <PVAL>Brown</PVAL>
  </PROP>
  <PROP NAME="clothing-sku.size">
    <PVAL>One Size</PVAL>
  <PROP NAME="record.id">
    <PVAL>
  clothing-sku-xsku1013..xprod1003.masterCatalog.en__US.plist3080003__plist3080002
    </PVAL>
  </PROP>
  <PROP NAME="record.source">
    <PVAL>
      ProductCatalog
    </PVAL>
  </PROP>
  <PROP NAME="record.type">
    <PVAL>
      clothing-sku
    </PVAL>
  </PROP>
</RECORD>
```

CategoryToDimensionOutputConfig Component

The `CategoryToDimensionOutputConfig` component specifies how to create Guided Search dimension value records that represent categories from the product catalog. This category dimension makes it possible to use Guided Search to navigate the categories of a catalog.

`CategoryToDimensionOutputConfig` creates dimension values using a special representation of the category hierarchy that is generated by the`/atg/commerce/endeca/index/CategoryTreeService` component, as described in the CategoryTreeService Class (page 20) section.

The following example shows an XML representation of a portion of a category dimension value record generated by `CategoryToDimensionOutputConfig`:

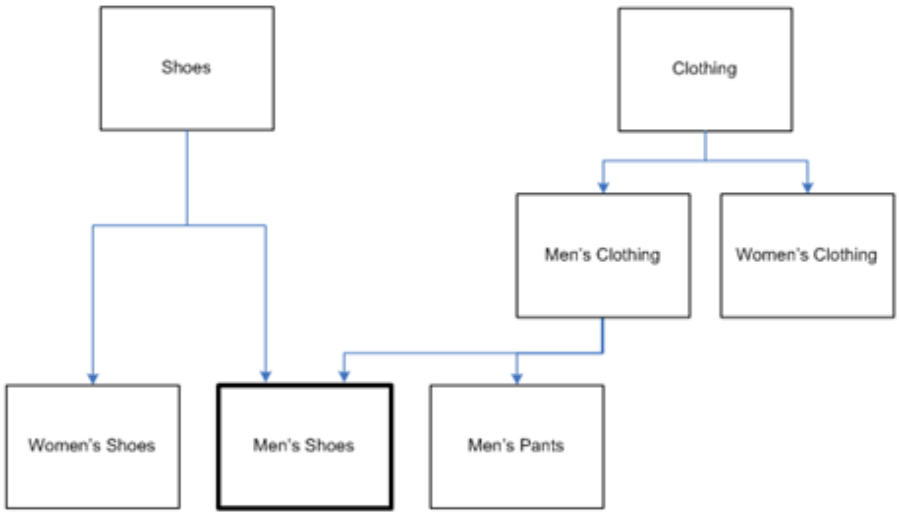
```
<RECORD>
```

```
  <PROP NAME="dimval.spec">
    <PVAL>cat10016.cat10014.catDeskLamps</PVAL>
  </PROP>
  <PROP NAME="Endeca.Id">
    <PVAL>product.category:cat10016.cat10014.catDeskLamps</PVAL>
  </PROP>
  <PROP NAME="category.rootCatalogId">
    <PVAL>masterCatalog</PVAL>
  </PROP>
  <PROP NAME="category.ancestorCatalogIds">
    <PVAL>masterCatalog</PVAL>
  </PROP>
  <PROP NAME="dimval.dimension_name">
    <PVAL>product.category</PVAL>
  </PROP>
  <PROP NAME="dimval.parent_spec">
    <PVAL>cat10016.cat10014</PVAL>
  </PROP>
  <PROP NAME="dimval.display_order">
    <PVAL>2</PVAL>
  </PROP>
  <PROP NAME="category.repositoryId">
    <PVAL>catDeskLamps</PVAL>
  </PROP>
  <PROP NAME="category.catalogs.repositoryId">
    <PVAL>masterCatalog,homeStoreCatalog</PVAL>
  </PROP>
  <PROP NAME="dimval.display_name">
    <PVAL>Desk Lamps</PVAL>
  </PROP>
</RECORD>
```

CategoryTreeService Class

The Guided Search integration uses the category hierarchy in the product catalog to construct a category dimension in Guided Search. In some cases, the hierarchy cannot be translated directly, because the Core Commerce catalog hierarchy supports categories with multiple parent categories, while Guided Search requires each dimension value to have a single parent.

For example, suppose you have the following category structure in your product catalog:



To deal with this structure, the Guided Search integration creates two different records for the Men's Shoes dimension value, one for each path to this category in the catalog hierarchy. These paths are computed by the `atg.commerce.endeca.index.dimension.CategoryTreeService` class.

The Guided Search integration includes a component of this class, `/atg/commerce/endeca/index/CategoryTreeService`. This component, which is run in the first phase of the indexing process, creates data structures in memory that represent all possible paths to each category in the product catalog. A category can have multiple parents, and those parents and their ancestors can each have multiple parents, so there can be any number of unique paths to an individual category.

The `CategoryToDimensionOutputConfig` component then uses the `/atg/commerce/endeca/index/CategoryPathVariantProducer` component to create multiple records for each category, one for each path computed by `CategoryTreeService`. For each path, the corresponding record uses the pathname as the value of its `dimval.spec` property; this makes it possible to differentiate records that represent different paths to the same category.

In the example above, two records are created for the Men's Shoes category. The `dimval.spec` entry in one of the records might be:

```
<PROP NAME="dimval.spec">
  <PVAL>catClothing.catMensClothing.catMensShoes</PVAL>
</PROP>
```

The `dimval.spec` entry in the other record for the category might be:

```
<PROP NAME="dimval.spec">
  <PVAL>catShoes.catMensShoes</PVAL>
</PROP>
```

Note that the period (`.`) is used as a separator in the property values rather the slash (`/`). This is done so the value can be passed to Guided Search through a URL query parameter when issuing a search query, without requiring any characters to be escaped.

RepositoryTypeHierarchyExporter Class

The `atg.endeca.index.dimension.RepositoryTypeHierarchyExporter` class creates Guided Search dimension value records from the hierarchy of repository item types, and submits those records to the CAS dimension values record store. This dimension is not typically displayed on a site, but can be used in determining which other dimensions to display. For example, Commerce Reference Store has a `furniture-sku` subtype that includes a `woodFinish` property that can be used as a Guided Search dimension. A site can include logic to detect whether the items returned from a search are of type `furniture-sku`, and display the `woodFinish` dimension if they are.

The Guided Search integration includes a component of class `RepositoryTypeHierarchyExporter`, `/atg/commerce/endeca/index/RepositoryTypeDimensionExporter`, that is configured to work with the `ProductCatalogOutputConfig` component. The `RepositoryTypeDimensionExporter` component outputs dimension value records for all of the repository item types referred to in the `ProductCatalogOutputConfig` definition file, as well as the ancestors and descendants of those item types. `RepositoryTypeDimensionExporter` does not create records for any item types that are not part of the hierarchy mentioned in the definition file.

There are additional components of class `RepositoryTypeHierarchyExporter` that create dimension value records representing the item types in the content management repository. See the *Indexing the Content Management Repository* (page 73) chapter for more information.

The following example shows a record produced by the `RepositoryTypeDimensionExporter` component for the `product` item type:

```
<RECORD>
  <PROP NAME="dimval.dimension_name">
    <PVAL>record.type</PVAL>
  </PROP>
  <PROP NAME="dimval.display_name">
    <PVAL>Product</PVAL>
  </PROP>
  <PROP NAME="Endeca.Id">
    <PVAL>record.type:product</PVAL>
  </PROP>
  <PROP NAME="dimval.spec">
    <PVAL>product</PVAL>
  </PROP>
  <PROP NAME="dimval.parent_spec">
    <PVAL>/</PVAL>
  </PROP>
</RECORD>
```

SchemaExporter Class

The `atg.endeca.index.schema.SchemaExporter` class is responsible for generating schema configuration and submitting it to the Endeca Configuration Repository. (See Submitting the Records (page 24) for information about this process.) The `/atg/commerce/endeca/index/SchemaExporter` component of this class examines the `ProductCatalogOutputConfig` definition file and generates a schema record for each specified property of a repository item type. The schema record indicates whether the property should be treated as a property or a dimension by Guided Search, whether it should be searchable, and the data type of the property or dimension.

Note, however, that these schema records are not in the format required by the CAS-based deployment template. Therefore, the `/atg/endeca/index/ConfigImportDocumentSubmitter` component converts the

schema data to Configuration Import API objects before submitting it to the Endeca Configuration Repository. See the Document Submitter Components (page 37) section for more information.

Indexing Multiple Languages

This section summarizes considerations that are specific to indexing data in multiple languages.

Data Records

If you are indexing data in multiple languages, separate data records must be generated for each language. For example, if a product has separate data for French and English (such as descriptions and color names), a French record and an English record must be generated. This is true regardless of whether you have a separate EAC application for each language or multiple languages in the same EAC application.

To generate the data records, the `ProductCatalogOutputConfig` component uses the `LocaleVariantProducer`, which ensures that separate records are created for each of the locales listed in the `/atg/endeca/ApplicationConfiguration` component's `locales` property. (See Using Variant Producers (page 63) for more information about `LocaleVariantProducer`.) If multiple languages are indexed by the same application, each data record includes a `product.language` property whose value identifies the language of the record. The language name is given in its own language. For example, the value for the German language is Deutsch.

Schema

If there is a separate application for each language, a separate schema is generated for each application. If there are multiple languages in an application, a single schema is generated based on the first locale listed in the `/atg/endeca/ApplicationConfiguration` component's `locales` property.

Dimension Values

If you are indexing data in multiple languages, separate dimension value records must be generated for each language. This is true regardless of whether you have a separate EAC application for each language or multiple languages in the same EAC application.

To generate category dimension value records, the `CategoryToDimensionOutputConfig` component uses the `LocaleVariantProducer` to create separate records for each of the locales listed in the `/atg/endeca/ApplicationConfiguration` component's `locales` property. The `RepositoryTypeDimensionExporter` component also generates separate records for each language.

If multiple languages are indexed in the same application, the records generated by the `/atg/commerce/endeca/index/RepositoryTypeDimensionExporter` component contain additional properties for the translated display names of the repository item types. These properties are named `displayName_`*`languageCode`*, where *`languageCode`* is the two-letter language code associated with one of the specified locales. For example:

```
<PROP NAME="displayName_en">
  <PVAL>Product</PVAL>
</PROP>
<PROP NAME="displayName_de">
  <PVAL>Produkt</PVAL>
</PROP>
<PROP NAME="displayName_es">
```

```
  <PVAL>Producto</PVAL>
</PROP>
```

Note that the property names shown in the example above are appropriate for use with CAS-based deployment templates, and assume that the name changes specified in `propertyNameReplacementMap` property of the `DimensionDocumentSubmitter` component have been applied. See RecordStoreDocumentSubmitter (page 37)for more information.

In addition, if you set the `multiLanguageSynonyms` property of the `RepositoryTypeDimensionExporter` component to `true`, then additional Guided Search record properties are generated to indicate that all translations of the same repository type are synonyms for searching. For example:

```
<PROP NAME="dimval.search_synonym">
  <PVAL>Product</PVAL>
  <PVAL>Produkt</PVAL>
  <PVAL>Producto</PVAL>
</PROP>
```

Submitting the Records

Once the records have been generated, they are submitted to Guided Search by components of classes that implement the `atg.repository.search.indexing.DocumentSubmitter` interface. The Guided Search integration includes these `DocumentSubmitter` components:

- `/atg/endeca/index/DataDocumentSubmitter` – This component of class `atg.endeca.index.RecordStoreDocumentSubmitter` submits records to the data record store (for example, `ATGen-data`).
- `/atg/endeca/index/DimensionDocumentSubmitter` -- This component of class `atg.endeca.index.RecordStoreDocumentSubmitter` submits records to the dimension values record store (for example, `ATGen-dimvals`).
- `/atg/endeca/index/ConfigImportDocumentSubmitter` -- This component of class `atg.endeca.index.ConfigImportDocumentSubmitter` converts the schema records to Configuration Import API objects and submits them to the Endeca Configuration Repository.

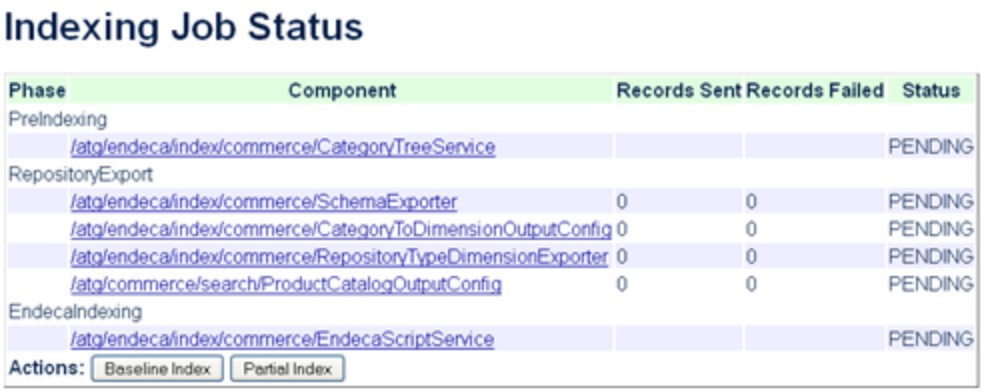
The `EndecaIndexingOutputConfig`, `RepositoryTypeHierarchyExporter`, and `SchemaExporter` classes each have a `documentSubmitter` property that is used to specify the document submitter component to use to submit records. The following table shows default values of the `documentSubmitter` property of each component of these classes:

Component	Record Submitter
`ProductCatalogOutputConfig`	`DataDocumentSubmitter`
`CategoryToDimensionOutputConfig`	`DimensionDocumentSubmitter`
`RepositoryTypeDimensionExporter`	`DimensionDocumentSubmitter`

Component	Record Submitter
SchemaExporter	ConfigImportDocumentSubmitter

Managing the Process

The `atg.endeca.index.admin.SimpleIndexingAdmin` class provides a mechanism for managing the process of generating records, submitting them to Guided Search, and invoking indexing. The Guided Search integration includes a component of this class, `/atg/commerce/endeca/index/ProductCatalogSimpleIndexingAdmin`, that is configured to manage indexing of the product catalog. The page for this component in the Component Browser of the Dynamo Server Admin presents a simple user interface for controlling and monitoring the process:



After the records are generated and submitted to Guided Search, `ProductCatalogSimpleIndexingAdmin` calls the `/atg/commerce/endeca/index/EndecaScriptService` component (of class `atg.endeca.eacclient.ScriptIndexable`). This component is responsible for invoking EAC scripts that trigger indexing.

The UI provides buttons for initiating a Guided Search baseline index or a partial update. Note that even if you click Partial Index, a baseline update may be triggered if the changes since the last baseline update necessitate it. See EndecaIndexingOutputConfig Components (page 28) for more information.

Viewing the Indexed Data

You can view the indexed data residing in your MDEX engines using Guided Search's JSP Reference Implementation. To use this reference implementation, do the following:

1. In a browser, navigate to `http://`*`host`*`:`*`port`*`/endeca_jspref`, where *`host`*`:`*`port`* refers to the name and port of the server hosting the Guided Search Tools and Frameworks installation, for example:

   ```
   http://localhost:8006/endeca_jspref
   ```

2. Click the ENDECA-JSP Reference Implementation link.

3. Enter an MDEX host and port, and then click Go.

4 Configuring the Indexing Components

This chapter provides detailed information about the Nucleus components in the Guided Search integration that are involved in indexing product catalog data, the functions these components perform, how they are configured, and how you can modify them to alter various aspects of indexing. It includes the following sections:

For information about the components for indexing data in the content management repository, such as articles and media content items, see the *Indexing the Content Management Repository* (page 73) chapter.

IndexingApplicationConfiguration Component

The `atg.endeca.index.configuration.IndexingApplicationConfiguration` class provides a central place for configuring various indexing settings. The Guided Search integration includes a component of this class, `/atg/endeca/index/IndexingApplicationConfiguration`. This component is configured by default with typical settings, but you can override these defaults when you use CIM to configure your Oracle Commerce Platform environment.

CASHostName

The hostname of the machine running CAS. The default setting is:

```
CASHostName=localhost
```

CASPort

The port number for accessing CAS. The default setting is:

```
CASPort=8500
```

eacHostName

The hostname of the EAC server. The default setting is:

```
eacHost=localhost
```

eacPort

The port number for accessing the EAC server. The default setting is:

```
eacPort=8888
```

routingObjectAdapter

A component of a class that implements the `atg.endeca.index.configuration.ContextRoutingObjectAdapter` interface. The specific class must reflect the routing strategy in use. See the *Routing* (page 9)chapter for more information.

applicationConfiguration

The component of class `atg.endeca.configuration.ApplicationConfiguration` used to configure global settings for the integration. The default setting is:

```
applicationConfiguration=/atg/endeca/ApplicationConfiguration
```

EndecaIndexingOutputConfig Components

The `atg.endeca.index.EndecaIndexingOutputConfig` class has a number of properties that configure various aspects of the record creation and submission process:

indexingApplicationConfiguration

The component of class `atg.endeca.index.configuration.IndexingApplicationConfiguration` used to configure indexing settings for the integration. For both the `ProductCatalogOutputConfig` and `CategoryToDimensionOutputConfig` components, the default setting is:

```
indexingApplicationConfiguration=\
  /atg/endeca/index/IndexingApplicationConfiguration
```

definitionFile

The full Nucleus pathname of the XML indexing definition file that specifies the repository item types and properties to include in the Guided Search records. For the `/atg/commerce/search/ProductCatalogOutputConfig` component, this property is set as follows:

```
definitionFile=/atg/commerce/endeca/index/product-sku-output-config.xml
```

For `/atg/commerce/endeca/index/CategoryToDimensionOutputConfig`:

```
definitionFile=/atg/commerce/endeca/index/category-dim-output-config.xml
```

See the *Configuring EndecaIndexingOutputConfig Definition Files* (page 49) chapter for information about the definition file's elements and attributes that configure how GSA repository items are transformed into Guided Search records.

repository

The full Nucleus pathname of the repository that the definition file applies to. For both the `ProductCatalogOutputConfig` and `CategoryToDimensionOutputConfig` components, this property is set to the product catalog repository:

```
repository=/atg/commerce/catalog/ProductCatalog
```

It is also possible to specify the repository in the indexing definition file using the `repository-path` attribute of the top-level `item` element. If the repository is specified in the definition file and also set by the component's `repository` property, the value set by the `repository` property overrides the value set in the definition file.

Note that in an ATG Content Administration environment, the repository should *not* be set to a versioned repository. Instead, it should be set to the corresponding unversioned target repository. For example, an `EndecaIndexingOutputConfig` component for a product catalog in an ATG Content Administration environment could be set to:

```
repository=/atg/commerce/catalog/ProductCatalog_production
```

repositoryItemGroup

A component of a class that implements the `atg.repository.RepositoryItemGroup` interface. This interface defines a logical grouping of repository items. Items that are not included in this logical grouping are excluded from the index. For the `CategoryToDimensionOutputConfig` component, this property is set by default to null (so no items are excluded). For the `ProductCatalogOutputConfig` component, `repositoryItemGroup` property is set by default to:

```
repositoryItemGroup=/atg/commerce/search/IndexedItemsGroup
```

The IndexedItemsGroup component uses this targeting rule set to select only products that have an ancestor catalog:

```
<ruleset>
  <accepts>
    <rule op=isNotNull>
      <valueof target="computedCatalogs">
    </rule>
  </accepts>
</ruleset>
```

This rule set ensures that the index does not include products that are not part of the catalog hierarchy.

It is also possible to specify a repository item group in the indexing definition file using the repository-item-group attribute of the top-level item element. If a repository item group is specified in the definition file and also by the component's repositoryItemGroup property, the value set by the repositoryItemGroup property overrides the value set in the definition file.

Note that the IndexedItemGroup component has a repository property that specifies the repository that the items are selected from. This value must match the repository that the ProductCatalogOutputConfig is associated with.

For more information about targeting rule sets, see the *Personalization Programming Guide*.

documentSubmitter

The component (typically of class atg.endeca.index.RecordStoreDocumentSubmitter) to use to submit records to the appropriate CAS record store. For the ProductCatalogOutputConfig component, this property is set as follows:

```
documentSubmitter=/atg/endeca/index/DataDocumentSubmitter
```

For the CategoryToDimensionOutputConfig component:

```
documentSubmitter=/atg/endeca/index/DimensionDocumentSubmitter
```

See Document Submitter Components (page 37) for more information.

forceToBaselineOnChange

If true, a baseline update is performed when a partial update is requested, if a value of a hierarchical dimension has been changed. For CatalogToDimensionOutputConfig, this property is set to true by default, because the component generates category dimension values, which are hierarchical. For ProductCatalogOutputConfig, this property is set to false by default, because the component does not generate hierarchical dimension values.

configRepositoryItemChangedProcessor

A component of a class that implements the atg.repository.search.indexing.ConfigRepositoryItemChangedProcessor interface. For CatalogToDimensionOutputConfig, this property is set to /atg/commerce/endeca/index/CategoryRepositoryItemChangedProcessor. This component is responsible for preventing category items from being added to the incremental item queue unnecessarily.

If the `forceToBaselineOnChange` property is `true`, a baseline update is triggered when a partial update is requested, if the incremental item queue contains any `category` items. In some cases, the baseline update is not really necessary, because the `category` item changes do not affect the category dimension values (for example, changes to properties that are not included in the indexed records). In this situation, `CategoryRepositoryItemChangedProcessor` prevents the changes from being added to the queue, so a baseline update is not triggered.

bulkLoader

A Nucleus component of class `atg.endeca.index.RecordStoreBulkLoaderImpl`. This is typically set to `/atg/search/repository/BulkLoader`. Any number of `EndecaIndexingOutputConfig` components can use the same bulk loader.

See Data Loader Components (page 33) for more information.

enableIncrementalLoading

If `true`, incremental loading is enabled.

incrementalLoader

A Nucleus component of class `atg.endeca.index.RecordStoreIncrementalLoaderImpl`. This is typically set to `/atg/search/repository/IncrementalLoader`. Any number of `EndecaIndexingOutputConfig` components can use the same incremental loader.

See Data Loader Components (page 33) for more information.

excludedItemsAncestorIds

A list of the IDs of the items whose child items should not be indexed. For example, Commerce Reference Store excludes products and SKUs that are not part of the standard catalog hierarchy (such as gift wrapping) by setting the `excludedItemsAncestorIds` property of the `ProductCatalogOutputConfig` component to:

```
excludedItemsAncestorIds=\
  NonNavigableProducts,homeStoreNonNavigableProducts
```

Note that an item is excluded only if all of its ancestor items are specified. So to exclude a product that is several levels deep in the catalog hierarchy, `excludedItemsAncestorIds` must list all of the categories in the path to the product you want to exclude. If there are multiple paths to an item, all of its ancestor categories in all of those paths must be listed in `excludedItemsAncestorIds`, or the item will not be excluded. For example, if all of a product's ancestors in one path are excluded, but there is another path to this product and some of the categories in that path are not excluded, the product will be indexed.

siteIDsToIndex

A list of site IDs of the sites to include in the index. The value of this property is used to automatically set the value of the `sitesToIndex` property, which is the actual property used to determine which sites to index. If `siteIDsToIndex` is explicitly set to a list of site IDs, `sitesToIndex` is set to the sites that have those IDs. If the value of `siteIDsToIndex` is null (the default), `sitesToIndex` is set to a list of all enabled sites. So it is only necessary to set `siteIDsToIndex` if you want to restrict indexing to only a subset of the enabled sites.

replaceWithTypePrefixes

A list of the property-name prefixes that should be replaced with the item type that the property is associated with. In this list, a period (`.`) specifies that a type prefix should be added to properties

of the top-level item, which is `product` for `ProductCatalogOutputConfig` and `category` for `CategoryToDimensionOutputConfig`.

For `ProductCatalogOutputConfig`, the `replaceWithTypePrefixes` property is set by default to:

```
replaceWithTypePrefixes=.,childSKUs
```

This means, for example, that the `brand` property of the `product` item is given the name `product.brand` in the output records, and the `onSale` property of the `sku` item (which appears in the definition file as the `childSKUs` property of the `product` item) is given the name `sku.onSale`. Properties that are specific to a `sku` subtype are prefixed with the subtype name in the output records. For example, Commerce Reference Store has a `furniture-sku` subtype, so the `woodFinish` property (which is specific to this subtype) is given the output name `furniture-sku.woodFinish`, while `onSale` (which is common to all SKUs) is given the name `sku.onSale`.

Adding these prefixes ensures that there is no duplication of property or dimension names in Guided Search, in case different repository item types (or records from other sources) have identically named properties.

For `CategoryToDimensionOutputConfig`, the `replaceWithTypePrefixes` property is set to:

```
replaceWithTypePrefixes=.
```

This means, for example, that the `ancestorCatalogIds` property of the `category` item is given the name `category.ancestorCatalogIds` in the output records.

If `replaceWithTypePrefixes` is null, the behavior is the same as if the property is set to a period; the type prefix is added to the names of the output properties of the top-level item. Note, however, that in this case the behavior is due to a default in the Java class, rather than the Nucleus configuration. So if you add a setting like the following in a properties file, the class default will no longer be in effect, which means the type prefix will not be added to properties of the top-level item:

```
replaceWithTypePrefixes+=childSKUs
```

To get the desired results, you should instead use a setting like this:

```
replaceWithTypePrefixes=.,childSKUs
```

prefixReplacementMap

A mapping of property-name prefixes to their replacements. This mapping is applied after any type prefixes are added by `replaceWithTypePrefixes`.

For `ProductCatalogOutputConfig`, `prefixReplacementMap` is set by default to:

```
prefixReplacementMap=\
  product.ancestorCategories=allAncestors
```

So, for example, the `ancestorCategories.displayName` property is renamed to `product.ancestorCategories.displayName` by applying `replaceWithTypePrefixes`, and then the result is renamed to `allAncestors.displayName` by applying `prefixReplacementMap`.

For `CategoryToDimensionOutputConfig`, `prefixReplacementMap` is set to null by default, so no prefix replacement is performed.

suffixReplacementMap

A mapping of property-name suffixes to their replacements. In addition to any mappings you specify in the properties file, the following mappings are automatically included:

```
$repositoryId=repositoryId,
$siteId=siteId,
$url=url,
$baseUrl=baseUrl
```

These mappings remove the dollar-sign ($) character from the names of special repository properties, because this character is not valid in Guided Search property names.

The `suffixReplacementMap` property is set to null by default for both `ProductCatalogOutputConfig` and `CategoryToDimensionOutputConfig`, which means only the automatic mappings are used. You can exclude the automatic mappings by setting the `addDefaultOutputNameReplacements` property to `false`.

Data Loader Components

The `EndecaIndexingOutputConfig` components specify how to generate records from items in the catalog repository, but the generation itself is performed by data loader components. Depending on your environment, data loading may be an operation that is performed occasionally (if the content rarely changes) or frequently (if the content changes often). To be as flexible as possible, the Guided Search integration provides two approaches to loading the data:

- **Bulk loading** generates the complete set of records for the catalog. Bulk loading is performed by the `atg.endeca.index.RecordStoreBulkLoaderImpl` class. The Guided Search integration includes a component of this class, `/atg/search/repository/BulkLoader`.
- **Incremental loading** generates only the records that have changed since the last load. The incremental loader records which repository items have changed since the last incremental or bulk load. It deletes the records that represent items that have been deleted, and creates records for any items that are new or have been modified.

  Incremental loading is performed by the `atg.endeca.index.RecordStoreIncrementalLoaderImpl` class. The Guided Search integration includes a component of this class, `/atg/search/repository/IncrementalLoader`.

Bulk loading and incremental loading are not mutually exclusive. For some environments, only bulk loading will be necessary, especially if content is updated only occasionally. For other environments, incremental loading will be needed to keep the search content up to date, but even in that case, you should perform a bulk load occasionally to ensure the integrity of the indexed data.

Note that Guided Search always does a baseline update after the Oracle Commerce Platform performs bulk loading, and typically does a partial update after incremental loading. In some cases, however, a baseline update may be triggered after incremental loading. For example, if incremental loading adds a new category dimension value, a baseline update must be performed. See EndecaIndexingOutputConfig Components (page 28) for information about how to configure this.

The IncrementalLoader component uses an implementation of the PropertiesChangedListener interface to monitor the repository for add, update, and delete events. It then analyzes these events to determine which ones necessitate updating records, and creates a queue of the affected repository items. When a new incremental update is triggered, the IncrementalLoader processes the items in the queue, generating and loading a new record for each changed repository item.

Tuning Incremental Loading

The number of changed items accumulating in the queue can vary greatly, depending on how frequently your data changes and how long you specify between incremental updates. Rather than processing all of the changes at once, the EndecaIndexingOutputConfig component groups changes in batches called generations.

The EndecaIndexingOutputConfig class has a maxIncrementalUpdatesPerGeneration property that specifies the maximum number of changes that can be assigned to a generation. By default, this value is 1000, but you can change this value if necessary. Larger generations require more Oracle Commerce Platform resources to process, but reduce the number of Guided Search jobs required (and hence the overhead associated with starting up and completing these jobs). Smaller generations require fewer Oracle Commerce Platform resources, but increase the number of Guided Search jobs.

CategoryTreeService

The following describes key properties of the atg.commerce.endeca.index.dimension.CategoryTreeService class and the default configuration of the /atg/commerce/endeca/index/CategoryTreeService component of this class:

indexingOutputConfig

The component of class atg.endeca.index.EndecaIndexingOutputConfig whose definition file should be used for generating schema records. By default, this property is set to:

```
indexingOutputConfig=/atg/commerce/search/ProductCatalogOutputConfig
```

catalogTools

The component of class atg.commerce.catalog.custom.CustomCatalogTools for accessing the catalog repository. By default, this property is set to:

```
catalogTools=/atg/commerce/catalog/CatalogTools
```

excludeRootCategories

A boolean that specifies whether dimension values should be created for root categories. This property defaults to false, meaning dimension values are created. Commerce Reference Store sets excludeRootCategories to true, because its root categories are not displayed and therefore should not have associated dimension values.

excludedItemsCategoryIds

A list of IDs of categories that dimension values should not be created for, because their child products and SKUs are excluded from indexing. If `excludedItemsCategoryIds` is not set explicitly, it is automatically set to the list of category IDs in the `excludedItemsAncestorIds` property of the `ProductCatalogOutputConfig` component.

Note that to prevent creation of a dimension value for a specific category, all of its ancestor and descendant categories must be specified in `excludedItemsCategoryIds`. If there are multiple paths to a category, a separate dimension value is created for the category for each path; if you exclude the dimension value associated with one path (by listing all of the categories in that path), that does not prevent the creation of dimension values for other paths.

excludedCategoryIds

A list of IDs of categories that dimension values should not be created for (in addition to any categories excluded by the values of `excludeRootCategories` and `excludedItemsCategoryIds`).

sitesForCatalogs

To create a representation of the category hierarchy in which each category dimension value has only one parent, the `CategoryTreeService` class creates data structures in memory that represent all possible paths to each category in the product catalog. In order to do this, it must be provided with a list of the catalogs to use for computing paths.

The `sitesForCatalogs`property specifies a list of sites, and `CategoryTreeService` uses the catalogs associated with these sites for computing paths. The `sitesForCatalogs` property cannot be set through a properties file; by default, it is set automatically to the value of the `sitesToIndex` property of the associated `EndecaIndexingOutputConfig` component. If `sitesToIndex` is null, `CategoryTreeService` instead uses the `rootCatalogsRQLString` property to determine the catalogs.

rootCatalogsRQLString

An RQL query that returns a list of catalogs. If `sitesForCatalogs` is null, the catalogs returned from this query are used. The query is set by default to:

```
rootCatalogsRQLString=\
  directParentCatalogs IS NULL AND parentCategories IS NULL
```

If `sitesForCatalogs` and `rootCatalogsRQLString` are both null, `CategoryTreeService` uses the `rootCatalogIds` property to determine the catalogs.

rootCatalogIds

An explicit list of catalog IDs of the catalogs to use. This list is used if `sitesForCatalogs` and `rootCatalogsRQLString` are both null. By default, `rootCatalogIds` is set to null.

RepositoryTypeDimensionExporter

This section describes key properties of the `atg.endeca.index.dimension.RepositoryTypeHierarchyExporter` class and the default configuration of the `/atg/commerce/endeca/index/RepositoryTypeDimensionExporter` component of this class.

dimensionName

The name to give the dimension created from the hierarchy of repository item types. This property is set by linking to the `recordTypeName` property of the `/atg/endeca/ApplicationConfiguration` component:

```
dimensionName^=/atg/endeca/ApplicationConfiguration.recordTypeName
```

If you want to change the value of the `dimensionName` property, you should do so by changing the value of `ApplicationConfiguration.recordTypeName` to ensure that other properties that link to it are changed as well.

indexingOutputConfig

The component of class `atg.endeca.index.EndecaIndexingOutputConfig` whose definition file should be used for generating dimension value records from the repository item-type hierarchy. Set by default to:

```
indexingOutputConfig=/atg/commerce/search/ProductCatalogOutputConfig
```

documentSubmitter

The component (typically of class `atg.endeca.index.RecordStoreDocumentSubmitter`) to use to submit records to the CAS dimension values record store. (See Document Submitter Components (page 37) for more information.) Set by default to:

```
documentSubmitter=/atg/endeca/index/DimensionDocumentSubmitter
```

SchemaExporter

The following describes key properties of the `atg.endeca.index.schema.SchemaExporter` class and the default configuration of the `/atg/commerce/endeca/index/SchemaExporter` component of this class:

indexingOutputConfig

The component of class `atg.endeca.index.EndecaIndexingOutputConfig` whose definition file should be used for generating schema records. Set by default to:

```
indexingOutputConfig=/atg/commerce/search/ProductCatalogOutputConfig
```

documentSubmitter

The component (typically of class `atg.endeca.index.ConfigImportDocumentSubmitter`) to use to submit schema data to the Endeca Configuration Repository. (See Document Submitter Components (page 37) for more information.) Set by default to:

```
documentSubmitter=/atg/endeca/index/ConfigImportDocumentSubmitter
```

dimensionNameProviders

An array of components of classes that implement the `atg.endeca.index.schema.DimensionNameProvider` interface. `SchemaExporter` uses these components to create references from attribute names to dimension names.

By default, `dimensionNameProviders` is set to:

```
dimensionNameProviders+=RepositoryTypeDimensionExporter
```

Document Submitter Components

As described above, each component that generates records has a `documentSubmitter` property that is set by default to a component of a class that implements the `atg.repository.search.indexing.DocumentSubmitter` interface. These components perform two main functions:

- Converting the output from the formats used by the older Forge-based deployment template to the formats used by the CAS-based deployment template.
- Submitting the data to Guided Search for indexing.

The Guided Search integration includes several `DocumentSubmitter` components:

- The `/atg/endeca/index/DataDocumentSubmitter` and `/atg/endeca/index/DimensionDocumentSubmitter` components are of class `atg.endeca.index.RecordStoreDocumentSubmitter`. This class submits records to CAS using the Record Store API. The `DimensionDocumentSubmitter` component is configured to rename the dimension value properties in the submitted records to reflect the naming conventions used with CAS-based deployment templates
- The `/atg/endeca/index/ConfigImportDocumentSubmitter` component is of class `atg.endeca.index.ConfigImportDocumentSubmitter`. This component converts schema records to the format used by Endeca Configuration Repository and submits the schema configuration to it using the Configuration Import API.

In addition, the Guided Search integration includes the `atg.repository.search.indexing.submitter.FileDocumentSubmitter` class, which you can use to submit records to files for debugging purposes.

This section discusses the various document submitter classes and components.

RecordStoreDocumentSubmitter

The following are key properties of the `RecordStoreDocumentSubmitter` components.

indexingApplicationConfiguration

The component of class `atg.endeca.index.configuration.IndexingApplicationConfiguration` used to configure indexing settings for the integration. The default setting is:

```
indexingApplicationConfiguration=\
  /atg/endeca/index/IndexingApplicationConfiguration
```

endecaDataStoreType

The type of the record store to submit to. This property is set to `data` for the `DataDocumentSubmitter` component, and `dimval` for the `DimensionDocumentSubmitter` component.

idPropertyName

The record property whose value is used as the unique identifier for the record. For the `DimensionDocumentSubmitter` component, this property is set to `Endeca.id`. For the `DataDocumentSubmitter` component, this property is set to `record.id` by linking to the `recordIdName` property of the `/atg/endeca/ApplicationConfiguration` component:

```
idPropertyName^=/atg/endeca/ApplicationConfiguration.recordIdName
```

If you want to change the value of `DataDocumentSubmitter.idPropertyName`, you should do so by changing the value of `ApplicationConfiguration.recordIdName` to ensure that other properties that link to it are changed as well.

propertyNameReplacementMap

By default, the `/atg/commerce/endeca/index/CategoryToDimensionOutputConfig` and `/atg/commerce/endeca/index/RepositoryTypeDimensionExporter` components output dimension value records whose property names reflect the older Forge-based deployment template rather than the CAS-based template currently recommended. To support the naming conventions used with CAS-based deployment templates, the `propertyNameReplacementMap` property of the `DimensionDocumentSubmitter` component is used to map the older-style names to the new ones. By default, this property is set as follows:

```
propertyNameReplacementMap=\
  dimval.qualified_spec=Endeca.Id,\
  dimval.dimension_spec=dimval.dimension_name,\
  dimval.prop.category.ancestorCatalogIds=category.ancestorCatalogIds,\
  dimval.prop.category.rootCatalogId=category.rootCatalogId,\
  dimval.prop.displayName_es=displayName_es,\
  dimval.prop.displayName_en=displayName_en,\
  dimval.prop.displayName_de=displayName_de,\
  dimval.prop.category.repositoryId=category.repositoryId,\
  dimval.prop.category.catalogs.repositoryId=category.catalogs.repositoryId,\
  dimval.prop.category.siteId=category.siteId
```

So, for example, when the `CategoryToDimensionOutputConfig` component outputs a `dimval.dimension_spec` property in the records it generates, `DimensionDocumentSubmitter` converts the property name to `dimval.dimension_name` before submitting the records.

The `propertyNameReplacementMap` property of the `DataDocumentSubmitter` component is null by default, because the new naming conventions affect only the properties of dimension value records, not data records.

flushAfterEveryRecord

A boolean that specifies whether to flush the buffer used by the connection to CAS after each record is processed. This property is set by default to `false`. Setting it to `true` during debugging can be helpful for determining which records are being rejected by CAS, because the errors will be isolated to specific records.

enabled

A boolean that specifies whether this component is enabled. This property is set by default to `true`, but it can be set to `false` to always report success without submitting records to CAS. (This is useful for debugging purposes when a CAS instance is not available.)

Reducing Logging Messages

In order to write records to the CAS record stores, the `atg.endeca.index.RecordStoreDocumentSubmitter` class imports classes from the Guided Search `com.endeca.itl.record` and `com.endeca.itl.recordstore` packages. These classes make use of the Apache CXF framework.

Using the default CXF configuration results in a large number of informational logging messages. The volume of the messages can result in problems, such as locking up of the terminal window. Therefore, it is a good idea to reduce the number of logging messages by setting the logging level of the `org.apache.cxf.interceptor.LoggingInInterceptor` and `org.apache.cxf.interceptor.LoggingOutInterceptor` loggers to `WARNING`.

The way to set these logging levels differs depending on your application server. See the documentation for your application for information.

ConfigImportDocumentSubmitter

The following are key properties of the `ConfigImportDocumentSubmitter` component. Note that in order to submit schema configuration to the Endeca Configuration Repository, `ConfigImportDocumentSubmitter` must access the credential store for the Oracle Commerce Workbench to obtain login information. This credential store is specified through properties of the `/atg/endeca/ApplicationConfiguration` component. See Configuring the ApplicationConfiguration Component (page 4) for more information.

indexingApplicationConfiguration

The component of class `atg.endeca.index.configuration.IndexingApplicationConfiguration` used to configure indexing settings for the integration. The default setting is:

```
indexingApplicationConfiguration=\
  /atg/endeca/index/IndexingApplicationConfiguration
```

workbenchImportOwner

The import owner associated with the submitted schema configuration. (The import owner identifies the source of the configuration.) This property is set by default to:

```
workbenchImportOwner=ATG
```

For more information about the import owner, see the *Oracle Commerce Guided Search Administrator's Guide*.

FileDocumentSubmitter

To help optimize and debug your output, you can have the generated records sent to files rather than to the Guided Search record stores. Doing this enables you to examine the output without triggering indexing, so you can determine if you need to make changes to the configuration of the record-generating components.

To direct output to files, create a component of class `atg.repository.search.indexing.submitter.FileDocumentSubmitter`, and set the `documentSubmitter` property of the record-generating components to point to the `FileDocumentSubmitter` component. A separate file is created for each record generated.

Note that the output records reflect the naming conventions and data formats used with Forge-based deployment templates, because the renaming and conversions done by the other document submitters do not occur. Therefore, if you are using a CAS-based deployment template, the output from `FileDocumentSubmitter` may not match the records actually submitted to Guided Search by `DimensionDocumentSubmitter` and `ConfigImportDocumentSubmitter`.

The location and names of the files are automatically determined based on the following properties of `FileDocumentSubmitter`:

baseDirectory

The pathname of the directory to write the files to.

filePrefix

The string to prepend to the name of each generated file. Default is the empty string.

fileSuffix

The string to append to the name of each generated file. Set this as follows:

```
fileSuffix=.xml
```

nameByRepositoryId

If `true`, each filename will be based on the repository ID of the item the file represents. If `false` (the default), files are named `0.xml`, `1.xml`, etc.

overwriteExistingFiles

If `true`, if the generated filename matches an existing file, the existing file will be overwritten by the new file. If `false` (the default), the new file will be given a different name to avoid overwriting the existing file.

EndecaScriptService

The `/atg/commerce/endeca/index/EndecaScriptService` component (of class `atg.endeca.eacclient.ScriptIndexable`) is responsible for invoking EAC scripts that trigger indexing.

The following are key properties of this component.

EACScriptTimeout

The maximum amount of time (in milliseconds) to wait for an EAC script to complete execution before throwing an exception. Set by default to 1800000 (1 hour). For large indexing jobs, you may need to increase this value to ensure `EndecaScriptService` does not time out before indexing completes.

enabled

A boolean that specifies whether this component is enabled. This property is set by default to `true`, but it can be set to `false` to always report success without invoking a script. (This is useful for debugging purposes when an EAC instance is not available.)

indexingApplicationConfiguration

The component of class `atg.endeca.index.configuration.IndexingApplicationConfiguration` used to configure indexing settings for the integration. The default setting is:

```
indexingApplicationConfiguration=\
  /atg/endeca/index/IndexingApplicationConfiguration
```

ProductCatalogSimpleIndexingAdmin

The `/atg/commerce/endeca/index/ProductCatalogSimpleIndexingAdmin` component (of class `atg.endeca.index.admin.SimpleIndexingAdmin`) manages the process of generating records, submitting them to Guided Search, and invoking indexing. The page for this component in the Component Browser of the Dynamo Server Admin presents a simple user interface for controlling and monitoring the process.

The `SimpleIndexingAdmin` class defines indexing in terms of an indexing job, which is made of up indexing phases, which in turn contain indexing tasks. Each indexing task is responsible for executing an individual `Indexable` component. Tasks within a phase may run in sequence or in parallel, but in either case all tasks in a phase must complete before the next phase can begin.

By default, the `ProductCatalogSimpleIndexingAdmin` defines three phases:

1. PreIndexing -- Runs `/atg/commerce/endeca/index/CategoryTreeService`.
2. RepositoryExport -- Runs these components in parallel:
   - `/atg/commerce/endeca/index/SchemaExporter`
   - `/atg/commerce/endeca/index/CategoryToDimensionOutputConfig`
   - `/atg/commerce/endeca/index/RepositoryTypeDimensionExporter`
   - `/atg/commerce/search/ProductCatalogOutputConfig`
3. EndecaIndexing -- Runs `/atg/commerce/endeca/index/EndecaScriptService`, which invokes Guided Search indexing scripts.

`ProductCatalogSimpleIndexingAdmin` reports information about an indexing job, such as the start and finish time of the job, the duration of each phase, the status of each task, and the number of records submitted.

You can invoke indexing jobs manually through the `ProductCatalogSimpleIndexingAdmin` user interface. In addition, the `SimpleIndexingAdmin` class implements the `atg.service.scheduler.Schedulable` interface, so it is also possible to configure the `ProductCatalogSimpleIndexingAdmin` component to invoke indexing jobs automatically on a specified schedule. (See the *Platform Programming Guide* for information about the `Schedulable` interface and other Scheduler services.)

Key configuration properties of `ProductCatalogSimpleIndexingAdmin` include:

phaseToPrioritiesAndTasks

This property defines the phases and tasks of an indexing job, and the order in which the phases are executed. It is a comma-separated list of phases, where the format of each phase definition is:

```
phaseName=priority:Indexable1;Indexable2;...;IndexableN
```

Phases are executed in priority order, with lower number priorities executed first.

By default, this is set to:

```
phaseToPrioritiesAndTasks=\
  PreIndexing=5:CategoryTreeService,\
  RepositoryExport=10:\
    SchemaExporter;\
    CategoryToDimensionOutputConfig;\
    RepositoryTypeDimensionExporter;\
    /atg/commerce/search/ProductCatalogOutputConfig,\
    EndecaIndexing=15:EndecaScriptService
```

runTasksWithinPhaseInParallel

A boolean that controls whether to run tasks within a phase in parallel. Set to `true` by default. If set to `false`, the tasks are executed in sequence, in the order specified in the `phaseToPrioritiesAndTasks` property. Setting `runTasksWithinPhaseInParallel` to `false` can simplify debugging, because when tasks are run in parallel, logging messages from multiple components may be interspersed, making them difficult to read.

enableScheduledIndexing

A boolean that controls whether to invoke indexing automatically on a specified schedule. Set to `false` by default.

baselineSchedule

A String that specifies the schedule for performing baseline updates. Set to null by default. If you set `enableScheduledIndexing` to `true`, set `baselineSchedule` to a String that conforms to one of the formats accepted by classes implementing the `atg.service.scheduler.Schedule` interface, such as `atg.service.scheduler.CalendarSchedule` or `atg.service.scheduler.PeriodicSchedule`. For example, to schedule a baseline update to run every Sunday at 11:30 pm:

```
baselineSchedule=calendar * * 7 * 23 30
```

partialSchedule

A String that specifies the schedule for performing partial updates. The format for the String is the same as the format used for `baselineSchedule`. Set to null by default.

retryInMs

The amount of time (in milliseconds) to wait before retrying a scheduled indexing job if the first attempt to execute it fails. Set by default to -1, which means no retry. If you change this value, you should set it to a

relatively short amount of time to ensure that the indexing job completes before the next scheduled job begins. If `ProductCatalogSimpleIndexingAdmin` estimates that the retried job will not complete before the next scheduled job, it skips the retry.

jobQueue

Specifies the component that manages queueing of index jobs. Set by default to `/atg/endeca/index/InMemoryJobQueue`. See Queueing Indexing Jobs (page 43) for more information.

indexingMessageSource

A component of class `atg.endeca.index.events.IndexingMessageSource` that sends a JMS message when an indexing job completes. By default, this property is null, but you can set it to the `/atg/endeca/index/events/IndexingMessageSource` component that is included with the Oracle Commerce Platform. This message source is preconfigured in Patch Bay.

Note, however, that there is no message sink preconfigured to listen for events sent by `IndexingMessageSource`. The Oracle Commerce Platform does provide an abstract class, `atg.endeca.index.events.IndexingMessageSink`, that you can extend to listen for indexing events. You will also need to create a component from the class you create and configure the message sink in Patch Bay.

For more information about JMS and Patch Bay, see the *Oracle Commerce Platform Message System* chapter in the *Platform Programming Guide*.

Queueing Indexing Jobs

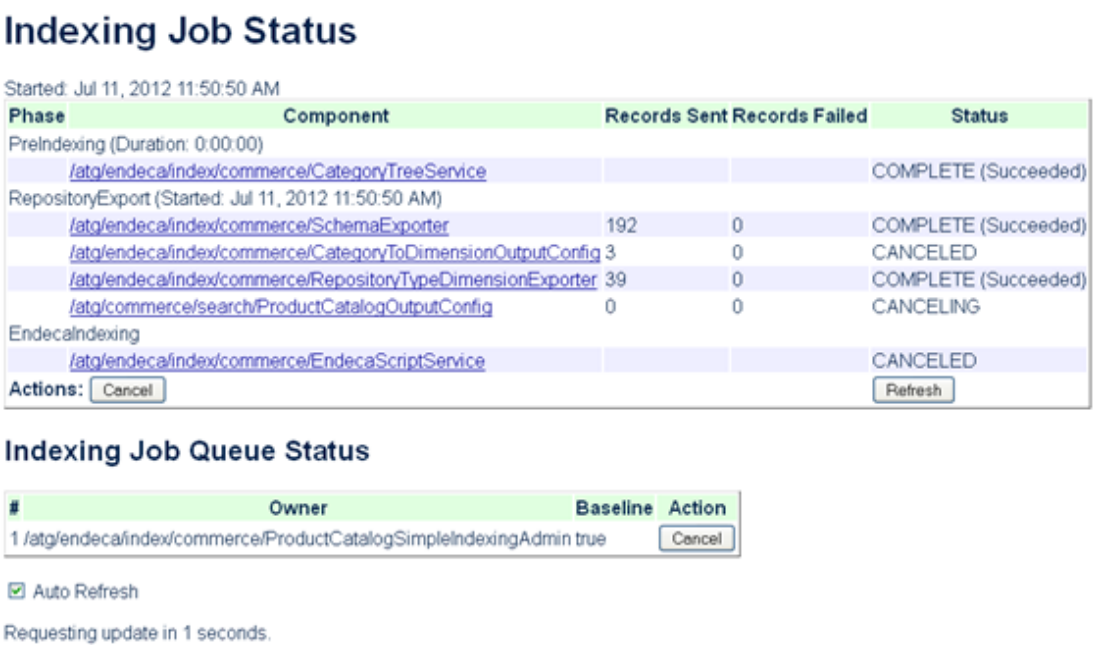
In certain cases, an indexing job cannot be executed immediately when it is invoked:

- If there is currently another indexing job running
- If an ATG Content Administration deployment is in progress

To handle these cases, `ProductCatalogSimpleIndexingAdmin` invokes the `/atg/endeca/index/InMemoryJobQueue` component. This component, which is of class `atg.endeca.index.admin.InMemoryJobQueue`, implements a memory-based indexing job queue that manages these jobs on a first-in, first-out basis.

In addition, the queue handles the case where an indexing job is in progress when an ATG Content Administration deployment is started. In this situation, the job in progress is stopped, moved to the top of the queue (ahead of any other pending jobs), and restarted when the deployment is complete.

Queued jobs are listed on the `ProductCatalogSimpleIndexingAdmin` page in the Component Browser of the Dynamo Server Admin. In the following example, an indexing job has been stopped due to an ATG Content Administration deployment, and moved to the queue to be restarted once the deployment completes:



ATG Content Administration Components

If your environment includes ATG Content Administration, be sure to include the `DCS.Endeca.Index.Versioned` module when you assemble the EAR file for your ATG Content Administration server. This module enables indexing jobs to be triggered automatically after a deployment, ensuring that changes deployed from ATG Content Administration are reflected in the index as quickly as possible. A full deployment triggers a baseline update, and an incremental deployment triggers a partial update.

Indexing can be configured to trigger either locally (on the ATG Content Administration server itself) or remotely (on the staging or production server). Note that even when indexing is executed on the ATG Content Administration server, the catalog repository that is indexed is the unversioned deployment target (`/atg/commerce/catalog/ProductCatalog_production`), not the versioned repository.

The Guided Search integration includes the `/atg/search/repository/IndexingDeploymentListener` component, which is of class `atg.epub.search.indexing.IndexingDeploymentListener`. This component listens for deployment events and, depending on the repositories involved, triggers one or more indexing jobs.

The `IndexingDeploymentListener` component has a `remoteSynchronizationInvokerService` property that is set by default to `/atg/search/SynchronizationInvoker`. The `SynchronizationInvoker` component, which is of class `atg.search.core.RemoteSynchronizationInvokerService`, controls whether indexing is invoked on the local (ATG Content Administration) server or on a remote system (such as the production server).

Specifying the Deployment Target

After you set your ATG Content Administration deployment topology and perform site initialization, configure the components of class `atg.endeca.index.EndecaIndexingOutputConfig` on the ATG Content

Administration server with the name of the deployment target. You typically set the `targetName` property of each `EndecaIndexingOutputConfig` component to `Production`:

```
targetName=Production
```

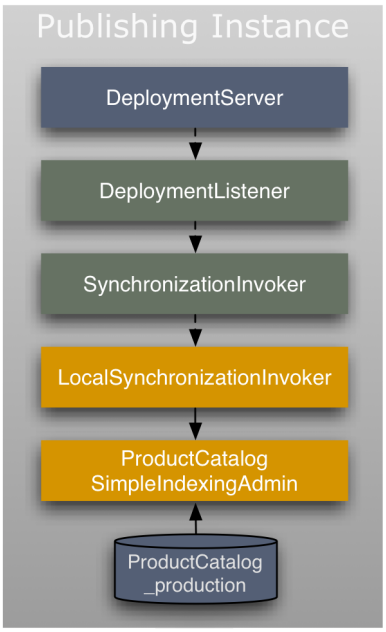
After setting the `targetName` properties, restart the Content Administration server so these settings take effect. When you restart, the `unversionedRepositoryPath` and `versionedRepositoryPath` properties of each `EndecaIndexingOutputConfig` component are automatically set, based on the deployment topology. These settings are needed in order for incremental loading to work properly.

Enabling Local Indexing

For local indexing (the default configuration), the `SynchronizationInvoker` component invokes the `/atg/endeca/index/LocalSynchronizationInvoker` component on the ATG Content Administration server to trigger the indexing job. This component, which is of class `atg.endeca.index.LocalSynchronizationInvoker`, is specified through the `localSynchronizationInvoker` property of the `SynchronizationInvoker` component:

```
localSynchronizationInvoker=/atg/endeca/index/LocalSynchronizationInvoker
```

The following diagram illustrates the configuration for local indexing:

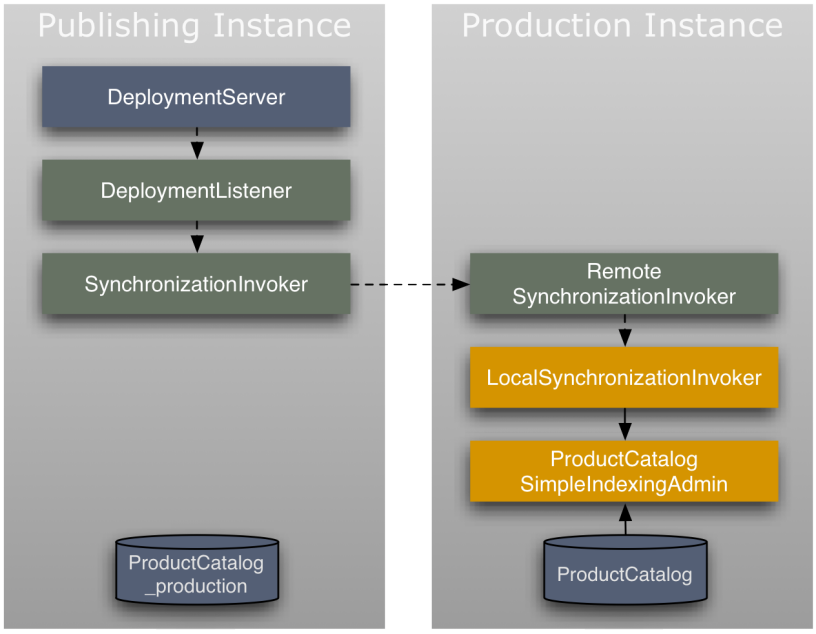


Enabling Remote Indexing

To enable remote indexing, modify the configuration of the `SynchronizationInvoker` component on the ATG Content Administration system so that it points to a `SynchronizationInvoker` component on the remote system, and configure the remote `SynchronizationInvoker` to point to a `LocalSynchronizationInvoker` on the remote system:

- On the ATG Content Administration system, set the `SynchronizationInvoker.host` property to the host name of the remote system, and set the `SynchronizationInvoker.port` property to the RMI port number to use for communication between systems. It is also a good idea to set the `SynchronizationInvoker.localSynchronizationInvoker` property on the ATG Content Administration system to null, to ensure local indexing is not triggered.
- On the remote system, ensure that the `SynchronizationInvoker.localSynchronizationInvoker` property is set to `/atg/endeca/index/LocalSynchronizationInvoker`.

The following diagram illustrates the configuration for remote indexing:



Triggering Indexing on Deployment

The following steps describe how indexing is triggered when a deployment occurs:

1. The `IndexingDeploymentListener` component detects the event.
2. The `IndexingDeploymentListener` examines the event to see the list of repositories being deployed.
3. The `IndexingDeploymentListener` compiles a list of the `EndecaIndexingOutputConfig` components that are associated with any of those repositories.
4. The `IndexingDeploymentListener` invokes the `LocalSynchronizationInvoker` component.
5. The `LocalSynchronizationInvoker` looks at the list of `EndecaIndexingOutputConfig` components and compiles a list of `SimpleIndexingAdmin` components that are associated with any of the `EndecaIndexingOutputConfig` components.
6. The `LocalSynchronizationInvoker` triggers an indexing job on each `SimpleIndexingAdmin` component in the list.

Note that the lists of `EndecaIndexingOutputConfig` and `SimpleIndexingAdmin` components are not configured explicitly. Instead, the `SimpleIndexingAdmin` components are automatically registered with the

LocalSynchronizationInvoker, and the EndecaIndexingOutputConfig components are automatically registered with the LocalSynchronizationInvoker and the IndexingDeploymentListener.

Viewing Records in the Component Browser

For debugging purposes, you can use the Component Browser of the Dynamo Server Admin to view records without submitting them to Guided Search. To do this, access the page for a component that generates records and follow the instructions below.

Note that the records displayed reflect the naming conventions and data formats used with Forge-based deployment templates, because the renaming and conversions done by DimensionDocumentSubmitter and ConfigImportDocumentSubmitter do not occur. Therefore, if you are using a CAS-based deployment template, the displayed records for CategoryToDimensionOutputConfig, RepositoryTypeDimensionExporter, and SchemaExporter may not match the records actually submitted to Guided Search.

ProductCatalogOutputConfig or CategoryToDimensionOutputConfig

The pages for the ProductCatalogOutputConfig and CategoryToDimensionOutputConfig components include a Test Document Generation section that you can use to view the output for a single repository item:

Fill in the repository ID of a product item (for the ProductCatalogOutputConfig component) or a category item (for the CategoryToDimensionOutputConfig component), and click Generate. The page will display the output records.

Click the Show Indexing Output Properties link to see descriptions of how the GSA repository-item properties are renamed in the Guided Search records, based on the values of various EndecaIndexingOutputConfig properties. (See the EndecaIndexingOutputConfig Components (page 28) section for information about these properties.)

RepositoryTypeDimensionExporter or SchemaExporter

The pages for the RepositoryTypeDimensionExporter and SchemaExporter components include a Show XML Output link. Each of these components produces a single output for the entire catalog. Click the link to view the output from the component.

5 Configuring EndecaIndexingOutputConfig Definition Files

This chapter describes various elements and attributes of `EndecaIndexingOutputConfig` XML definition files that you can use to control the content of the output records created from the product catalog. It includes the following sections:

Definition File Format

An `EndecaIndexingOutputConfig` indexing definition file begins with a top-level `item` element that specifies the item descriptor to create records from, and then lists the properties of that item type to include. The properties appear as `property` elements within a `properties` element.

The top-level `item` element in the definition file can contain child `item` elements for properties that refer to other repository items (or arrays, Collections, or Maps of repository items). Those child `item` elements in turn can contain `property` and `item` elements themselves.

The following example shows a simple definition file for indexing a product catalog repository:

```
<item item-descriptor-name="product" is-document="true">
  <properties>
    <property name="creationDate" type="date"/>
    <property name="brand" is-dimension="true" type="string"
      text-searchable="true"/>
    <property name="description" text-searchable="true"/>
    <property name="longDescription" text-searchable="true"/>
    <property name="displayName" text-searchable="true"/>
  </properties>

  <item is-multi="true" property-name="childSKUs">
    <properties>
```

```
      <property name="quantity" type="integer"/>
      <property name="description" text-searchable="true"/>
      <property name="displayName" text-searchable="true"/>
      <property name="color" is-dimension="true" type="string"
         text-searchable="true"/>
    </properties>

  <item is-multi="true" property-name="parentCategories"
      parent-property="childProducts">
    <properties>
      <property name="description" text-searchable="true"/>
      <property name="longDescription" text-searchable="true"/>
      <property name="displayName" text-searchable="true"/>
    </properties>
  </item>
</item>
```

Note that in this example, the `is-document` attribute of the top-level `item` element is set to `true`. This attribute specifies that a record should be generated for each item of that type (in this case, each `product` item). This means that each record indexed by Guided Search corresponds to a product, so that when a user searches the catalog, each individual result returned represents a product. The definition file specifies that each output record should include information about the product's parent categories and child SKUs (as well as the product itself), so that users can search category or SKU properties in addition to product properties.

If, instead, you want to generate a separate record per `sku` item, you set `is-document` to `true` for the `childSKUs` `item` element and to `false` for the `product` `item` element. In that case, the product properties (such as `brand` in the example above) are repeated in each record.

When you configure the Guided Search integration in CIM, you select whether to index by product or SKU. Your selection determines whether certain application modules are included in your EAR files. These modules configure the `is-document` attributes and other related settings appropriately for the option you select. See Oracle Commerce Platform Modules (page 7) for information about these modules.

Automatically Included Properties

In addition to the properties you specify in the definition file, records generated by the Oracle Commerce Platform also automatically include a few special properties that identify the document-level repository items represented in the records. These properties initially include a dollar-sign ($) character in their names, but are renamed for inclusion in the submitted records, because this character is not valid in Guided Search property names. These renamed output properties include the following:

- `record.id` – Output name of the `$docId` property, whose value is used to uniquely identify a record. The output name is specified through the `recordIdName` property of the `/atg/endeca/ApplicationConfiguration` component.
- `record.source` – Output name of the `$repository.repositoryName` property, whose value identifies the name of the source repository (for example, `ProductCatalog`).
- `record.type` – Output name of the `$itemDescriptor.itemDescriptorName` property, whose value identifies the repository item type used to generate the record. This property is included for the document-level item type and any subtypes of that item type. (For example, in Oracle Commerce Reference Store, `record.type` values appear for the `sku` item type and the `clothing-sku` and `furniture-sku` subtypes.)
- *`item-type`*`.siteId` – Output name for *`item-type`*`.$siteId` properties, which contain repository IDs for the sites the items are associated with. See Including siteId Properties (page 56) for more information.

- *item-type*.url and *item-type*.baseUrl – Output names for *item-type*.$url and *item-type*.$baseUrlproperties, which contain the URLs for the repository items the records represent. The difference between an *item-type*.url property and the corresponding *item-type*.baseUrl property (such as product.url and product.baseUrl) is that the url property includes query parameters and the baseUrl property does not. This means that if a VariantProducer is used to generate multiple records from the same repository item, product.baseUrl will be the same for each record, but the product.url query parameters will differ between records, making it possible to distinguish each record from the others.

If you want to include any of these properties for item types other than the document-level one, you can add them through the definition file. For example, if you enable SKU-based indexing by including the DCS.Endeca.Index.SKUIndexing module, some of these properties are output for products as well as SKUs, because they are explicitly declared in the ProductCatalogOutputConfig definition file.

You may also want to explicitly declare properties even if they are included automatically, so you can specify certain attributes (such as setting the is-dimension property to true, as discussed in the next section) or override the default output name. You can also suppress the inclusion of automatically included properties by setting the suppress attribute to true, as discussed in Suppressing Properties (page 56).

In addition to the properties listed above, which are output only for the document-level item type, *item-type*.repositoryId properties are automatically included for all item types included in the records. These properties are the output names for *item-type*.$repositoryId properties, which contain the repository IDs for the repository items the records represent. For example, a record for a product might include product.repositoryId, sku.repositoryId, and allAncestors.repositoryId properties.

Specifying Guided Search Schema Attributes

You can use various attributes of the property element to specify the way properties of repository items should be treated in the MDEX. The SchemaExporter component then uses the values of these attributes in the schema configuration it creates.

To specify the data type of a property, you use the type attribute. The value of this attribute can be date, string, boolean, integer, or float. For example:

```
<property name="quantity" type="integer"/>
```

If a type value is not specified, it defaults to string.

You can designate a property as searchable, as a dimension, or both. To make a property searchable, set the text-searchable attribute to true. To make a property a Guided Search dimension, set the is-dimension attribute to true. In the following example, the color property is both a dimension and searchable:

```
<property name="color" is-dimension="true" text-searchable="true"/>
```

If is-dimension is true, you can use the multiselect-type attribute to specify whether the customer can select multiple values of the dimension at the same time. The value of this attribute can be multi-or (combine using Boolean OR), multi-and (combine using Boolean AND), or none (the default, meaning multiselect is not supported for this dimension). For example:

```
<property name="brand" is-dimension="true" multiselect-type="multi-or"/>
```

Multiselect logic works as follows:

- Combining with Boolean OR returns results that match any of the selected values. For example, for a `color` dimension, if the user selects `yellow` and `orange`, a given item is returned if its `color` value is `yellow` or `orange`.
- Combining with Boolean AND returns results that match all of the selected values. For example, suppose a product representing a laser printer has a `paperSizes` property that is an array of the paper sizes the printer accepts, and you have a dimension based on this property. If the user selects `A4` and `letter` for this dimension, a given item is returned only if its `paperSizes` property includes both `letter` and `A4`.

Automatically Generating Dimension Values

If `is-dimension` is `true` for a repository item property, by default Guided Search examines the data and automatically generates non-hierarchical dimension values for the values of that property. For example, if the `color` property has values of `orange`, `yellow`, and `blue`, three dimension values are generated, representing the values of the property.

For a hierarchical dimension, though, the dimension value records must be explicitly created by the Guided Search integration. This is done by the `CategoryToDimensionOutputConfig` component (for the product categories) and the `RepositoryTypeDimensionExporter` component (for the catalog repository item-type hierarchy).

To prevent automatic generation of dimension values for a property, set the `autogen-dimension-values` attribute to `false`. For example, the dimension for the repository item-type hierarchy is defined like this:

```
<property autogen-dimension-values="false"
  name="$itemDescriptor.itemDescriptorName" is-dimension="true"/>
```

Specifying Properties for Indexing

This section discusses how to specify various properties of catalog items for inclusion in the MDEX, and options for how these properties should be handled.

Specifying Multi-Value Properties

In most cases, you specify a multi-value property, such as an array or Collection, using the `property` element, just as you specify a single-value property. In the following example, the `features` property stores an array of Strings:

```
<properties>
  <property name="creationDate" type="date"/>
  <property name="brand" is-dimension="true" type="string"
    text-searchable="true"/>
  <property name="displayName" type="string" text-searchable="true"/>
  <property name="features" type="string" text-searchable="true"/>
</properties>
```

Notice that `features` is specified in the same way as `creationDate`, `brand`, and `displayName`, which are all single-value properties. The output will include a separate entry for each value in the `features` array.

If a property is an array or Collection of repository items, you specify it using the `item` element, and set the `is-multi` attribute to `true`. For example, in a product catalog, a `product` item will typically have a multi-valued `childSKUs` property whose values are the various SKUs for the product. You might specify the property like this:

```
<item property-name="childSKUs" is-multi="true">
  <properties>
    <property name="color" is-dimension="true" type="string"
      text-searchable="true"/>
    <property name="description" type="string" text-searchable="true"/>
  </properties>
</item>
```

If you index by product, the output records will include the `color` and `description` value for each of the product's SKUs.

Specifying Map Properties

To specify a Map property, you use the `item` element, set the `is-multi` attribute to `true`, and use the `map-iteration-type` attribute to specify how to output the Map entries. If the Map values are primitives or Strings, set `map-iteration-type` to `wildcard`, as in this example:

```
<item property-name="personalData" is-multi="true" map-iteration-type="wildcard">
  <properties>
    <property name="*" type="string"/>
  </properties>
</item>
```

In the output, the Map keys are treated as subproperties of the Map property, and the Map values are treated as the values of these subproperties. All of the Map entries are included in the output. So, for example, the output from the definition file entry shown above might look like this:

```
<PROP NAME="personalData.firstName">
  <PVAL>Fred</PVAL>
</PROP>
<PROP NAME="personalData.age">
  <PVAL>37</PVAL>
</PROP>
<PROP NAME="personalData.height">
  <PVAL>68</PVAL>
</PROP>
```

If you want to output only a subset of the Map entries, explicitly specify the keys to include, rather than using the wildcard character (`*`). For example:

```
<item property-name="personalData" is-multi="true" map-iteration-type="wildcard">
  <properties>
    <property name="firstName" type="string" text-searchable="true"/>
    <property name="height" type="string"/>
  </properties>
</item>
```

Maps of Repository Items

If the Map values are repository items, set `map-iteration-type` to `values`, and specify the properties of the repository item that you want to output. For example, suppose you want to index a `productInfos` Map property whose keys are product IDs and whose values are `productInfo` items:

```
<item property-name="productInfos" is-multi="true" map-iteration-type="values">
  <properties>
    <property name="displayName" type="string" text-searchable="true"/>
    <property name="size" type="integer" is-dimension="true"/>
  </properties>
</item>
```

The output will include `displayName` and `size` tags for each `productInfo` item in the Map. In this case, the Map keys are ignored, the properties of the repository items are treated as subproperties of the Map property, and the values of the items are treated as the values of the subproperties. The output looks like this:

```
<PROP NAME="productInfos.displayName">
  <PVAL>Funny Hat</PVAL>
</PROP>
<PROP NAME="productInfos.size">
  <PVAL>8</PVAL>
</PROP>
<PROP NAME="productInfos.displayName">
  <PVAL>Clown Shoes</PVAL>
</PROP>
<PROP NAME="productInfos.size">
  <PVAL>14</PVAL>
</PROP>
```

Specifying Properties of Item Subtypes

A repository item type can have subtypes that include additional properties that are not part of the base item type. This feature is commonly used in the Oracle Commerce Platform catalog for the SKU item type. A SKU subtype might add properties that are specific to certain SKUs but which are not relevant for other SKUs.

When you list properties to index, you can use the subtype attribute of the property element to specify properties that are unique to a specific item subtype. For example, suppose you have a furniture-sku subtype that adds properties specific to furniture SKUs. You might specify your SKU properties like this:

```
<item property-name="childSKUs">
  <properties>
    <property name="description" type="string" text-searchable="true"/>
    <property name="color" type="string" text-searchable="true"
      is-dimension="true"/>
    <property name="woodFinish" subtype="furniture-sku" type="string"
      text-searchable="true"/>
  </properties>
</item>
```

This specifies that the `description` and `color` properties should be included in the output for all SKUs, but for SKUs whose subtype is `furniture-sku`, the `woodFinish` property should also be included.

The `item` element also has a `subtype` attribute for specifying a subtype-specific property whose value is a repository item. If `woodFinish` is a repository item, the example above would look something like this:

```
<item property-name="childSKUs">
  <properties>
    <property name="description" type="string" text-searchable="true"/>
    <property name="color" type="string" text-searchable="true"
      is-dimension="true"/>
  </properties>
  <item property-name="woodFinish" subtype="furniture-sku"/>
    <properties>
      <property name="texture" type="string" text-searchable="true"/>
      <property name="stainType" type="string" text-searchable="true"/>
    </properties>
  </item>
</item>
```

Specifying a Default Property Value

You may find it useful to specify a default value for certain indexed properties. For example, suppose you are indexing address data, and for some addresses no value appears in the repository for the `city` property. In these cases, you could set the property value in the index to be "city unknown." A user could then search for this phrase and return the addresses whose `city` property is null.

To set a default value, you use the `default-value` attribute of the `property` element. For example:

```
<property name="city" type="string" text-searchable="true"
  default-value="city unknown"/>
```

Specifying Non-Repository Properties

When you index a repository, you can include in the index additional properties that are not part of the repository itself. For example, you might want to include a `creationDate` property to record the current time when a record is created. The value for this property could be generated by a custom property accessor that invokes the Java `Date` class.

To specify a property like this, use the `is-non-repository-property` attribute of the `property` element. This attribute indicates that the property is not actually stored in the repository, and prevents warnings from being thrown when the `EndecaIndexingOutputConfig` component starts up. Note that you must also specify a custom property accessor that is responsible for obtaining the property values:

```
<property name="creationDate" is-non-repository-property="true"
   type="date" property-accessor="dateAccessor"/>
```

If no actual property accessor is needed, set the `property-accessor` attribute to `null`. For example, you might do this if you have a default value that you always want to use for the property:

```
<property name="creationDate" is-non-repository-property="true"
   type="date" default-value="Mon Mar 15 16:07:15 EDT 2010"
```

```
   property-accessor="null"/>
```

See Using Property Accessors (page 61) for more information about custom property accessors.

Suppressing Properties

The output records automatically include certain special repository item properties, as discussed in Automatically Included Properties (page 50). These properties provide information that identifies the repository items represented in a record, and they are indicated by a dollar-sign ($) prefix: for example, `$repositoryId` and `$url`. The dollar signs are removed by default in the output records, because Guided Search property names cannot include them, and in some cases the properties are renamed.

You may want to return these properties in search results, to enable accessing the indexed repository and repository items in page code. If you do not need a property, it is a good idea to exclude it from the index, as it may significantly increase the size of the index. For example, most of these properties are included only for the document-level item type, but the `$repositoryId` property is included for every item type. To suppress it for a specific item type, use the `suppress` attribute. For example:

```
<item property-name="parentCategories" is-document="false">
  <properties>
    <property name="$repositoryId" suppress="true"/>
  </properties>
</item>
```

Including siteId Properties

If you are using the Oracle Commerce Platform multisite support, many of the item types in the catalog repository have a context membership property (named `siteIds` by default) whose value is a comma-separated list of the repository IDs of the sites an item appears on. For example, if you have three sites whose repository IDs are `siteA`, `siteB`, and `siteC`, and a certain item is available on `siteA` and `siteC` (but not `siteB`), the value of the item's context membership property would be `siteA,siteC`.

For the document-level item type, the records generated by the Oracle Commerce Platform include special *`item-type`*`.$siteId` properties that represent the repository item's context membership property. These *`item-type`*`.$siteId` properties are renamed to *`item-type`*`.siteId` in the generated records. The records include a separate *`item-type`*`.siteId` entry for each site listed in the context membership property. For example:

```
<PROP NAME="product.siteId">
  <PVAL>
    storeSiteUS
  </PVAL>
</PROP>
<PROP NAME="product.siteId">
  <PVAL>
    storeSiteDE
  </PVAL>
</PROP>
```

Note that the output records include entries only for sites that are listed in the `sitesToIndex` property of the `EndecaIndexingOutputConfig` component. For example, if the value of a item's context membership

property is `siteA`, `siteB`, `siteC`, but `sitesToIndex` lists only `siteB` and `siteC`, the record will not include an entry for `siteA`. If an item's context membership property is null, or if it lists only sites that are not listed in the `sitesToIndex` property, no record is generated for the item.

For information about context membership properties, see the *Multisite Administration Guide*.

Item Types Lacking a Context Membership Property

If an item type does not have a context membership property, another mechanism is needed for including *`item-type`*`.siteId` values in the generated records. In this situation, you can use the `sites-property-name` and `sitegroups-property-name` attributes of the `item` element to specify the names of the properties that hold references to sites and site groups in the site repository.

For example, the Commerce Reference Store location repository has a `location` item type that represents the geographic location of a physical store. This item type does not have a context membership property, but has `sites` and `siteGroups` properties that contain references to sites and site groups in the site repository. Commerce Reference Store indexes `location` items so customers can use guided navigation to find stores in specific locations. The indexing definition file includes:

```
<item item-descriptor-name="location" is-document="true"
  sites-property-name="sites" sitegroups-property-name="siteGroups">
```

Note that the specified properties must contain references to the actual site and site group items in the site repository, not the site ID strings.

Renaming an Output Property

By default, the name of a property in an output record is based on its name in the repository, with modifications applied based on the values of the `replaceWithTypePrefixes`, `prefixReplacementMap`, and `suffixReplacementMap` properties of the `EndecaIndexingOutputConfig` component. (See the EndecaIndexingOutputConfig Components (page 28) section for information about these properties.)

You can instead specify the output property name by using the `output-name` attribute of the `property` element. For example:

```
<property name="material" output-name="product.fabric"
  text-searchable="true" is-dimension="true"/>
```

Note that the exact `output-name` value you specify is used with no modifications. So in this example, the item-type prefix is explicitly included.

Translating Property Values

In some cases, the property values that you want to include in the index (and therefore in the generated records) may not be the actual values used in the repository. You may want to normalize values (for example, index the color values Rose, Vermilion, Crimson, and Ruby all as Red, so they are all treated as the same dimension value). Or you may want to translate values into another language (for example., index the color value Green as Vert, so when a customer searches for Vert, green items are returned).

To translate property values for indexing, you use the `translate` child element of the `property` element. The `translate` element has an `input` attribute for specifying a property value found in the repository, and an `output` attribute for specifying the value to translate this to in the output records. For example:

```
<property name="color" text-searchable="true" is-dimension="true">
  <translate input="Rose" output="Red"/>
  <translate input="Vermilion" output="Red"/>
  <translate input="Crimson" output="Red"/>
  <translate input="Ruby" output="Red"/>
</property>
```

The `property` element also has `prefix` and `suffix` child elements that you can use to append a text string before or after the output property values. For example, you can use the `suffix` element to add units to the property values:

```
<property name="length">
  <suffix value=" cm"/>
</property>
```

Note that the `prefix` and `suffix` values are concatenated to the property values exactly as specified, with no additional spaces. If you want spaces before the `suffix` string or after the `prefix` string, include the spaces in the `value` attribute, as in the example above.

You can use the `prefix`, `suffix`, and `translate` elements individually or in combination. The following example translates the size values S, M, and L to "size small," "size medium," and "size large," to make it easier for customers to search for specific sizes:

```
<property name="size" text-searchable="true" is-dimension="true">
  <prefix value="size "/>
  <translate input="S" output="small"/>
  <translate input="M" output="medium"/>
  <translate input="L" output="large"/>
</property>
```

Translating Based on Locale

The `prefix`, `suffix`, and `translate` elements all have optional `locale` attributes that allow you to specify different values for different locales. For example:

```
<property name="onSale" is-dimension="true">
  <translate locale="en_US" input="true" output="on sale"/>
  <translate locale="fr_FR" input="true" output="à la vente"/>
</property>
<property name="weight">
  <suffix locale="en_US" output=" grams"/>
  <suffix locale="fr_FR" output=" grammes"/>
</property>
```

When the records are generated, the `EndecaIndexingOutputConfig` component determines which tags to use based on the current locale. So if the locale is `en_US`, only the tags that specify that locale are applied.

Multilingual environments typically use the `LocaleVariantProducer`, which generates multiple records for each indexed item, one record for each locale specified in its `locales` array property. (See Using Variant Producers (page 63) for more information.) If the value of the `locales` array is `en_US`, `fr_FR`, two sets of records are generated, one using the `translate`, `prefix`, and `suffix` tags whose locale is `en_US`, and one using the tags whose locale is `fr_FR`.

If a tag does not specify a locale, that tag is used as the default when the current locale does not match any of the other tags. In the following example, Scarlet is translated to Rouge if the locale is `fr_FR`, but is translated to Red for any other locale:

```
<property name="color" text-searchable="true" is-dimension="true">
  <translate input="Scarlet" output="Red"/>
  <translate locale="fr_FR" input="Scarlet" output="Rouge"/>
</property>
```

Using Monitored Properties

By default, the `IncrementalLoader` determines which changes necessitate updates by monitoring the properties specified in the XML definition file. In some cases, however, the properties you want to monitor are not necessarily the ones that you want to output. This is especially the case if you are outputting derived properties, because these properties do not have values of their own.

For example, suppose you are indexing a `user` item type that has `firstName` and `lastName` properties, plus a `fullName` derived property whose value is formed by concatenating the values of `firstName` and `lastName`. You might want to output the `fullName` property, but to detect when the value of this property changes, you need to monitor (but not necessarily output) `firstName` and `lastName`.

You can do this by including a `monitor` element in your definition file to specify properties that should be monitored but not output. For example:

```
<properties>
  <property name="fullName" text-searchable="true"/>
</properties>
<monitor>
  <property name="firstName"/>
  <property name="lastName"/>
</monitor>
```

For information about derived properties, see the *Repository Guide*.

You can also monitor properties in a different repository from the one being indexed. For example, if you are using price lists, changes to `price` items in the price list repository may necessitate reindexing products or SKUs in the catalog repository that are referenced by these `price` items. The `atg.repository.search.indexing.listener.QueueingPropertiesChangeListener` class provides a mechanism for triggering reindexing of items in one repository based on changes to items in another repository. See QueueingPropertiesChangeListener (page 128) in the *Handling Price Lists* (page 125) chapter for more information.

Filtering Properties of Specific Repository Items

In some cases, you may want to output the values of a property for some repository items of a certain type but not for others of that type. For example, you may want to output the value of the `longDescription` property of most `product` items, but omit this property for a few specific `product` items.

The Oracle Commerce Platform includes an interface, `atg.repository.search.indexing.IndexingPropertyFilter`, for filtering properties of specific repository items. This interface defines a single method:

```
filterOutputProperties(RepositoryItem pItem,
  OutputProperty[] pOutputProperties)
```

This method is used to implement the logic that determines which property values to exclude from output records.

The Oracle Commerce Platform also includes a class that implements this interface, `atg.repository.search.indexing.filter.GSAPropertyFilter`. This class has two properties that are used to specify the property values to exclude:

idToType
A Map in which the keys are IDs of repository items and the values are their item types.

propsToFilter
A List of the properties of the items specified by `idToType` whose values should be excluded from the output. Note that the property names you supply should be the output names used in records, after any prefix or suffix replacement or property renaming.

Commerce Reference Store includes a component of this class, `/atg/commerce/endeca/index/GSAPropertyFilter`, that is configured as follows:

```
idToType=rootCategory=category,\
         homeStoreRootCategory=category
propsToFilter=allAncestors.displayName
```

To apply a `GSAPropertyFilter` component to an item type, you use the `filter` attribute of the `item` element in the `EndecaIndexingOutputConfig` definition file. For example, Commerce Reference Store adds this attribute to the `ancestorCategories` item specification in the definition file of the `ProductCatalogOutputConfig` component:

```
<item is-multi="true" property-name="ancestorCategories"
  filter="/atg/commerce/endeca/index/GSAPropertyFilter">
```

The `filterOutputProperties()` method of the `GSAPropertyFilter` class examines the `idToType` property to see which repository items to filter. In Commerce Reference Store, the property is configured so that filtering is done for the `category` items whose IDs are `rootCategory` and `homeStoreRootCategory`. It then uses the value of the component's `propsToFilter` property to determine which properties to exclude the values of. `propsToFilter` is set to `allAncestors.displayName`, so the value of this property is not output for the `rootCategory` and `homeStoreRootCategory` categories. (This is done to address a problem in Commerce Reference Store where searching for "root" would return every product in the catalog repository, because that word appears in the value of the `displayName` property of both items, and every product has one of these categories as an ancestor category.)

You can create other components of class `GSAPropertyFilter` and configure the `idToType` and `propsToFilter` properties to filter different item types and properties, or you can implement different filtering logic by writing your own class that implements the `IndexingPropertyFilter` interface.

6 Customizing the Output Records

This chapter describes interfaces and classes that can be used to customize the records created by the Guided Search integration. It discusses the following topics:

In addition to the classes described here, the Guided Search integration includes property accessors and variant producers for accessing price data in price lists. See the *Handling Price Lists* (page 125) chapter for more information.

For additional information about the classes and interfaces described in this chapter, see the *ATG Platform API Reference*.

Using Property Accessors

Property values are read from the product catalog through an implementation of the `atg.repository.search.indexing.PropertyAccessor` interface. For most properties, the default is to use the `atg.repository.search.indexing.PropertyAccessorImpl` class, which just invokes the `RepositoryItem.getPropertyValue()` method. You can write your own implementations of `PropertyAccessor` that use custom logic for determining the values of properties that you specify. The simplest way to do this is to subclass `PropertyAccessorImpl`.

In an `EndecaIndexingOutputConfig` definition file, you can specify a custom property accessor for a property by using the `property-accessor` attribute. For example, suppose you have a Nucleus component named `/mystuff/MyPropertyAccessor`, of a custom class that implements the `PropertyAccessor` interface. You can specify it in the definition file like this:

```
<property name="myProperty"
  property-accessor="/mystuff/MyPropertyAccessor"/>
```

The value of the `property-accessor` attribute is the absolute path of the Nucleus component. To simplify coding of the definition file, you can map `PropertyAccessor` Nucleus components to simple names, and

use those names as the values of `property-accessor` attributes. For example, if you map the `/mystuff/MyPropertyAccessor` component to the name `myAccessor`, the above tag becomes:

```
<property name="myProperty" property-accessor="myAccessor"/>
```

You can perform this mapping by setting the `propertyAccessorMap` property of the `EndecaIndexingOutputConfig` component. This property is a Map in which the keys are the names and the values are `PropertyAccessor` Nucleus components that the names represent. For example:

```
propertyAccessorMap+=\
  myAccessor=/mystuff/MyPropertyAccessor
```

FirstWithLocalePropertyAccessor

The `atg.repository.search.indexing.accessor` package includes a subclass of `PropertyAccessorImpl` named `FirstWithLocalePropertyAccessor`. This property accessor works only with derived properties that are defined using the `firstWithLocale` derivation method. `FirstWithLocalePropertyAccessor` determines the value of the derived property by looking up the `currentDocumentLocale` property of the `Context` object. Typically, this property is set by the `LocaleVariantProducer`, as described in Accessing the Context Object (page 64).

You can specify this property accessor in your definition file using the attribute value `firstWithLocale`. (Note that you do not need to map this name to the property accessor in the `propertyAccessorMap`.) For example:

```
<property name="displayName" property-accessor="firstWithLocale"/>
```

For information about the `firstWithLocale` derivation method, and about derived properties in general, see the *Repository Guide*.

LanguageNameAccessor

The `atg.endeca.index.accessor.LanguageNameAccessor` class, which is a subclass of `atg.repository.search.indexing.PropertyAccessorImpl`, returns the name of the language that a record is in. The Guided Search integration includes a component of this class, `/atg/endeca/index/accessor/LanguageNameAccessor`, which the `ProductCatalogOutputConfig` uses to obtain the value of the `product.language` property:

```
<property name="language" is-dimension="true" type="string"
  property-accessor="/atg/endeca/index/accessor/LanguageNameAccessor"
  output-name="product.language" is-non-repository-property="true"/>
```

GenerativePropertyAccessor

The `atg.repository.search.indexing.accessor` package includes a subclass of `PropertyAccessorImpl` named `GenerativePropertyAccessor`. This is an abstract class that adds the ability to generate multiple property names and associated values for a single property tag in the indexing definition file.

You can write your own subclass of `GenerativePropertyAccessor`. Your subclass must implement the `getPropertyNamesAndValues` method. This method returns a Map in which each key is a property name, and the corresponding Map value contains the value to be associated with the property name.

Category Dimension Value Accessors

Several property accessors are used by the `CategoryToDimensionOutputConfig` component to extract the values of various dimension value attributes from the data structures created by the `CategoryTreeService` component.

A component of class `atg.endeca.index.accessor.ConstantValueAccessor`, `/atg/commerce/endeca/index/accessor/DimensionSpecPropertyAccessor`, obtains the value of the `dimval.dimension_spec` attribute, which is a unique identifier for the dimension (typically `product.category`).

Several components of class `atg.commerce.endeca.index.dimension.CategoryNodePropertyAccessor`, also in the `/atg/commerce/endeca/index/accessor/` Nucleus folder, obtain the values of various dimension value attributes. The following table lists these property accessors and describes the attributes they obtain values for. Note that the property names shown in the table are appropriate for use with CAS-based Guided Search deployment templates, and assume that the name changes specified in `propertyNameReplacementMap` property of the `DimensionDocumentSubmitter` component have been applied. See RecordStoreDocumentSubmitter (page 37) for more information.

Property Accessor	Property
`RootCatalogPropertyAccessor`	`category.rootCatalogId` -- The repository ID of the root catalog the category belongs to (e.g., `masterCatalog`).
`SpecPropertyAccessor`	`dimval.spec` -- A unique identifier for the dimension value that includes the path information to distinguish it from other dimension values for the same category (e.g., `cat10016.cat10014`).
`QualifiedSpecPropertyAccessor`	`Endeca.Id` -- A qualified identifier for the dimension value consisting of the `dimval.dimension_name` value and the `dimval.spec` value (e.g., `product.category:cat10016.cat10014`).
`ParentSpecPropertyAccessor`	`dimval.parent_spec` -- A reference to the category's parent category (e.g., `cat10016`).
`DisplayOrderPropertyAccessor`	`dimval.display_order` -- An integer specifying the order the category is displayed in, relative to its sibling categories.

Using Variant Producers

By default, for the repository item type designated by the `is-document` attribute, the `EndecaIndexingOutputConfig` component generates one record per item. In some cases, however, you may

want to generate more than one record for each repository item. For example, suppose you have a repository whose text properties are stored in both French and English, and the language displayed is determined by the user's locale setting. In this case you typically want to create two records from each repository item, one with the text content in French, and the other one in English.

To handle situations like this, the Oracle Commerce Platform provides an interface named `atg.repository.search.indexing.VariantProducer`. You can write your own implementations of the `VariantProducer` interface, or you can use implementations included with the Oracle Commerce Platform. This interface defines a single method, `prepareNextVariant()`, for determining the number and type of variants to produce. Depending on how your repository is organized, implementations of this method can use a variety of approaches for determining how to generate variant records.

LocaleVariantProducer

The Guided Search integration includes an implementation of the `VariantProducer` interface, `atg.repository.search.indexing.producer.LocaleVariantProducer`, for generating variant records for different locales. It also includes a component of this class, `/atg/commerce/search/LocaleVariantProducer`.

The `LocaleVariantProducer` class has a `locales` property where you specify the list of locales to generate variants for. By default, this property is linked to the value of the `locales` property of the `/atg/endeca/ApplicationConfiguration` component:

```
locales^=/atg/endeca/ApplicationConfiguration.locales
```

You specify the `VariantProducer` components to use by setting the `variantProducers` property of the `EndecaIndexingOutputConfig` component. Note that this property is an array; you can specify any number of `VariantProducer` components. For example:

```
variantProducers=/atg/commerce/search/LocaleVariantProducer,\
  /mystuff/MyVariantProducer
```

If you specify multiple variant producers, the `EndecaIndexingOutputConfig` generates a separate variant for each possible combination of values of the variant criteria. For example, suppose you use the configuration shown above and `MyVariantProducer` creates three variants (1, 2, and 3). The total number of variants generated for each repository item is six (French 1, English 1, French 2, English 2, French 3, and English 3).

Accessing the Context Object

Classes that implement the `PropertyAccessor` or `VariantProducer` interface must be stateless, because they can be accessed by multiple threads at the same time. Rather than maintaining state themselves, these classes instead use an object of class `atg.repository.search.indexing.Context` to store state information and to pass data to each other. The `Context` object contains the current list of parent repository items that were navigated to reach the current item, the current URL (if any), the current collected output values (if any), and status information.

One of the main uses of the `Context` object is to store information used to determine what variant to generate next. For example, each time a new record is generated, the `LocaleVariantProducer` uses the next value in its `locale` array to set the `currentDocumentLocale` property of the `Context` object. A `PropertyAccessor` instance might read the `currentDocumentLocale` property and use its current value to determine the locale to use for the property.

Note that classes that implement the `PropertyFormatter` or `PropertyValuesFilter` interface (described below) are applied after all of the output properties have been gathered, so these classes do not have access to the `Context` object.

For more information about the `Context` object, see the *ATG Platform API Reference*.

CategoryPathVariantProducer

The `/atg/commerce/endeca/index/CategoryPathVariantProducer` component is used by the `CategoryToDimensionOutputConfig` component to produce multiple records per category (one record for each unique path computed by `CategoryTreeService`). The `CategoryPathVariantProducer` component is of class `atg.commerce.endeca.index.dimension.CategoryPathVariantProducer`, which implements the `atg.repository.search.indexing.VariantProducer` interface. In each record this variant producer creates, the value of the record's `dimval.spec` property is the unique pathname that the record represents. For example:

The `CategoryPathVariantProducer` component is added to the `CategoryToDimensionOutputConfig` component's variantProducers property by default:

```
variantProducers+=\
   CategoryPathVariantProducer
```

See the CategoryTreeService Class (page 20) section for more information about how category path variants are computed.

CustomCatalogVariantProducer

In addition to the `category`, `product`, and `sku` items, the catalog repository includes `catalog` items that represent different hierarchies of categories and products. Each user is assigned one catalog, and sees the navigational structure, products and SKUs, and property values associated with that catalog. A given product may appear in multiple catalogs. The `product` repository item type includes a `catalogs` property whose value is a Set of the catalogs the product is included in.

Depending on how your catalog repository is configured, the property values of individual categories, products, or SKUs may vary depending on the catalog. If so, when you index the catalog, you may need to generate multiple records for each product or SKU (one for each catalog the item is included in).

To support creation of multiple records per product or SKU, the Guided Search integration uses the `/atg/commerce/search/CustomCatalogVariantProducer` component. This component is of class `atg.commerce.search.producer.CustomCatalogVariantProducer`, which implements the `atg.repository.search.indexing.VariantProducer` interface. The variant producer iterates through each catalog individually, so that each record contains only the property values associated with a single catalog.

The `CustomCatalogVariantProducer` component is added to the `ProductCatalogOutputConfig` component's variantProducers property by default:

```
variantProducers+=\
   CustomCatalogVariantProducer
```

The mechanism used for retrieving catalog-specific property values differs depending on the property. For `category`, `product`, or `sku` item properties that use the `atg.commerce.dp.CatalogMapDerivation` class to derive catalog-specific values, the correct values are automatically obtained by that class.

To get the value of the `catalogs` property of the `product` item, the `ProductCatalogOutputConfig` component is configured by default to use the `/atg/commerce/search/CustomCatalogPropertyAccessor` component. This component is of class `atg.commerce.search.producer.CustomCatalogPropertyAccessor`, which implements the `atg.repository.search.indexing.PropertyAccessor` interface. This accessor returns, for each record, only the specific catalog the record applies to. The accessor is specified in the `/atg/commerce/endeca/index/product-sku-output-config.xml` definition file:

```
<item is-multi="true" property-name="catalogs"
   property-accessor="customCatalog">
```

The `CustomCatalogPropertyAccessor` component is mapped to the name `customCatalog` by the `ProductCatalogOutputConfig` component's `propertyAccessorMap` property:

```
propertyAccessorMap+=\
   customCatalog=CustomCatalogPropertyAccessor
```

UniqueSiteVariantProducer

If you want to create separate records for each site, you can do so by using the `/atg/search/repository/UniqueSiteVariantProducer` component. This component is of class `atg.commerce.endeca.index.producer.CommerceUniqueSiteVariantProducer`, which implements the `atg.repository.search.indexing.VariantProducer` interface.

`UniqueSiteVariantProducer` creates a separate record for each site that meets both of these criteria:

- The ID of the site is included in the `siteIds` property of the item being indexed.
- The site is listed in the `sitesToIndex` property of the `EndecaIndexingOutputConfig` component that invokes the variant producer.

For example, if you are indexing by product and the value of a product's `siteIds` property is `siteE`, `siteF`, `siteG`, and the `sitesToIndex` property is set to sites B, E, and F, `UniqueSiteVariantProducer` creates two records, one for site E and one for site F. The records are virtually identical, except that each one has a different value for the `siteId` property.

To use the `UniqueSiteVariantProducer`, add it to the `ProductCatalogOutputConfig` component's `variantProducers` property:

```
variantProducers+=\
    /atg/search/repository/UniqueSiteVariantProducer
```

MultipleSiteVariantProducer

If you are using the `GroupingApplicationRoutingStrategy`, as described in the *Routing* (page 9)chapter, you need to ensure that separate records are created for each EAC application. For example, if you have a total of five sites, with one EAC application handling the data from two of the sites and another handling the data for the other three, two sets of records should be created, each reflecting the group of sites handled by one of the EAC applications.

To create these records, add the /atg/endeca/index/producer/MultipleSiteVariantProducer component to the ProductCatalogOutputConfig component's variantProducers property. The MultipleSiteVariantProducer component is of class atg.endeca.index.producer.MultipleSiteVariantProducer, which is a subclass of the atg.endeca.index.producer.GroupingVariantProducer abstract class. MultipleSiteVariantProducer creates a separate variant for each grouping listed in the GroupingApplicationRoutingStrategy component's applicationGroupingMap property. For example, suppose this property is set to:

```
applicationGroupingMap=\
  footwearStores=shoeSiteUS|shoeSiteCanada,\
  apparelStores=clothesSiteUS|clothesSiteUK|clothesSiteCanada
```

MultipleSiteVariantProducer will create two sets of records, one for each EAC application listed in the Map keys.

Using Property Formatters

If a property takes an object as its value, the data loader must convert that object to a string to include it in an output record. The PropertyFormatter interface defines methods for performing this conversion.

By default, the data loaders use the implementation class atg.endeca.index.formatter.EndecaPropertyFormatter. This class invokes the object's getLong() method for numbers or getTime() method for dates; for booleans, it converts the value to the String "0" (false) or "1" (true). For other objects, it calls the object's toString() method.

You can write your own implementations of PropertyFormatter that use custom logic for performing the conversion. The simplest way to do this is to subclass EndecaPropertyFormatter.

In an EndecaIndexingOutputConfig definition file, you can specify a custom property formatter by using the formatter attribute. For example, suppose you have a Nucleus component named /mystuff/MyPropertyFormatter, of a custom class that implements the PropertyFormatter interface. You can specify it in the definition file like this:

```
<property name="myProperty" formatter="/MyStuff/MyPropertyFormatter"/>
```

The value of the formatter attribute is the absolute path of the Nucleus component. To simplify coding of the definition file, you can map PropertyFormatter Nucleus components to simple names, and use those names as the values of formatter attributes. For example, if you map the /mystuff/MyPropertyFormatter component to the name myFormatter, the above tag becomes:

```
<property name="myProperty" formatter="myFormatter"/>
```

You can perform this mapping by setting the formatterMap property of the EndecaIndexingOutputConfig component. This property is a Map in which the keys are the names and the values are PropertyFormatter Nucleus components that the names represent.

Using Property Value Filters

In some cases, it is useful to filter a set of property values before outputting a record. For example, suppose each record represents a product whose SKUs all have the same display name. Rather than outputting the `displayName` property value of each SKU, you could include `displayName` in the record only once, by using a filter that removes duplicate property values.

The `PropertyValuesFilter` interface defines a method for filtering property values. The `atg.repository.search.indexing.filter` package includes several implementations of this interface:

- `UniqueFilter` removes duplicate property values, returning only the unique values.
- `ConcatFilter` concatenates all of the property values into a single string.
- `UniqueWordFilter` removes any duplicate words in the property values, and then concatenates the results into a single string.
- `HtmlFilter` removes any HTML markup from the property values.

This section provides information about what these filters do and when they're appropriate.

In an `EndecaIndexingOutputConfig` definition file, you can specify property filters by using the `filter` attribute. Note that you can use multiple filters on the same property. The value of the `filter` attribute is a comma-separated list of Nucleus components. The component names must be absolute pathnames.

To simplify coding of the definition file, you can map `PropertyValuesFilter` Nucleus components to simple names, and use those names as the values of `filter` attributes. You can perform this mapping by setting the `filterMap` property of the `EndecaIndexingOutputConfig` component. This property is a Map in which the keys are the names and the values are `PropertyFilter` Nucleus components that the names represent.

Note, however, that you do not need to perform this mapping to use the `UniqueFilter`, `ConcatFilter`, `UniqueWordFilter`, or `HtmlFilter` class. These classes are mapped by default to the following names:

Filter Class	Name
`UniqueFilter`	`unique`
`ConcatFilter`	`concat`
`UniqueWordFilter`	`uniqueword`
`HtmlFilter`	`html`

So, for example, you can specify `UniqueFilter` like this:

```
<property name="color" filter="unique"/>
```

UniqueFilter

You may be able to reduce the size of your index by filtering the property values to remove redundant entries. For example, suppose a record represents a product whose SKUs have a `size` property, with values of small,

medium, and large; multiple SKUs have the same `size` value, and are differentiated by other properties (e.g., `color`). The entries for `size` in a record might be:

```
<PROP NAME="sku.size">
  <PVAL>medium</PVAL>
  <PVAL>large</PVAL>
  <PVAL>medium</PVAL>
  <PVAL>small</PVAL>
  <PVAL>medium</PVAL>
  <PVAL>small</PVAL>
</PROP>
```

By filtering out redundant entries, you can reduce this to:

```
<PROP NAME="sku.size">
  <PVAL>medium</PVAL>
  <PVAL>large</PVAL>
  <PVAL>small</PVAL>
</PROP>
```

To automatically perform this filtering, specify the `UniqueFilter` class in the XML definition file:

```
<property name="size" filter="unique"/>
```

As a general rule, it is a good idea to specify the `unique` filter for a property if multiple items in a record may have identical values for that property. If you specify this filter for a property and every value of that property in a record is unique (or if only one item with that property appears in the record), the `unique` filter will have no effect on the record (either negative or positive). However, executing this filter increases processing time to create the record, so it is a good idea to specify it only for properties that will benefit from it.

ConcatFilter

You may also be able to reduce the size of your index by concatenating the values of text properties. For example, suppose each record represents a product whose SKUs have a `color` property, with values of red, green, blue, and yellow. The entries for `color` in a record might be:

```
<PROP NAME="sku.color">
  <PVAL>red</PVAL>
  <PVAL>green</PVAL>
  <PVAL>blue</PVAL>
  <PVAL>yellow</PVAL>
</PROP>
```

By concatenating the values, you can reduce this to:

```
<PROP NAME="sku.color">
  <PVAL>red green blue yellow</PVAL>
</PROP>
```

To combine these values into a single tag, specify the `ConcatFilter` class in the XML definition file:

```
<property name="color" filter="concat"/>
```

This setting invokes an instance of the `atg.repository.search.indexing.filter.ConcatFilter` class. Note that you do not need to create a Nucleus component to use this filter.

You can use both the `unique` and `concat` filters on the same property, by setting the value of the `filter` attribute to a comma-separated list. The filters are invoked in the order that they are listed, so it is important to put the `unique` filter first for it to have an effect. For example:

```
<property name="color" filter="unique,concat"/>
```

UniqueWordFilter

The `atg.repository.search.indexing.filter.UniqueWordFilter` class removes any duplicate words in the property values, and then concatenates the results into a single string. For example, suppose a product's SKUs have a `size` property, and the resulting entries in a record are:

```
<PROP NAME="sku.size">
  <PVAL>medium</PVAL>
  <PVAL>large</PVAL>
  <PVAL>x large</PVAL>
  <PVAL>xx large</PVAL>
</PROP>
```

By applying `UniqueWordFilter`, you can reduce this to:

```
<PROP NAME="sku.size">
  <PVAL>medium large x xx</PVAL>
</PROP>
```

Note that `UniqueWordFilter` converts all Strings to lowercase, so that redundant words are eliminated even if they do not have identical case.

You can specify `UniqueWordFilter` in the XML definition file like this:

```
<property name="size" filter="uniqueword"/>
```

You do not need to create a Nucleus component to use this filter.

Although `UniqueWordFilter` removes redundancies and concatenates values, it is not equivalent to using a combination of `UniqueFilter` and `ConcatFilter`. `UniqueFilter` considers the entire string when it eliminates redundant values, not individual words. In this example, each complete string is unique, so `UniqueFilter` would not actually eliminate any values, and the result would be:

```
<PROP NAME="sku.size">
  <PVAL>medium large x large xx large</PVAL>
```

```
</PROP>
```

Note: You should use `UniqueWordFilter` carefully, as under certain circumstances it can have undesirable effects. If you use a dictionary that includes multi-word terms, searches for those terms may not return the expected results, because the filter may rearrange the order of the words in the index.

HtmlFilter

The `atg.repository.search.indexing.filter.HtmlFilter` class removes any HTML markup from a property value. This is useful, for example, if text properties include tags for bolding or italicizing certain words, as in this `longDescription` property of a product:

```
You'll <b>love</b> this Italian <i>leather</i> sofa!
```

Because the HTML markup is included in the index, searches may return unexpected results. In this example, searching for "leather sofa" might not return the product, because that string does not actually appear in the `longDescription` property.

Using `HtmlFilter`, this value appears in the index as:

```
<PROP NAME="product.longDescription">
  <PVAL>You'll love this Italian leather sofa!</PVAL>
</PROP>
```

Now a search for "leather sofa" will find the value in this property and return this product.

7 Indexing the Content Management Repository

In addition to the products, SKUs, and other items stored and managed through the product catalog repository, the Oracle Commerce Platform includes support for storing and managing HTML articles and digital media through the Web Content Management (WCM) feature. These items are maintained in the content management repository, as described in the *Core Commerce Programming Guide*.

This chapter describes Oracle Commerce Platform components that you can use to index the content in this repository so it can be used for searching and guided navigation. It includes the following sections:

For information about creating and editing articles and media items, see the *Merchandising Guide for Business Users*.

Overview of Indexing Web Content

To provide robust support for Web content, the Oracle Commerce Platform includes a content management repository for storing HTML articles and digital media. This repository, `/atg/content/ContentManagementRepository`, has two main item types:

- `article` -- Intended for HTML documents. Text elements are stored in properties of the item (such as the `body`, `headline`, and `abstract` properties) and can be used for searching and guided navigation.
- `mediaContent` -- Intended for binary content, including video, audio, image, and PDF files. The item has a `url` property that contains a URL that points to an external binary file, and string properties such as `title` and `description` that can be used for searching and guided navigation.

The repository and its item types are defined in the top-level `ContentMgmt` module of the Oracle Commerce Platform. This module has a `ContentMgmt.Endeca.Index` submodule that configures the `EndecaIndexingOutputConfig` components and related components for indexing `article` and `mediaContent` items.

WCM EndecaIndexingOutputConfig Components

The `ContentMgmt.Endeca.Index` module contains two components of class `atg.endeca.index.EndecaIndexingOutputConfig`:

- The `/atg/content/search/ArticleOutputConfig` component specifies how to create data records that represent `article` items in the content management repository.
- The `/atg/content/search/MediaContentOutputConfig` component specifies how to create data records that represent `mediaContent` items in the content management repository.

This section describes the default configuration of these components. For more information about `EndecaIndexingOutputConfig` components, see the *Overview of Indexing* (page 17) and *Configuring the Indexing Components* (page 27) chapters.

indexingApplicationConfiguration

The component of class `atg.endeca.index.configuration.IndexingApplicationConfiguration` used to configure indexing settings for the integration. For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, the default setting is:

```
indexingApplicationConfiguration=\
  /atg/endeca/index/IndexingApplicationConfiguration
```

definitionFile

The full Nucleus pathname of the XML indexing definition file that specifies the repository item types and properties to include in the Guided Search records. For the `ArticleOutputConfig` component, this property is set as follows:

```
definitionFile=/atg/content/endeca/index/article-output-config.xml
```

This file specifies the properties of the `article` item type to include in the index. The `article.headline`, `article.abstract`, `article.author`, `article.body`, and `article.tag` output properties are specified as text searchable. The `article.siteId`, `article.author`, and `article.tag` properties are specified as dimensions.

For the `MediaContentOutputConfig` component:

```
definitionFile=/atg/content/endeca/index/mediaContent-output-config.xml
```

This file specifies the properties of the `mediaContent` item type to include in the index. The `mediaContent.title`, `mediaContent.description`, `mediaContent.mediaType`, and `mediaContent.tag` output properties are specified as text searchable. The `mediaContent.siteId`, `mediaContent.mediaType`, and `mediaContent.tag` properties are specified as dimensions.

Note that the `MediaContentOutputConfig` definition file sets the output name of the `mediaContent.$url` property (which holds the URL of the repository item) to `mediaContent._url`, to override the default output name of `mediaContent.url`. This is done to avoid a naming conflict with the `url` property of the `mediaContent` item type, which has a `url` property that holds the URL of the binary media file the item

represents. So in output records, `mediaContent.url` is used for the URL of the binary media file, while `mediaContent._url` is used for the URL of the `mediaContent` repository item. For example:

```
<PROP NAME="mediaContent._url">
  <PVAL>
    atgrep:/ContentManagementRepository/mediaContent/m011?locale=en_US
  </PVAL>
</PROP>
<PROP NAME="mediaContent.url">
  <PVAL>
    /crsdocroot/content/images/articles/banner/marathon_mania.jpg
  </PVAL>
</PROP>
```

repository

The full Nucleus pathname of the repository that the definition file applies to. For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, this property is set to the content management repository:

```
repository=/atg/content/ContentManagementRepository
```

In an ATG Content Administration environment, the repository should be set to the corresponding unversioned target repository:

```
repository=/atg/content/ContentManagementRepository_production
```

documentSubmitter

The component (typically of class `atg.endeca.index.RecordStoreDocumentSubmitter`) to use to submit records to the appropriate CAS record store. For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, this property is set as follows:

```
documentSubmitter=/atg/endeca/index/DataDocumentSubmitter
```

See Document Submitter Components (page 37) for more information.

forceToBaselineOnChange

If `true`, a baseline update is performed when a partial update is requested, if a value of a hierarchical dimension has been changed. For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, this property is set to `false` by default, because neither component generates hierarchical dimension values.

bulkLoader

A Nucleus component of class `atg.endeca.index.RecordStoreBulkLoaderImpl`. For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, this property is set to `/atg/search/repository/BulkLoader`. This is the same bulk loader used by the `ProductCatalogOutputConfig` and `CategoryToDimensionOutputConfig` components.

See Data Loader Components (page 33) for more information.

enableIncrementalLoading

If `true`, incremental loading is enabled. This property is set to `true` for both the `ArticleOutputConfig` and `MediaContentOutputConfig` components.

incrementalLoader

A Nucleus component of class `atg.endeca.index.RecordStoreIncrementalLoaderImpl`. For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, this property is set to `/atg/search/repository/IncrementalLoader`. This is the same incremental loader used for the `ProductCatalogOutputConfig` and `CategoryToDimensionOutputConfig` components.

See Data Loader Components (page 33) for more information.

siteIDsToIndex

A list of site IDs of the sites to include in the index. For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, this property is null by default, which means the `sitesToIndex` property (which is the actual property used to determine which sites to index) is automatically set to all enabled sites. Set `siteIDsToIndex` only if you want to restrict indexing to only a specific subset of the enabled sites.

replaceWithTypePrefixes

A list of the property-name prefixes that should be replaced with the item type the property is associated with. If this property is null (the default setting for both the `ArticleOutputConfig` and `MediaContentOutputConfig` components), the type prefix is added to the names of the output properties of the top-level item (`article` for the `ArticleOutputConfig` component, `mediaContent` for the `MediaContentOutputConfig` component). See the EndecaIndexingOutputConfig Components (page 28) of the *Configuring the Indexing Components* (page 27) chapter for more information about setting this property.

prefixReplacementMap

A mapping of property-name prefixes to their replacements. This mapping is applied after any type prefixes are added by `replaceWithTypePrefixes`.

For both the `ArticleOutputConfig` and `MediaContentOutputConfig` components, this property is null by default, which means no prefix replacement is performed.

suffixReplacementMap

A mapping of property-name suffixes to their replacements. If this property is null (the default setting for both the `ArticleOutputConfig` and `MediaContentOutputConfig` components), these automatic mappings are used:

```
$repositoryId=repositoryId,
$siteId=siteId,
$url=url,
$baseUrl=baseUrl
```

These mappings remove the dollar-sign ($) character from the names of special repository properties, because this character is not valid in Guided Search property names.

You can exclude the automatic mappings by setting the `addDefaultOutputNameReplacements` property to `false`.

WCM Dimension Exporter Components

The `ContentMgmt.Endeca.Index` module contains two components of class `atg.endeca.index.dimension.RepositoryTypeHierarchyExporter`:

- The `/atg/content/endeca/index/ArticleDimensionExporter` component outputs dimension value records for the `article` item type and related item types.
- The `/atg/content/endeca/index/MediaContentDimensionExporter` component outputs dimension value records for the `mediaContent` item type and related item types.

These dimension values are added to the `record.type` dimension, which represents the hierarchy of repository item types.

This section describes the default configuration of these components. For more information about `RepositoryTypeHierarchyExporter` components, see the *Overview of Indexing* (page 17) and *Configuring the Indexing Components* (page 27) chapters.

dimensionName

The name to give the dimension created from the hierarchy of repository item types. For both the `ArticleDimensionExporter` and `MediaContentDimensionExporter` components, this property is set by linking to the `recordTypeName` property of the`/atg/endeca/ApplicationConfiguration` component:

```
dimensionName^=/atg/endeca/ApplicationConfiguration.recordTypeName
```

If you want to change the value of the `dimensionName` property, you should do so by changing the value of `ApplicationConfiguration.recordTypeName` to ensure that other properties that link to it are changed as well.

indexingOutputConfig

The component of class `atg.endeca.index.EndecaIndexingOutputConfig` whose definition file should be used for generating dimension value records from the repository item-type hierarchy. For the `ArticleDimensionExporter` component, this property is set by default to:

```
indexingOutputConfig=/atg/content/search/ArticleOutputConfig
```

For the `MediaContentDimensionExporter` component, this property is set by default to:

```
indexingOutputConfig=/atg/content/search/MediaContentOutputConfig
```

documentSubmitter

The component (typically of class `atg.endeca.index.RecordStoreDocumentSubmitter`) to use to submit records to the CAS dimension values record store. (See Document Submitter Components (page 37) for more information.) For both the `ArticleDimensionExporter` and `MediaContentDimensionExporter` components, this property is set by default to:

```
documentSubmitter=/atg/endeca/index/DimensionDocumentSubmitter
```

WCM Schema Exporter Components

The `ContentMgmt.Endeca.Index` module contains two components of class `atg.endeca.index.schema.SchemaExporter`:

- The `/atg/content/endeca/index/ArticleSchemaExporter` component generates schema configuration for each property of the `article` item type specified in the `ArticleOutputConfig` definition file.
- The `/atg/content/endeca/index/MediaContentSchemaExporter` component generates schema configuration for each property of the `mediaContent` item type specified in the `MediaContentOutputConfig` definition file.

The schema configuration generated for each repository item-type property specifies whether it should be treated as a property or a dimension by Guided Search, whether it should be searchable, and the data type of the property or dimension.

This section describes the default configuration of these components. For more information about `SchemaExporter` components, see the *Overview of Indexing* (page 17) and *Configuring the Indexing Components* (page 27) chapters.

This section describes the default configuration of these components.

indexingOutputConfig

The component of class `atg.endeca.index.EndecaIndexingOutputConfig` whose definition file should be used for generating schema records. For the `ArticleSchemaExporter` component, this property is set by default to:

```
indexingOutputConfig=/atg/content/search/ArticleOutputConfig
```

For the `MediaContentSchemaExporter` component, this property is set by default to:

```
indexingOutputConfig=/atg/content/search/MediaContentOutputConfig
```

documentSubmitter

The component (typically of class `atg.endeca.index.ConfigImportDocumentSubmitter`) to use to submit schema data to the Endeca Configuration Repository. (See Document Submitter Components (page 37) for more information.) For both the `ArticleSchemaExporter` and `MediaSchemaDimensionExporter` components, this property is set by default to:

```
documentSubmitter=/atg/endeca/index/ConfigImportDocumentSubmitter
```

dimensionNameProviders

An array of components of a class that implements the `atg.endeca.index.schema.DimensionNameProvider` interface. `SchemaExporter` uses these components to create references from attribute names to dimension names.

For the `ArticleSchemaExporter` component, `dimensionNameProviders` is set to:

```
dimensionNameProviders+=ArticleDimensionExporter
```

For the `MediaContentSchemaExporter` component, `dimensionNameProviders` is set to:

```
dimensionNameProviders+=MediaContentDimensionExporter
```

WCM SimpleIndexingAdmin Component

The `ContentMgmt.Endeca.Index` module includes a `/atg/content/endeca/index/ContentMgmtSimpleIndexingAdmin` component (of class `atg.endeca.index.admin.SimpleIndexingAdmin`) for managing the process of indexing data from the content management repository. This component is similar to the `/atg/commerce/endeca/index/ProductCatalogSimpleIndexingAdmin` component, except that it is configured by default to index `article` and `mediaContent` items rather than items in the product catalog repository. In addition, the `ContentMgmtSimpleIndexingAdmin` is configured to use a different `EndecaScriptService` component (`/atg/content/endeca/index/EndecaScriptService`) to invoke EAC scripts, but this component's configuration is identical to that of the `/atg/commerce/endeca/index/EndecaScriptService` component described in the EndecaScriptService (page 40) section of the *Configuring the Indexing Components* (page 27) chapter.

If you prefer, you can configure a single `SimpleIndexingAdmin` component to manage indexing of both repositories. Oracle Commerce Reference Store uses this approach, reconfiguring the `ProductCatalogSimpleIndexingAdmin` component to invoke the `Indexable` components associated with both repositories.

This section describes the default configuration of `ContentMgmtSimpleIndexingAdmin`. For more information about these properties, see the ProductCatalogSimpleIndexingAdmin (page 41) section of the *Configuring the Indexing Components* (page 27) chapter.

phaseToPrioritiesAndTasks

This property defines the phases and tasks of an indexing job, and the order in which the phases are executed. By default, this is set to:

```
phaseToPrioritiesAndTasks=\
  RepositoryExport=10:\
    ArticleSchemaExporter;\
    ArticleDimensionExporter;\
    /atg/content/search/ArticleOutputConfig;\
    MediaContentSchemaExporter;\
    MediaContentDimensionExporter;\
    /atg/content/search/MediaContentOutputConfig,\
  EndecaIndexing=15:EndecaScriptService
```

runTasksWithinPhaseInParallel

A boolean that controls whether to run tasks within a phase in parallel. Set to `true` by default. If set to `false`, the tasks are executed in sequence, in the order specified in the `phaseToPrioritiesAndTasks` property.

enableScheduledIndexing

A boolean that controls whether to invoke indexing automatically on a specified schedule. Set to `false` by default.

baselineSchedule

A String that specifies the schedule for performing baseline updates. Set to null by default. If you set `enableScheduledIndexing` to `true`, set `baselineSchedule` to a String that conforms to one of the formats accepted by classes implementing the `atg.service.scheduler.Schedule` interface.

partialSchedule

A String that specifies the schedule for performing partial updates. The format for the String is the same as the format used for `baselineSchedule`. Set to null by default.

retryInMs

The amount of time (in milliseconds) to wait before retrying a scheduled indexing job if the first attempt to execute it fails. Set by default to -1, which means no retry. If you change this value, you should set it to a relatively short amount of time to ensure that the indexing job completes before the next scheduled job begins. If `ContentMgmtSimpleIndexingAdmin` estimates that the retried job will not complete before the next scheduled job, it skips the retry.

jobQueue

Specifies the component that manages queueing of index jobs. Set by default to `/atg/endeca/index/InMemoryJobQueue`, which is the same component used by `ProductCatalogSimpleIndexingAdmin`.

8 Indexing Dynamic Item Types and Properties

Creating new item descriptors and properties in a repository typically involves modifying the repository's database schema and XML definition file, and then restarting your application to make the changes available. If you want to avoid restarting, however, the Oracle Commerce Platform provides an alternate mechanism for dynamically creating subtypes of existing item types and adding properties to static and dynamic item types. This mechanism involves creating metadata items for the subtypes and properties through a Content Administration project and deploying these metadata items. The system then generates the new subtypes and properties from the metadata, and adds them to the repository definition automatically. Data for these properties is stored in database tables that are included specifically for this purpose in the default schema.

The Oracle Commerce Platform can create Endeca records from these items and properties and submit them to Oracle Commerce Guided Search for indexing. You can add dynamic item types and properties to the definition files of your `EndecaIndexingOutputConfig` components in the same way that you add static item types and properties.

However, adding these item types and properties in this way requires restarting the Oracle Commerce Platform to make the `EndecaIndexingOutputConfig` definition file changes available. To avoid restarting, you can use an alternate approach to specify dynamic item types and properties for indexing without modifying `EndecaIndexingOutputConfig` definition files. Instead, you provide the necessary specifications when you create the metadata items that the dynamic types and properties are generated from.

This chapter describes how to specify indexing information for dynamic item types and properties without requiring a restart. It includes the following sections:

Note that this chapter assumes you are already familiar with dynamic item types and properties. See the *Creating Dynamic Item Types and Properties* chapter of the *Content Administration Programming Guide*.

Updating the Indexing Components

To enable specifying dynamic properties for indexing through attributes of the associated metadata items, you must first change the configuration of some of the indexing components. The Oracle Commerce Platform includes a subclass of the `EndecaIndexingOutputConfig` class, `atg.endeca.index.DynamicEndecaIndexingOutputConfig`. For repositories that support dynamic item

types or properties, change the class of the corresponding `EndecaIndexingOutputConfig` components to the `DynamicEndecaIndexingOutputConfig` class:

```
$class=atg.endeca.index.DynamicEndecaIndexingOutputConfig
```

The Oracle Commerce Platform also includes a subclass of the `SchemaExporter` class, `atg.endeca.index.schema.DynamicSchemaExporter`. For each `EndecaIndexingOutputConfig` component whose class is set to `DynamicEndecaIndexingOutputConfig`, set the class of the corresponding `SchemaExporter` component to `DynamicSchemaExporter`:

```
$class=atg.endeca.index.schema.DynamicSchemaExporter
```

In addition, for repositories that support dynamic item types or properties, you should set the `forceToBaselineOnChange` property to `true` on associated components of the `RepositoryTypeHierarchyExporter` and `DynamicSchemaExporter` classes. These settings ensure that a baseline index is performed when you add or modify dynamic item types or properties. New dynamic item types and properties do not appear in the MDEX until a baseline index has been performed.

Specifying Dynamic Items and Properties for Indexing

If a static item type is included in an `EndecaIndexingOutputConfig` definition file, then any dynamic item types that are descendants of that item type are automatically available for indexing as well. For example, if you create an `electricalProduct` subtype of the `product` item type (which is included in the `ProductCatalogOutputConfig` definition file), `electricalProduct` items will be indexed, as will any subtypes of the `electricalProduct` item type.

To specify that a dynamic property should be included in the index (either a dynamic property of a static item type or a subtype-specific property of a dynamic subtype), you set the `searchable` attribute of the property to `true`. In addition, you need to set at least one of the following attributes to `true` to specify how the property is handled in the MDEX:

- `textSearchable` – Setting to `true` specifies that the values of the property should be treated as searchable text. Equivalent to the `text-searchable` attribute in `EndecaIndexingOutputConfig` definition files.
- `wildcardSearchable` – Setting to `true` specifies that the values of the property should be treated as searchable text and support the use of the asterisk (`*`) as a wildcard in search terms. Equivalent to the `wildcard-searchable` attribute in `EndecaIndexingOutputConfig` definition files.
- `dimension` -- Setting to `true` specifies that the property should be treated as a dimension. Equivalent to the `is-dimension` attribute in `EndecaIndexingOutputConfig` definition files.

To set these attributes, you create `das_gsa_dynamic_attr` metadata items that are associated with the dynamic property when it is generated. For example, the following XML import file creates a `das_gsa_dynamic_prop` metadata item for a dynamic property, and creates `das_gsa_dynamic_attr` items that set attributes to specify that the property should be included in the MDEX:

```
<add-item item-descriptor="das_gsa_dynamic_prop" id="wattage"
```

```
  repository="/atg/repository/dynamic/DynamicMetadataRepository"
  no-checkin="false">
  <set-property name="property_name"><![CDATA[wattage]]></set-property>
  <set-property name="item_descriptor"><![CDATA[electricalProduct]]>
    </set-property>
  <set-property name="data_type"><![CDATA[float]]></set-property>
  <set-property name="repository">
    <![CDATA[/atg/commerce/catalog/ProductCatalog]]></set-property>
</add-item>

<add-item item-descriptor="das_gsa_dynamic_attr" id="wattageAttr1"
   no-checkin="false">
   <set-property name="attribute_name"><![CDATA[writable]]></set-property>
   <set-property name="item_descriptor"><![CDATA[electricalProduct]]>
     </set-property>
   <set-property name="property_name"><![CDATA[wattage]]></set-property>
   <set-property name="repository">
     <![CDATA[/atg/commerce/catalog/ProductCatalog]]></set-property>
   <set-property name="is_dynamic_property"><![CDATA[true]]></set-property>
   <set-property name="data_type"><![CDATA[string]]></set-property>
   <set-property name="value"><![CDATA[true]]></set-property>
</add-item>

<add-item item-descriptor="das_gsa_dynamic_attr" id="wattageAttr2"
   no-checkin="false">
   <set-property name="attribute_name"><![CDATA[searchable]]></set-property>
   <set-property name="item_descriptor"><![CDATA[electricalProduct]]>
     </set-property>
   <set-property name="property_name"><![CDATA[wattage]]></set-property>
   <set-property name="repository">
     <![CDATA[/atg/commerce/catalog/ProductCatalog]]></set-property>
   <set-property name="is_dynamic_property"><![CDATA[true]]></set-property>
   <set-property name="data_type"><![CDATA[string]]></set-property>
   <set-property name="value"><![CDATA[true]]></set-property>
</add-item>

<add-item item-descriptor="das_gsa_dynamic_attr" id="wattageAttr3"
   no-checkin="false">
   <set-property name="attribute_name"><![CDATA[dimension]]></set-property>
   <set-property name="item_descriptor"><![CDATA[electricalProduct]]>
     </set-property>
   <set-property name="property_name"><![CDATA[wattage]]></set-property>
   <set-property name="repository">
     <![CDATA[/atg/commerce/catalog/ProductCatalog]]></set-property>
   <set-property name="is_dynamic_property"><![CDATA[true]]></set-property>
   <set-property name="data_type"><![CDATA[string]]></set-property>
   <set-property name="value"><![CDATA[true]]></set-property>
</add-item>
```

Indexing settings that are specified through metadata item attributes override equivalent settings specified in an `EndecaIndexingOutputConfig` definition file. In the example above, if the dynamic property is specified for indexing in the definition file and its `is-dimension` attribute is set to `false`, this value is overridden by the `dimension` attribute setting in the `das_gsa_dynamic_attr` metadata item. Similarly, if you want to disable indexing of a property specified in an `EndecaIndexingOutputConfig` definition file, you can do this by creating a `das_gsa_dynamic_attr` metadata item that sets the `searchable` attribute to `false`.

Note that the available options for configuring indexing settings through repository item attributes are limited. Equivalent attributes exist only for a subset of the indexing settings that can be configured through `EndecaIndexingOutputConfig` definition files.

Specifying the Output Property Name

By default, the output name of a dynamic property in generated records is:

```
item-type.property-name
```

For example, a `weight` dynamic property of the `sku` static item type would appear in the MDEX as `sku.weight`. You can override the default output property name by setting the optional `outputName` attribute for the dynamic property. For example, for a property to appear as `sku.weightInGrams`:

```
<add-item item-descriptor="das_gsa_dynamic_attr" id="weightAttr1"
   no-checkin="false">
   <set-property name="attribute_name"><![CDATA[outputName]]></set-property>
   <set-property name="item_descriptor"><![CDATA[sku]]>
     </set-property>
   <set-property name="property_name"><![CDATA[weight]]></set-property>
   <set-property name="repository">
     <![CDATA[/atg/commerce/catalog/ProductCatalog]]></set-property>
   <set-property name="is_dynamic_property"><![CDATA[true]]></set-property>
   <set-property name="data_type"><![CDATA[string]]></set-property>
   <set-property name="value"><![CDATA[sku.weightInGrams]]></set-property>
</add-item>
```

The `outputName` attribute is equivalent to the `output-name` attribute in `EndecaIndexingOutputConfig` definition files.

For dynamic properties of dynamic subtypes, be sure to use the `outputName` attribute to specify the output name, even if you are not overriding the default value. Doing this ensures that the correct *`item-type`* prefix is included. For example, if you have an `electricalProduct` subtype of the `product` item type, and you add a `wattage` property to `electricalProduct`, you could specify the output name as follows:

```
<add-item item-descriptor="das_gsa_dynamic_attr" id="wattageAttr4"
   no-checkin="false">
   <set-property name="attribute_name"><![CDATA[outputName]]></set-property>
   <set-property name="item_descriptor"><![CDATA[electricalProduct]]>
     </set-property>
   <set-property name="property_name"><![CDATA[wattage]]></set-property>
   <set-property name="repository">
     <![CDATA[/atg/commerce/catalog/ProductCatalog]]></set-property>
   <set-property name="is_dynamic_property"><![CDATA[true]]></set-property>
   <set-property name="data_type"><![CDATA[string]]></set-property>
   <set-property name="value"><![CDATA[electricalProduct.wattage]]></set-property>
</add-item>
```

Adding Properties to a Search Interface

In addition to marking dynamic properties as searchable as described above, you must also add them to a search interface in Oracle Commerce Guided Search. See the *Oracle Commerce Guided Search MDEX Engine Developer's Guide* for information about search interfaces.

9 Query Integration

The Oracle Commerce Core Platform provides two options when querying for content served by the Oracle Commerce Assembler:

- Invoking the Assembler via a servlet as part of the Core Platform's request handling pipeline. This option allows the call to the Assembler to happen early in the page's life cycle, which is desirable when the bulk of the page's content is served by the Assembler.
- Invoking the Assembler from within a page, using a servlet bean. This option allows the call to the Assembler to occur on a just-in-time basis for the portion of the page that requires Assembler-served content. This approach is desirable when only a small portion of the page requires Assembler content.

The remainder of this chapter provides more detail on both configurations and the components that facilitate them. It includes these sections:

Content Item Classes

Similar to HTTP requests, requests that are made to the Assembler use the paradigm of a request object and a response object. Both of these objects are of type

com.endeca.infront.assembler.ContentItem. There are two subclasses of ContentItem, depending on the type of content being requested: com.endeca.infront.cartridge.ContentInclude and com.endeca.infront.cartridge.ContentSlotConfig.

ContentInclude is used to request pages defined in the Site Pages section of Experience Manager. Invoking the Assembler for a page request is also referred to as "invoking the Assembler with a ContentInclude." The handler for the ContentInclude component first tries to retrieve the content at the exact URI specified in the ContentInclude. If there is no content at that location, the handler attempts to find the deepest matching path. For example, assume a browse page exists in the Experience Manager Site Pages definitions for SiteA. Passing in a /browse path for SiteA will match this browse page. Passing in a /browse/seo/url path will also match this page because the deepest matching path that the handler can find for /browse/seo/url is /browse (this example assumes that a browse/seo/url page does not exist in Experience Manager).

ContentSlotConfig is used to request content from a content folder that has been defined in the Content section of Experience Manager. Invoking the Assembler for a content folder request is also referred to as "invoking the Assembler with a ContentSlot item." A content folder request must specify the name of the content folder and the number of items to retrieve from that folder. The handler for ContentSlotConfig uses these parameters to form a content trigger request that fetches the top item (or items) from the folder by priority. The Assembler then processes the content items from the folder and returns them as part of the response for rendering.

A third class, com.endeca.infront.cartridge.RedirectAwareContentInclude, also exists. RedirectAwareContentInclude is a subclass of the ContentInclude class and it supports requests for configurations that use keyword redirects. The remainder of this chapter makes a distinction between ContentInclude, ContentSlotConfig, and RedirectAwareContentInclude classes when necessary. When the distinction is not required, the more general ContentItem is used.

Note: For more information on the ContentInclude, ContentSlotConfig, and RedirectAwareContentInclude classes and their associated handler classes, refer to the *Oracle Commerce Guided Search Assembler Application Developer's Guide*.

Invoking the Assembler in the Request Handling Pipeline

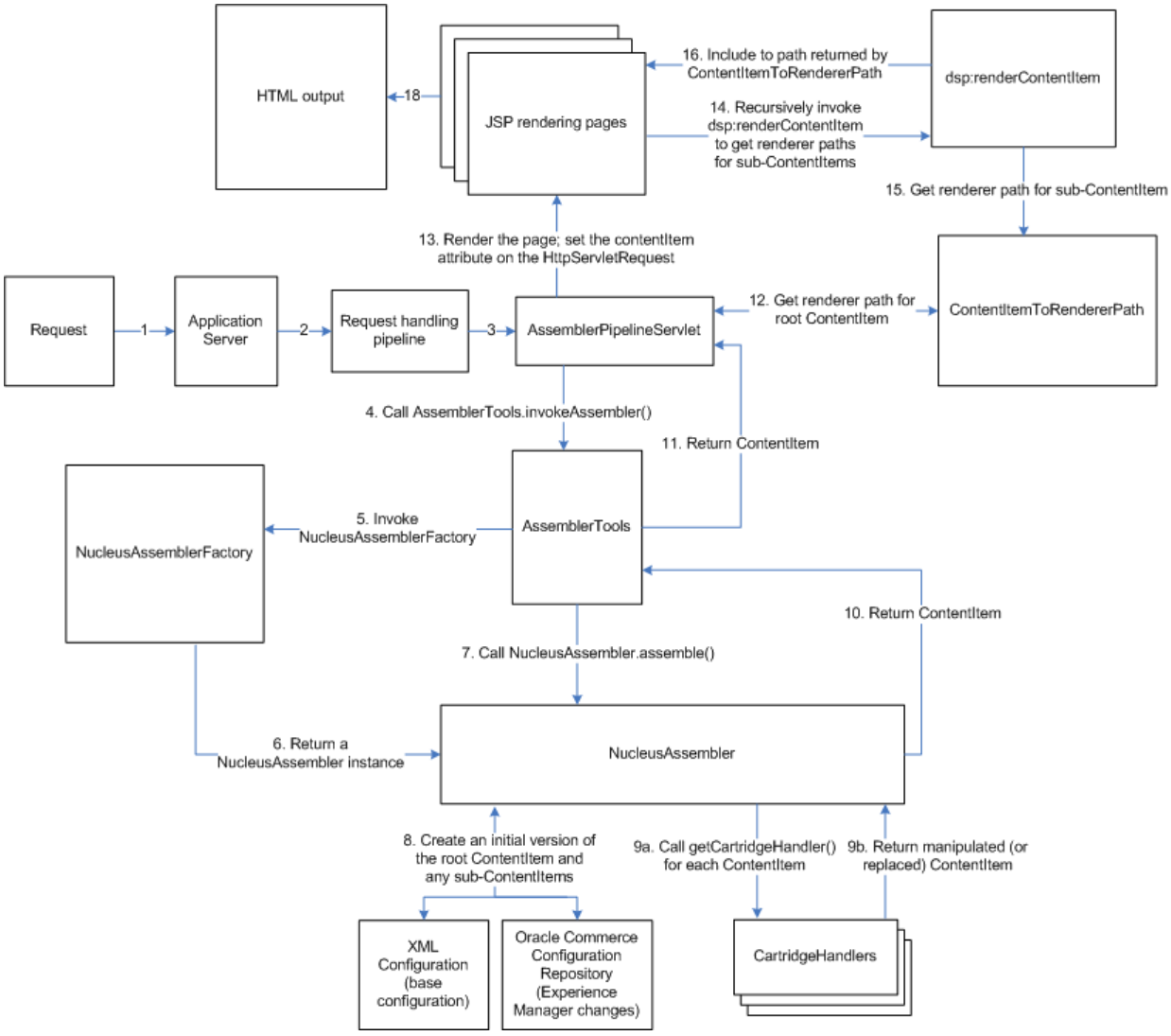
In this option, the Assembler is invoked early in the page rendering process as part of the Oracle Commerce Core Platform request handling pipeline. This option is appropriate when the bulk of a page's content is served by the Assembler. This guide refers to these pages as "Assembler-driven pages."

Assembler-driven pages are generally those pages that benefit greatly from increased merchandiser control. For example, a home page is a good candidate to be Assembler-driven because merchandisers want to customize their site's home page based on the season, a current sale, or a customer's profile. A search results page is also a good candidate because merchandisers may want to control the order of search results, specify special brand landing pages for particular searches, and so on. The Oracle Commerce Experience Manager tool, which works in conjunction with the Assembler API, is designed to facilitate increased merchandiser control, therefore pages that need a high level of merchandiser control are best served through the Assembler API/Experience Manager combination.

The content that the Assembler returns to the client browser can take several forms: JSP, XML, or JSON, as described in the following sections.

Using a JSP Renderer to Render Content

The request-handling architecture for an Assembler-driven JSP page looks like this:



In this diagram, the following happens:

1. The application server receives a request.
2. The application server passes the request to the Oracle Commerce Core Platform request handling pipeline.
3. The request handling pipeline does some preliminary work, such as setting up the profile and determining which Oracle Commerce Platform site the request is for. At the appropriate point, the pipeline invokes the `/atg/endeca/assembler/AssemblerPipelineServlet`.
4. The `AssemblerPipelineServlet` determines if the request is for a page or a content folder in Experience Manager and creates either a `RedirectAwareContentInclude` object (for a page) or a `ContentSlotConfig` object (for a content folder). Then, `AssemblerPipelineServlet` calls the `invokeAssembler()` method on the `/atg/endeca/assembler/AssemblerTools` component and passes it the request object it created.

5. The AssemblerTools component invokes the createAssembler() method on the /atg/endeca/assembler/NucleusAssemblerFactory component.

6. The NucleusAssemblerFactory component returns an atg.endeca.assembler.NucleusAssembler instance.

7. The AssemblerTools component invokes the assemble() method on the NucleusAssembler instance and passes it the request object. The handler for the request object (which may be the RedirectAwareContentIncludeHandler or ContentSlotConfigHandler, depending on the type of request object passed in) resolves a connection to the Workbench and/or the MDEX. For page requests, the handler also invokes a series of other components that transform the request URL into a URI that contains the path to the page in Experience Manager.

8. The NucleusAssembler instance assembles the content for the request URI. Content, in this case, corresponds to a hierarchical set of cartridges and their associated data. For each cartridge, the content starts with any default data that was specified in the Experience Manager cartridge configuration files when the cartridge was added to the page. That data is further modified and augmented with any data stored in the Oracle Commerce Configuration Repository (that is, changes made and saved via the Experience Manager UI).

9. Next, the NucleusAssembler instance calls the NucleusAssembler.getCartridgehandler() method, passing in the cartridge's ContentItem type, to retrieve the correct handler for the cartridge. The handler gets resolved and executed and the results are stored in the cartridge's associated ContentItem. This process happens recursively so that the assembled content takes the form of a response ContentItem that consists of a root ContentItem which may have sub-ContentItem objects as attributes.

   Note: If a cartridge handler does not exist for a ContentItem, the initial version of the item, created in step 8, is returned.

10. The NucleusAssembler instance returns the root ContentItem to the AssemblerTools component.

11. The AssemblerTools component returns the root ContentItem to AssemblerPipelineServlet.

12. The AssemblerPipelineServlet component calls the /atg/endeca/assembler/cartridge/renderer/ContentItemToRendererPath component to get the path to the renderer (in this case, a JSP file) for the root ContentItem. The ContentItemToRendererPath component uses pattern matching to match the ContentItem type to a JSP file; for example, in Commerce Reference Store, if the ContentItem type is Breadcrumbs, the JSP file is /cartridges/Breadcrumbs/Breadcrumbs.jsp.

    Note: See ContentItemToRendererPath (page 112) for more details on how the renderer path is calculated.

13. The AssemblerPipelineServlet component sets the assembled ContentItem as a contentItem parameter on the HttpServletRequest, then forwards the request to the JSP determined by the ContentItemToRendererPath component

14. Due to the nested nature of ContentItems, the JSP for the root ContentItem may have to render sub-ContentItems, and those sub-ContentItems may have their own sub-ContentItems as well. As such, each JSP renderer, from the root on down, must include dsp:renderContentItem tags for its immediate sub-ContentItems. This configuration creates a recursive scenario that allows all sub-ContentItems to be rendered.

15. The dsp:renderContentItem tag invokes the ContentItemToRendererPath component to retrieve the JSP renderer for the current sub-ContentItem. The retrieved JSP is then included in the rendered page.

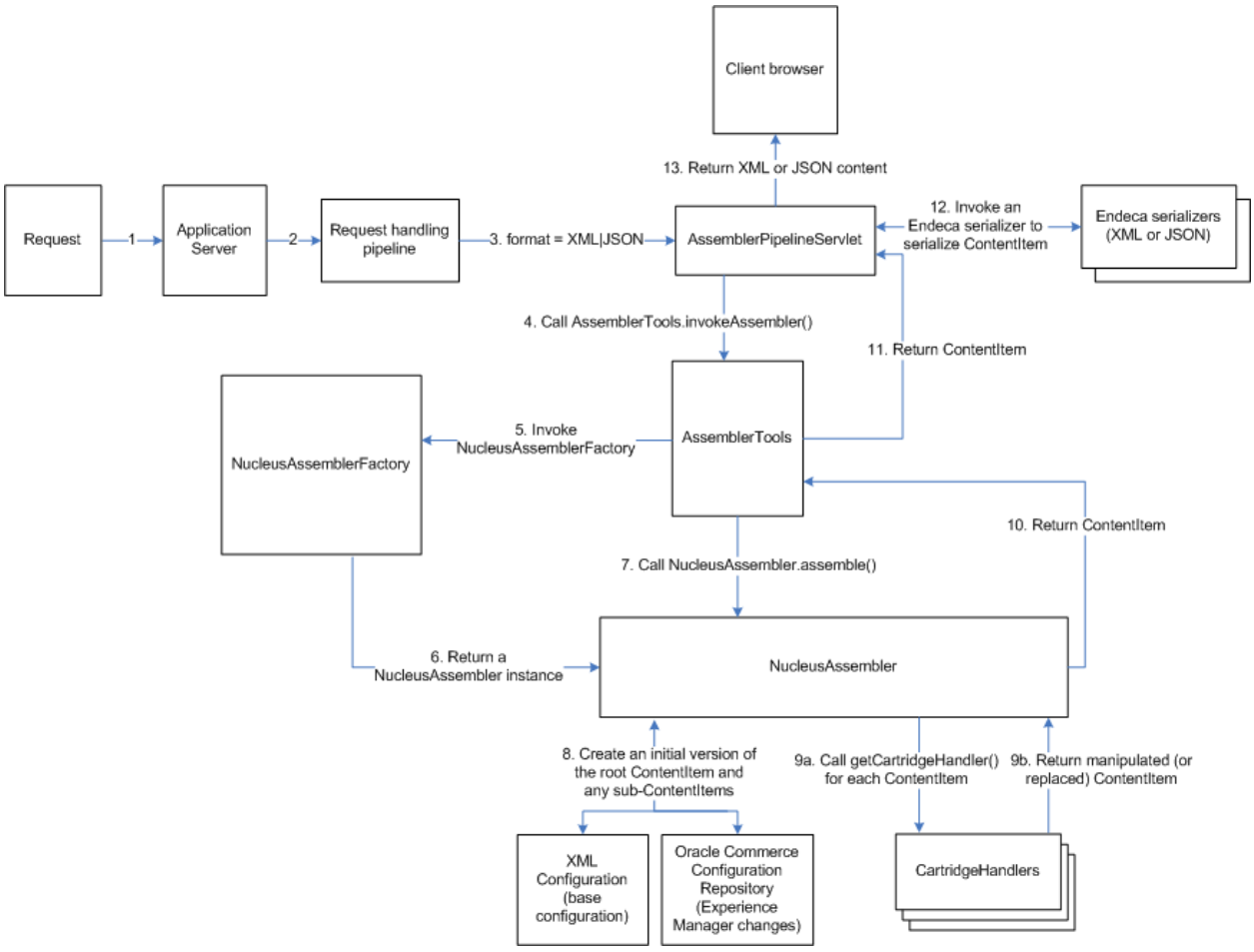
    The dsp:renderContentItem tag also sets the contentItem attribute on the HttpServletRequest, thereby making each sub-ContentItem available to its renderer; however, this value lasts only for the duration of the include so that after the include is done, the contentItem attribute's value returns to the root ContentItem.

16.The JSPs returned by the `ContentItemToRendererPath` component are included in the response.

17.The response is returned to the browser.

Rendering XML or JSON Content

The process for handling XML or JSON output is very similar to that for JSPs, with some minor modifications. The architecture diagram for an XML or JSON response looks like the following (note that this diagram is identical to the JSP diagram except for steps 13 and 14):



Serializing the content to XML or JSON is controlled by the `AssemblerPipelineServlet.formatParamName` property. This property specifies the name of the request parameter that must be passed in order to serialize the content. This property defaults to `format`, meaning that, in order to serialize output, the request must include a `format` parameter with an acceptable value. Acceptable values are `xml` and `json`. For example, the following URL returns `json` for a content folder request:

```
http://localhost:8080/assembler/assembler?assemblerContentCollection=/content/
BrowsePageCollection&format=json
```

This example returns `json` for a page request:

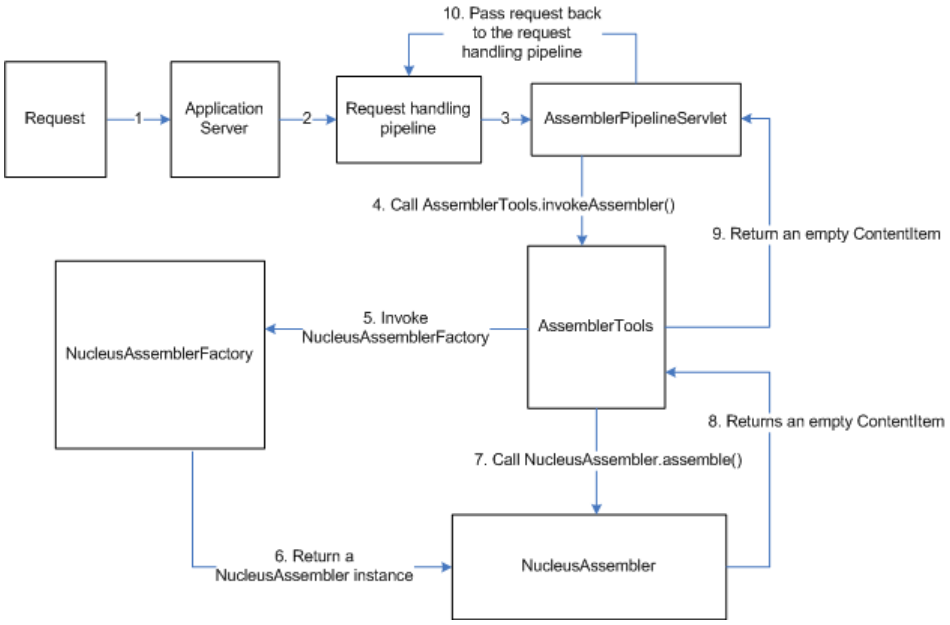
```
http://localhost:8080/assembler/browse?format=json
```

If the request specifies the `format` parameter and either `XML` or `JSON` as the value, then after the `AssemblerPipelineServlet` component receives the response `ContentItem` from `AssemblerTools`, it calls the appropriate serializer to reformat the response into XML or JSON, respectively. The `AssemblerPipelineServlet` component then returns the reformatted content to the client browser.

Setting the `AssemblerPipelineServlet.formatParamName` property to `null` disables the serializing feature and suppresses the rendering of the response entirely. This feature allows you to suppress content as needed in production environments.

When the Assembler Returns an Empty ContentItem

In the case where the `NucleusAssembler` instance returns a null response or the response `ContentItem` contains an `@error` key (in other words, the request is not an Assembler request), the `AssemblerPipelineServlet` component simply passes the request back to the Core Platform request handling pipeline for further processing. This scenario is shown in the diagram below:



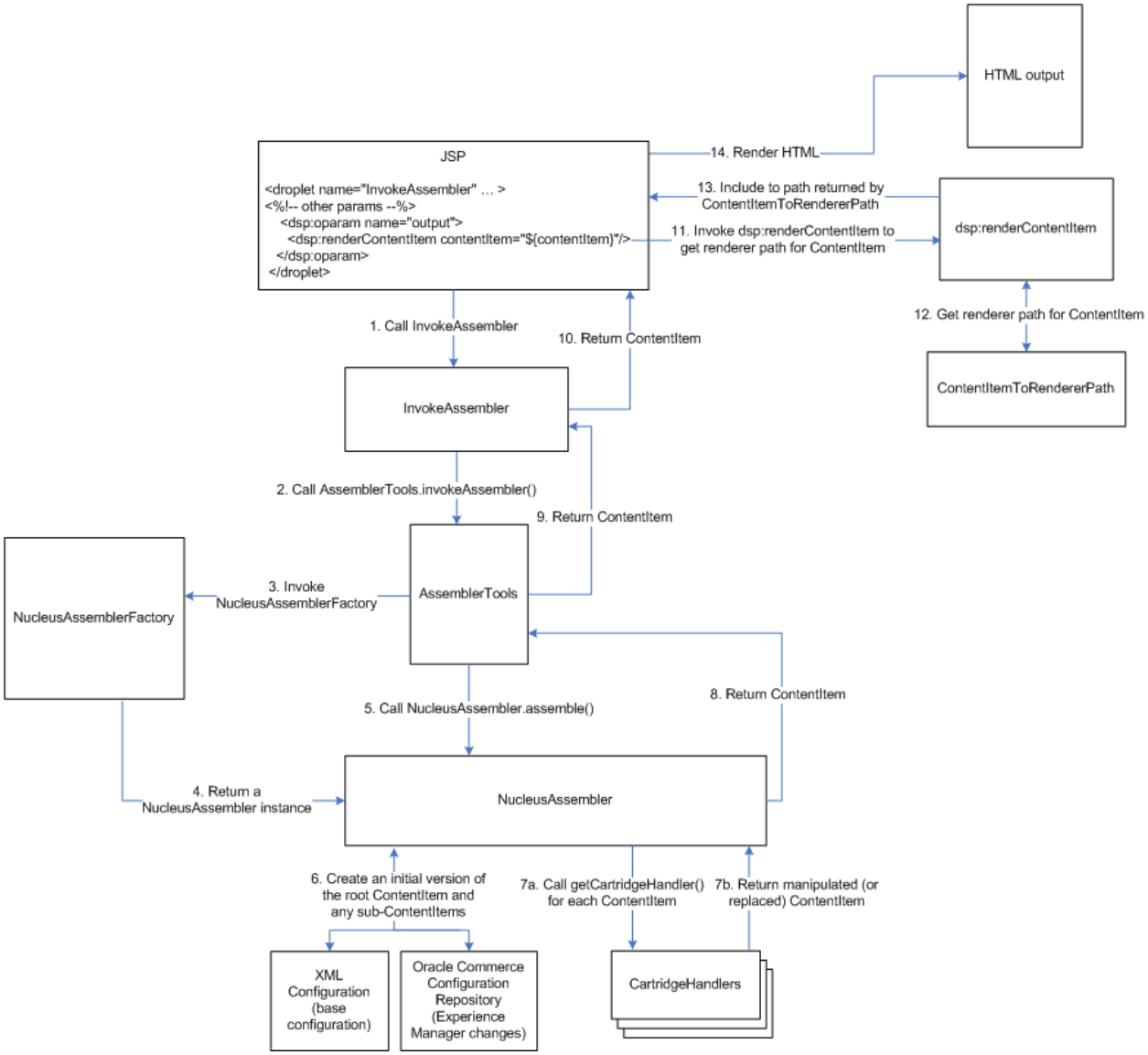
Note that you can configure an application to bypass the `AssemblerPipelineServlet` and avoid this scenario. For more information, see the AssemblerPipelineServlet (page 93) section.

Invoking the Assembler using the InvokeAssembler Servlet Bean

Invoking the Assembler from within a page, using a servlet bean, allows the call to the Assembler to occur on a just-in-time basis for the portion of the page that requires Assembler-served content. This approach is desirable

when only a small portion of the page requires Assembler content. This guide refers to these pages as "Nucleus-driven pages."

The request-handling architecture for an Nucleus-driven JSP page looks like this:



In this diagram, the following happens:

1. The JSP page code calls the `InvokeAssembler` servlet bean and passes it either the `includePath` parameter, for a page request, or the `contentCollection` parameter, for a content folder request.
2. The `InvokeAssembler` servlet bean parses the `includePath` or `contentCollection` parameter into a request object, in the form of a `RedirectAwareContentInclude` object (for a page) or a `ContentSlotConfig` object (for a content folder). Then, the `InvokeAssembler` servlet bean calls the `invokeAssembler()` method on the `/atg/endeca/assembler/AssemblerTools` component and passes it the request object it created.
3. The `AssemblerTools` component invokes the `createAssembler()` method on the `/atg/endeca/assembler/NucleusAssemblerFactory` component.

4. The `NucleusAssemblerFactory` component returns an `atg.endeca.assembler.NucleusAssembler` instance.

5. The `AssemblerTools` component invokes the `assemble()` method on the `NucleusAssembler` instance and passes it the request object. The handler for the request object (which may be the `RedirectAwareContentIncludeHandler` or `ContentSlotConfigHandler`, depending on the type of request object passed in) resolves a connection to the Workbench and/or the MDEX. For page requests, the handler also invokes a series of other components that transform the request URL into a URI that contains the path to the page in Experience Manager.

6. The `NucleusAssembler` instance assembles the correct content for the request URI. Content, in this case, corresponds to a hierarchical set of cartridges and their associated data. For each cartridge, the content starts with any default data that was specified in the Experience Manager cartridge configuration files when the cartridge was added to the page. That data is further modified and augmented with any data stored in the Oracle Commerce Configuration Repository (that is, changes made and saved via the Experience Manager UI).

7. Next, the `NucleusAssembler` instance calls the `NucleusAssembler.getCartridgehandler()` method, passing in the cartridge's `ContentItem` type, to retrieve the correct handler for the cartridge. The handler gets resolved and executed and the results are stored in the cartridge's associated `ContentItem`. This process happens recursively so that the assembled content takes the form of a response `ContentItem` that consists of a root `ContentItem` which may have sub-`ContentItem` objects as attributes.

   Note: If a cartridge handler does not exist for a `ContentItem`, the initial version of the item, created in step 6, is returned.

8. The `NucleusAssembler` instance returns the root `ContentItem` to the `AssemblerTools` component.

9. The `AssemblerTools` component returns the root `ContentItem` to the `InvokeAssembler` servlet bean.

10. When the `ContentItem` is not empty, the `InvokeAssembler` servlet bean's `output` oparam is rendered. In this example, we assume that the `output` oparam uses a `dsp:renderContentItem` tag to call the `/atg/endeca/assembler/cartridge/renderer/ContentItemToRendererPath` component to get the path to the JSP renderer for the root `ContentItem`. However, choosing when and how many times to invoke `dsp:renderContentItem` depends on what the application needs to do. It may make sense to invoke `dsp:renderContentItem` for the root `ContentItem`, and then recursively invoke `dsp:renderContentItem` for all the sub-`ContentItems` via additional `dsp:renderContentItem` tags. Alternatively, you could take a more targeted approach where you invoke `dsp:renderContentItem` for individual sub-`ContentItems` as needed.

    Note that the `dsp:renderContentItem` tag also sets the `contentItem` attribute on the `HttpServletRequest`, thereby making the `ContentItem` available to the renderers. This value lasts for the duration of the `include` only.

11. The `ContentItemToRendererPath` component returns the correct renderer for the `ContentItem`.

12. The JSP returned by `ContentItemToRendererPath` is included in the response.

13. The response is returned to the browser.

Choosing Between Pipeline Invocation and Servlet Bean Invocation

When choosing whether to use pipeline invocation or servlet bean invocation to retrieve content from the Assembler, it is useful to keep in mind the following considerations:

- The pipeline servlet operates at an HTTP request level. HTTP requests often map to entire pages in Experience Manager, making such pages good candidates for pipeline servlet invocation.
- The servlet bean is useful when only a portion of a page needs to be managed by the Experience Manager user. This type of page can use the servlet bean to request that portion's content from the Assembler.
- For performance reasons, Oracle recommends minimizing the number of servlet bean invocations on any given page.
- Cartridges that are intended to work on the same result set should all be retrieved during the same Assembler invocation, regardless of the invocation type you use. For example, the search results, breadcrumbs, and navigation cartridges should all return content that is based on the same results set.
- If your business users need the ability to create their own page URLs, for example, `/browse/WinterSale`, those pages should be managed in Experience Manager and they should be retrieved via pipeline servlet invocation to ensure that the URL is recognized as an Assembler URL and properly directed to the Assembler. Conversely, if you have pages whose URLs must not be edited, you can manage those pages as Nucleus-driven pages and provide access to any configurable content in Experience Manager through a servlet bean.

Components for Invoking the Assembler

This section provides more details on the components that invoke the Assembler.

AssemblerPipelineServlet

The `/atg/endeca/assembler/AssemblerPipelineServlet` component is part of Oracle Commerce Core Platform request handling pipeline and it is of class `atg.endeca.assembler.AssemblerPipelineServlet`. `AssemblerPipelineServlet`'s primary task is to invoke the Assembler, passing in a `ContentInclude` (for a page request) or a `ContentSlotConfig` (for a content folder request). `AssemblerPipelineServlet` is started when the Oracle Commerce Platform server is started. The `/Initial.properties` file under `DAF.Endeca.Assembler` configures this behavior by adding `AssemblerPipelineServlet` to its initial services.

```
initialServices+=\
  /atg/endeca/assembler/AssemblerPipelineServlet
```

On invocation of the `AssemblerPipelineServlet.service()` method, several items are checked to determine whether or not the servlet should execute:

- The `AssemblerPipelineServlet.enable` property: If this property is set to `false`, the servlet is disabled and the request will be passed. This property defaults to `true`.

- The `atg.assembler` context parameter: A web application must explicitly set the `atg.assembler` context parameter to true in its `web.xml` file, otherwise the `AssemblerPipelineServlet` will pass the request. To set the `atg.assembler` context parameter to `true`, add the following to the application's `web.xml` file:

  ```
  <context-param>
  <param-name>atg.assembler</param-name>
  <param-value>true</param-value>
  </context-param>
  ```

  Applications that never have a need to invoke the Assembler, should set `atg.assembler` to `false` to bypass the servlet and avoid making requests to the Assembler.

- The MIME type of the request: `AssemblerPipelineServlet` uses the request URI to determine the MIME type of the request. If `AssemblerPipelineServlet` is not allowed to process the specified MIME type, it passes the request. By default, the `AssemblerPipelineServlet` component passes all known MIME types and only executes for a null MIME type. See Bypassing or Invoking the Assembler Based On MIME Type (page 95) for more information on customizing the MIME types that the `AssemblerPipelineServlet` is allowed to execute.

- The `AssemblerPipelineServlet.ignoreRequestURIPattern` property: This optional property contains a regular expression that defines a pattern for URIs that should be disallowed. When this property is set, the request URI is compared against the specified regular expression and, if the current URI matches the regular expression, the request is passed. Out of the box, this property is not set.

If all of the above checks pass, `AssemblerPipelineServlet` executes. Its first task is to determine whether the request is a page request or a content folder request. `AssemblerPipelineServlet` makes this determination based on the URL, as described in the following sections.

Content Folder Request Identification and Handling

The URL for a content folder request has some additional requirements that the URL for a page request does not have. Specifically, the URL for a content folder must have an `/assembler` sub-path and an `assemblerContentCollection` request parameter. For example:

```
/crs/storeus/assembler/?assemblerContentCollection=Search Box Auto Suggest Content
```

The `/assembler` sub-path can take any of these forms:

- `/assembler`
- `<context-root>/assembler` (for example, `crs/assembler`)
- `<site.productionURL>/assembler` (for example, `/crs/storeus/assembler`)

The `assemblerContentCollection` request parameter must specify the name of a content folder. If these content folder URL conditions are met, `AssemblerPipelineServlet` creates a `ContentSlotConfig` object and passes it to the Assembler:

```
contentItem = new ContentSlotConfig(content, ruleLimit);
```

A content folder URL may also include the optional `assemblerRuleLimit` request parameter. This is an integer value that is used as an argument to the `ContentSlotConfig` constructor. It determines the number of items to return from the content folder. If `assemblerRuleLimit` is not set or is an invalid value, then the default value of 1 is used.

```
/crs/storeus/assembler/?assemblerContentCollection=Search Box Auto Suggest
 Content&assemblerRuleLimit=3
```

If the content folder does not exist, the Assembler returns a content item whose `contents` value is empty. For example, this URL:

```
http://localhost:8080/assembler/assembler?assemblerContentCollection=/content/
BrowsePageCollection&format=json
```

Results in this data:

```
{"@type":"ContentSlot","contents":[],"ruleLimit":1,"contentCollection":"\/content\/
BrowsePageCollection"}
```

Page Request Identification and Handling

If the URL does not fit the requirements for a content folder request, the `AssemblerPipelineServlet` component assumes that this is a page request. A page request URL must be transformed into a URI that matches one of the pages defined Experience Manager. See the Calculating the Content Path from the Page Request URL (page 97) section for details on how the URI is calculated.

Bypassing or Invoking the Assembler Based On MIME Type

By default, the `AssemblerPipelineServlet` limits its Assembler invocation to request paths that do not match a known MIME type. It does this via a reference to the `/atg/dynamo/servlet/pipeline/MimeTyper` component, which is part of the Oracle Commerce Core Platform system that routes and executes requests based on matching MIME types. This configuration prevents the `AssemblerPipelineServlet` from intercepting requests for JSP, CSS, HTML, and JavaScript files, among others.

You can add allowed MIME types or disable Assembler invocation for unknown MIME types using the following `AssemblerPipelineServlet` configurable properties:

```
# Whether to invoke the Assembler for a potential match on a request
# that doesn't match a known MIME type (typically a directory).
#
# assembleUnknownMimeTypes=true

# A String array of allowed MIME types. Defaults to null, but
# can be set to a MIME type if you want to pass certain extensions to
# the Assembler (for example, ".asm" or ".endeca").
#
# allowedMimeTypes=
```

See the *Platform Programming Guide* for more information on the `MimeTyper` component.

InvokeAssembler

The `/atg/endeca/assembler/droplet/InvokeAssembler` servlet bean, which is of class `atg.endeca.assembler.droplet.InvokeAssembler`, provides a means of invoking the Assembler via a servlet bean on a page. It is useful on pages that contain mostly Nucleus-driven content, with a section of

Assembler-based content. Note that, for pages that have multiple sections of Assembler content, you should consider combining the requests for that content into a single `InvokeAssembler` call for performance reasons.

Input Parameters

The `InvokeAssembler` servlet bean has two input parameters, `includePath` and `contentCollection`, described below. Note that you must provide either an `includePath` or a `contentCollection` parameter, but you cannot provide both.

includePath
Use the `includePath` parameter for a page request. The path you specify must correspond to the name of a page in Experience Manager and is relative to the current site. For example, if `includePath` is set to `/browse` and the current site is Site A, the content for Site A's `browse` page is retrieved. `InvokeAssembler` creates a `ContentInclude` component and sets its `contentUri` property from the `includePath` parameter.

contentCollection
Use the `contentCollection` parameter for a content folder request. The value you provide for `contentCollection` must correspond to the name of a content folder in Experience Manager, for example, `Search Box Auto Suggest Content`. `InvokeAssembler` creates a `ContentSlotConfig` component and inserts the `contentCollection` parameter in its `contentUri` property. Note that the `ContentSlotConfig` component specifies both the content folder and the number of content items to return from that folder. The number of items to return is specified using the `InvokeAssembler.ruleLimit` parameter, described next.

ruleLimit
This optional parameter is used in conjunction with the `contentCollection` parameter to specify the number of items that should be returned from the specified content folder.

Output Parameters

The `InvokeAssembler` servlet bean has one output parameter, `contentItem`. This parameter contains the root `ContentItem` returned by the Assembler. If this content item is empty, the request was not an Assembler request.

Open Parameters

The `InvokeAssembler` has two open parameters.

output
Rendered when the Assembler returns a `ContentItem`.

error
Rendered if the Assembler returns a `ContentItem` with an `@error` key. The presence of this key indicates that the `ContentItem` does not contain any content because the Assembler threw an exception or returned an error.

Example

This code snippet shows how to use the `InvokeAssembler` servlet bean on a page:

```
<dsp:importbean bean="/atg/endeca/assembler/droplet/InvokeAssembler"/>
<dsp:droplet name="InvokeAssembler">
  <dsp:param name="includePath" value="/browse"/>
  <dsp:oparam name="output">
    <dsp:getvalueof var="contentItem"
                    vartype="com.endeca.infront.assembler.ContentItem"
                    param="contentItem" />
  </dsp:oparam>
</dsp:droplet>
```

Accessing Commonly Used Functionality in AssemblerTools

The /atg/endeca/assembler/AssemblerTools component provides commonly used functionality to other query integration components. This component's functionality includes:

- Making the actual content request to the Assembler by invoking the assemble() method on the NucleusAssembler instance and passing it the request ContentItem.
- Assisting other components by calculating a content path based on the page request URL. The content path identifies the page in Experience Manager whose content should be rendered.
- Identifying the renderer mapping component to use for the request.

The AssemblerTools component is of class atg.endeca.assember.AssemblerTools and it has the following core method:

```
public ContentItem invokeAssembler(ContentItem pContentItem)
```

Creating the Assembler Instance and Starting Content Assembly

The AssemblerTools component has a configurable property, assemblerFactory, that out of the box is set to /atg/endeca/assembler/NucleusAssemblerFactory. The NucleusAssemblerFactory component is responsible for creating the Assembler instance that collects and organizes content. The AssemblerTools.invokeAssembler() method calls createAssembler() on the NucleusAssemblerFactory component to create an Assembler instance and then it calls assemble() on that instance to begin the content assembly process. More details on the NucleusAssemblerFactory component can be found in the Querying the Assembler (page 106) section.

Calculating the Content Path from the Page Request URL

Note: The information in this section applies to page requests processed by the AssemblerPipelineServlet only. For information about page requests that are processed using the InvokeAssembler servlet bean, see the InvokeAssembler (page 95) section.

For page requests processed by the AssemblerPipelineServlet, the AssemblerTools.getContentPath() method calculates the content path to pass to the Assembler (for example, /browse). The content path identifies the page in Experience Manager whose content should be rendered. The content path is relative to the current site; for example, if the current site is storeus and the content path is /browse, then the /browse page will be retrieved for the storeus site.

The getContentPath() method extracts the content path from the request URL by removing substrings that match the values of properties of the siteConfiguration item for the site. For example, if a request is made to http://localhost:8080/crs/storeus/browse/:

1. The getContentPath() method gets the request URI using the atg.servlet.ServletUtil class. In this case, the request URI is:

   /crs/storeus/browse/

2. If the AssemblerTools.removeSiteBaseURL property is true, getContentPath() compares the request URI with the site base URL (the value of the siteConfiguration item's productionURL property). If the AssemblerTools.includeAdditionalProductionURLs property is also true, the getContentPath() method compares the request URI with the values of the siteConfiguration item's additionalProductionURLs property, as well as with the value of productionURL. If one of the URLs matches a substring of the request URI, getContentPath() removes that substring from the request URI.

3. If the AssemblerTools.removeContextRoot property is true and the site base URL has not been removed, getContentPath() compares the request URI with the context root (the value of the siteConfiguration item's contextRoot property). If there is a match, getContentPath() removes the context root from the request URI.

So, in this example, if one of the production URLs is /crs/storeus, the resulting content path is /browse/. If none of the URLs match the request URI and the context root is /crs, the resulting content path is /storeus/browse/.

Identifying the Renderer Mapping Component to Use for the Request

The AssemblerTools.defaultContentItemToRendererPath property specifies the default component that should be used to map a response ContentItem to its correct renderer. Having this default ensures that the same mapping component is used across all web sites:

```
# Our default service for mapping from a ContentItem to the path of
# its corresponding JSP rendering page
defaultContentItemToRendererPath=cartridge/renderer/ContentItemToRendererPath
```

You can override this setting on a web application-specific basis by specifying a context-param in your application's web.xml file. The name of the parameter must be contentItemToRendererPath and the value must specify the Nucleus path of the mapping component you want to use:

```
<context-param>
  <param-name>contentItemToRendererPath</param-name>
  <param-value>Nucleus-path-to-mapper</param-value>
</context-param>
```

Creating the SiteState Component

For page requests, a request-scoped /atg/endeca/assembler/SiteState component must be resolved. This component contains the siteId for the current request as well as the page URI, or contentPath, being requested. Page requests begin when a ContentInclude object is passed to the NucleusAssembler component, after which the ContentIncludeHandler is invoked to resolve the SiteState component and the page content.

The SiteState component has the following properties:

- siteId: The Guided Search Site ID for the site that the current request resolves to, for example, /storeSiteUS. Note that the Guided Search Site ID identifies the correct site within the EAC application. This ID is distinct and different from the site ID that is part of a site's definition in the Site repository. Each site in

the Site repository will have a site ID. It may also have a corresponding Guided Search Site ID. The purpose of the Guided Search Site ID property is to create a mapping between a site definition in the Site repository and its corresponding site in the EAC application.

- `contentPath`: The path to the page for the current request, for example, `/browse`. This path is relative to the site specified in the `siteId` property.
- `properties`: This map provides a storage mechanism for additional properties you may want to include with your `SiteState` component. Out of the box, it is empty.

To resolve the `ContentIncludeHandler` component's reference to the `SiteState` component, Nucleus calls the `createSiteState()` method of the `com.endeca.infront.site.SiteStateBuilder` class. The `SiteStateBuilder` class is a factory class that constructs a `SiteState` component for the current request. To create the `SiteState` component, the `SiteStateBuilder` class determines both the site context and the `contentPath` of the Experience Manager page being requested, as described below.

To establish the site context, the `SiteStateBuilder` component uses a series of parsers. Each parser contains logic that determines the Guided Search site ID for the current request and then returns a `SiteState` object with a populated `siteId` property. A parser must implement the `atg.endeca.assembler.multisite.SiteStateParser` interface, which has the following core method:

```
public interface SiteStateParser {
    public SiteState parseSiteState(HttpServletRequest request, SiteManager
                                    siteManager);
}
```

The `SiteStateBuilder.siteStateParsers` property contains a list of parsers that are executed in the configured order to resolve the site context. Out of the box, this property is set to a single parser, the `/atg/endeca/assembler/multisite/SiteStateParser` component, which looks at the site context for the request and extracts the Guided Search Site ID from that site's definition in the Site repository.

In the event that the parsers defined in the `siteStateParsers` property fail to determine a Guided Search site ID and return a `SiteState` object, the `SiteStateBuilder.defaultSiteStateParser` property references a default `SiteStateParser`. Out of the box, the `defaultSiteStateParser` property references the `/atg/endeca/assembler/multisite/DefaultSiteStateParser` component. This component uses the default site configured for the EAC application to determine the site context. Note that it is the Experience Manager administrator's responsibility to specify the default site when creating an EAC application. If a default site is not specified for an EAC application, one of the following scenarios occurs:

- If there is only one site, then that site is used as the default.
- If there are multiple sites in the EAC application, the `SiteState.siteId` property will contain a value of `@error:siteNotFound`, which leads the Assembler to return a `FileNotFound` error.

Assuming one of the parsers executed successfully, the result is a `SiteState` object with the Guided Search site ID stored in its `siteId` property.

After determining the site context, the `SiteStateBuilder` class invokes the `/atg/endeca/assembler/multisite/ContentPathTranslator` component specified in the `SiteStateBuilder.contentPathTranslator` property. This component translates the original request URL into an Experience Manager content path, for example, `/browse`. To calculate the content path, the `ContentPathTranslator` component calls the `AssemblerTools.getContentPath()` method. This method encapsulates the Oracle Commerce Core Platform's logic for calculating the content path from the request URL (note that this logic could be replaced by a different application-specific class and method). See the Calculating the Content Path from the Page Request URL (page 97) section for details on how the `getContentPath()` method works.

After the SiteStateParser and ContentPathTranslator components have executed, a SiteState component exists for the current request and it has the site context and content path information the ContentIncludeHandler needs to locate the correct content in the Experience Manager pages hierarchy.

Defining Global Assembler Settings

The /atg/endeca/assembler/cartridge/manager/AssemblerSettings component defines global Assembler settings and is referenced by various components. The NucleusAssemblerSettings component is of class atg.endeca.assembler.NucleusAssemblerSettings, which is an extension of the class com.endeca.infront.assembler.AssemblerSettings. It has the following properties:

- defaultExperienceManagerPrefix: Defaults to /pages. This value is used by the /atg/endeca/assembler/cartridge/manager/WorkbenchContentSource component when it calculates the absolute path to a page in Experience Manager. All page content in Experience Manager resides under a /pages root.
- defaultGuidedSearchPrefix: Defaults to /service. This value is used by the /atg/endeca/assembler/cartridge/manager/WorkbenchContentSource component when it calculates the absolute path to a page in an application that uses Guided Search only (that is, without the Experience Manager).
- experienceManager: Defaults to true. Used by the AssemblerTools.isExperienceManager() method to determine if Experience Manager is available.

Connecting to the Workbench and MDEX

Some cartridges need to communicate with the EAC applications managed by the Workbench server while others need to communicate directly with the MDEX engines to do their work. The Guided Search integration includes a number of components to facilitate both types of communication.

AssemblerApplicationConfiguration Component

The atg.endeca.assembler.configuration.AssemblerApplicationConfiguration class configures the following:

- Workbench host information
- MDEX host and port information
- The method to use (direct calls to the Workbench versus retrieving content that the Workbench has stored on the file system) when retrieving content in a multi-EAC application environment.

This information complements the configuration stored the /atg/endeca/ApplicationConfiguration component, and enables communication with both the EAC applications managed by the Workbench server and any MDEX instances.

The Guided Search integration includes a component of the AssemblerApplicationConfiguration class, /atg/endeca/assembler/AssemblerApplicationConfiguration, that other components reference to retrieve the Workbench and MDEX connection details. The

AssemblerApplicationConfiguration component also has an applicationConfiguration property that points to the ApplicationConfiguration component:

```
applicationConfiguration=/atg/endeca/ApplicationConfiguration
```

This section provides information on how the AssemblerApplicationConfiguration component calculates these details, while the sections after provide information on the components that use them. The following chapter, *Retrieving Promoted Content* (page 115), provides details on the different content retrieval methods and how to configure them.

Creating Application-specific Workbench Connections

Note: This section introduces the WorkbenchContentSource and DefaultWorkbenchContentSource components, in the context of what the AssemblerApplicationConfiguration component does with them. Additional information is provided about these component types in the following sections.

The /atg/endeca/assembler/cartridge/manager/WorkbenchContentSource component holds details for connecting to a particular EAC application managed by the Workbench server (or, to be more specific, it functions as an alias for other components that calculate the connection details based on the environment and the current request). It is a requirement that a globally-scoped com.endeca.infront.content.source.WorkbenchContentSource object be instantiated for each EAC application in your environment before any content requests are made. Environments that have multiple EAC applications (for example, a separate application for each language or site), will need multiple WorkbenchContentSource components. The AssemblerApplicationConfiguration component is responsible for creating these components when necessary.

To create the application-specific WorkbenchContentSource components, the AssemblerApplicationConfiguration component resolves a prototype-scoped /atg/endeca/assembler/cartridge/manager/PrototypeWorkbenchContentSource component, which is of class atg.endeca.assembler.content.ExtendedWorkbenchContentSource, and inserts it into the Nucleus global scope under a new name that follows this pattern:

WorkbenchContentSource_*EAC-application-key*

Adding the *EAC-application-key* to the end of the WorkbenchContentSource component name uniquely identifies the WorkbenchContentSource component as the one to use for a given EAC application.

The PrototypeWorkbenchContentSource configuration includes a $basedOn property that references the /atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource component, where arguments for the WorkbenchContentSource constructor are provided. The PrototypeWorkbenchContentSource component gets its settings from the DefaultWorkbenchContent component, with the exception of the EAC application name, which it gets from the AssemblerApplicationConfiguration component's currentInitializingWorkbenchContentSourceApplicationName property.

Determining Which MDEX to Use

The AssemblerApplicationConfiguration component determines which host name and port to use to connect to the correct MDEX engine for any given request. The /atg/endeca/assembler/cartridge/manager/MdexResource component, which represents the connection to a single MDEX, refers to the AssemblerApplicationConfiguration component when creating a connection for a specific request.

The MDEX host and port values are stored in the AssemblerApplicationConfiguration.currentMdexHostname and AssemblerApplicationConfiguration.currentMdexPort properties, respectively. The

AssemblerApplicationConfiguration component includes configuration settings that specify how the currentMdexHost and currentMdex port properties are determined.

To direct a request to the correct MDEX, you configure the applicationKeyToMdexHostAndPort property on the /atg/endeca/ApplicationConfiguration component. The AssemblerApplicationConfiguration component also has an applicationKeyToMdexHostAndPort property that is set automatically to the value of the ApplicationConfiguration.applicationKeyToMdexHostAndPort property.

The applicationKeyToMdexHostAndPort property is a map where the keys identify each EAC application and the values specify the host names and port numbers for the MDEX engines associated with each application.

If you have a single EAC application, the key is default, and you map it to the MDEX as follows:

```
applicationKeyToMdexHostAndPort=\
   default=host:port
```

For example:

```
applicationKeyToMdexHostAndPort=\
   default=myHost.example.com:15300
```

Note that if the applicationKeyToMdexHostAndPort property is not set, the hostname and port are obtained from the values of the AssemblerApplicationConfiguration.defaultMdexHostName and AssemblerApplicationConfiguration.defaultMdexPort properties . These properties default to localhost and 15000 respectively, but you can explicitly set them to other values. However, if you set applicationKeyToMdexHostAndPort, it overrides these properties.

If you have multiple applications (for example, a separate application for each language), the keys depend on the routing strategy you are using:

- If you are using SingleApplicationRoutingStrategy and you have a separate MDEX for each language, the keys are the two-letter codes for the languages (for example, en for English, fr for French, it for Italian).
- If you are using SiteApplicationRoutingStrategy and you have a separate MDEX for each site, the keys are the site IDs.
- If you are using SiteApplicationRoutingStrategy and you have a separate MDEX for each combination of site and language, each key is formed by concatenating the site ID with the language code, separated by the underscore character (for example, storeSiteUS_fr).
- If you are using GroupingApplicationRoutingStrategy, the keys are the names of EAC applications.

To determine which MDEX to direct a request to, the component specified by the AssemblerApplicationConfiguration.routingObjectAdapter property examines the request and related objects to find locale and site information. This component, which is of a class that implements the RequestRoutingObjectAdapter interface, must match the routing strategy you are using. For example, if your routing strategy is SiteApplicationRoutingStrategy, the routingObjectAdapter property should be set to SiteRequestRoutingObjectAdapter.

Based on the information returned by the RequestRoutingObjectAdapter component, the AssemblerApplicationConfiguration component retrieves the key in one of two ways:

- If the routing strategy is SingleApplicationRoutingStrategy or SiteApplicationRoutingStrategy, the key is retrieved from the ApplicationConfiguration component.

- If the routing strategy is `GroupingApplicationRoutingStrategy`, the key is retrieved from the `GroupingApplicationRoutingStrategy` component.

The key is then used to retrieve the host and port values from the `applicationKeyToMdexHostAndPort` map.

For example, if your environment has two EAC applications to support two languages, English and German, and your routing strategy is `SingleApplicationRoutingStrategy`, the `applicationKeyToMdexHostAndPort` setting might be:

```
applicationKeyToMdexHostAndPort=\
     en=localhost:15000,\
     de=localhost:15100
```

For more information about the `ApplicationConfiguration` component, see the Configuring the ApplicationConfiguration Component (page 4) section of the *Introduction* (page 1). For more information about routing strategies, see the *Routing* (page 9)chapter.

Connecting to an MDEX

The `/atg/endeca/assembler/cartridge/manager/MdexResource` component, of class `com.endeca.infront.navigation.model.MdexResource`, is a request-scoped component that represents a connection to a single MDEX. The `NucleusAssembler` uses this component to connect to the correct MDEX for content.

The `MdexResource` component has `host` and `port` properties that represent the MDEX host and port to use for the current request. The `MdexResource` component gets the values for these properties from the `AssemblerApplicationConfiguration` component, specifically, the `AssemblerApplicationConfiguration.currentMdexHostName` and `AssemblerApplicationConfiguration.currentMdexPort` properties.

Connecting to the Workbench Server

Oracle Commerce Core Platform has several components for creating a connection to an EAC application managed by the Workbench server. The connection components can vary depending on whether your environment has a single EAC application or multiple applications (for example, to support multiple languages). Here is a brief overview of the process:

1. On startup, the `/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource` component is instantiated. This component contains details for connecting to a default EAC application.

2. If the environment has more than one EAC application, the `AssemblerApplicationConfiguration` component creates globally-scoped, `WorkbenchContentSource_`*`EAC-application-key`* components for each EAC application. Each component has a suffix that identifies which EAC application the component is for, for example, `WorkbenchContentSource_en` and `WorkbenchContentSource_de`. These application-specific components have a set of properties that are comparable to those in the `DefaultWorkbenchContentSource`, but contain values that are specific to each individual EAC application.

3. The `NucleusAssembler` resolves the `/atg/endeca/assembler/cartridge/manager/WorkbenchContentSource` component. This component in turn resolves either the `/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource` component or the `/atg/endeca/assembler/cartridge/manager/PerApplicationWorkbenchContentSourceResolver` as the `WorkbenchContentSource` to use for the current request.

4. If the DefaultWorkbenchContentSource is resolved, the connection details defined by this component are used when retrieving content.

5. If the PerApplicationWorkbenchContentSourceResolver is resolved, the component relies on the AssemblerApplicationConfiguration to determine what the current EAC application is and then it references the correct EAC application-specific WorkbenchContentSource that the AssemblerApplicationConfiguration component has already created in step 2.

The remaining sections provide more details on the individual Workbench-related components.

WorkbenchContentSource

The /atg/endeca/assembler/cartridge/manager/WorkbenchContentSource component represents the connection to a particular EAC application managed by the Workbench server. The NucleusAssembler class uses this component to connect to an EAC application and request content, using the content retrieval method specified.

Out of the box, the WorkbenchContentSource component uses a $basedOn property set to the /atg/endeca/assembler/cartridge/manager/PerApplicationWorkbenchContentSourceResolver, which is a request-scoped component that determines which EAC application-specific WorkbenchContentSource to use, based on the current request. This default configuration is primarily intended to support environments that have multiple EAC applications, although it works for single-application environments as well.

The WorkbenchContentSource properties file also includes some configuration, which has been commented out, that is more efficient for environments that have a single EAC application:

```
# $class=atg.nucleus.GenericReference
# $scope=global
# componentPath=DefaultWorkbenchContentSource
```

This configuration creates a globally-scoped WorkbenchContentSource component that gets its connection details from the /atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource component. This approach is more efficient for a single EAC application environment because it avoids having to resolve the WorkbenchContentSource for every request. If you have a single EAC application environment, you can use this configuration instead.

The following sections provide some additional details on the DefaultWorkbenchContentSource and PerApplicationWorkbenchContentSource components that provide the connection details stored in a WorkbenchContentSource component.

DefaultWorkbenchContentSource

The /atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource component, is a globally-scoped component of class atg.endeca.assembler.content.ExtendedWorkbenchContentSource. In a single EAC application environment, the DefaultWorkbenchContentSource component provides connection details for the single EAC application managed by the Workbench server that should be used for all requests. In a multi-application environment, this component provides connection details to a default EAC application when the PerApplicationWorkbenchContentSourceResolver cannot resolve an application-specific WorkbenchContentSource.

Out of the box, this component is included in the initialServices property of the /initial component, to ensure that it is created on start up.

```
initialServices+=\
   /atg/endeca/assembler/AssemblerPipelineServlet,\
```

```
/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource
```

The `DefaultWorkbenchContentSource` component has a set of properties that are used to create the `WorkbenchContentSource` that is used to connect to the Workbench. The `DefaultWorkbenchContentSource` component gets the values for some of these properties from the `ApplicationConfiguration` and `AssemblerApplicationConfiguration` components. It is the responsibility of these other two components to calculate the correct EAC application and Workbench server connection details to use. The `DefaultWorkbenchContentSource` properties include:

- `appName`: The EAC application name. Defaults to `/atg/endeca/ApplicationConfiguration.defaultApplicationName`.
- `host`: The Workbench server host name. Defaults to `../../AssemblerApplicationConfiguration.workbenchHostName`.
- `serverPort`: The port number that `WorkbenchContentSource` components must use to retrieve content from the Workbench. Defaults to 8007.

  Note that the `serverPort` property refers to the port used for content retrieval (8007, by default), as opposed to the port used to connect to the Workbench (8006, by default).
- `storeFactory`: A reference to the store factory to use for retrieving content for the default EAC application. Defaults to `/atg/endeca/assembler/cartridge/manager/DefaultFileStoreFactory`. See the *Retrieving Promoted Content* (page 115) chapter for more information.
- `defaultSiteRootPath`: The site root path to use when calculating the absolute path to the content being retrieved. Defaults to `AssemblerSettings.defaultExperienceManagerPrefix`.
- `siteManager`: A reference to the `SiteManager` component for retrieving site-based information, such as the current site, for the request. Defaults to `/atg/endeca/assembler/multisite/SiteManager`.

PerApplicationWorkbenchContentSourceResolver

In an environment that has multiple EAC applications, it is the `/atg/endeca/assembler/cartridge/manager/PerApplicationWorkbenchContentSourceResolver` component's responsibility to determine the correct globally-scoped, application-specific `WorkbenchContentSource` component to use for the current request. This component also defines a default `WorkbenchContentSource` component to use if an application-specific version cannot be found. `PerApplicationWorkbenchContentSourceResolver` is of class `atg.endeca.assembler.configuration.PerEndecaApplicationGenericReference`, which extends the `atg.nucleus.GenericReference` class to calculate the correct component to reference based on the EAC application key of the current request.

Note that `PerApplicationWorkbenchContentSourceResolver` is request-scoped. This means that the globally-scoped `WorkbenchContentSource` component that it resolves and references gets inserted into the request scope as an alias. This effectively allows the application to resolve the `WorkbenchContentSource` component on a per-request basis.

To perform its tasks, the `PerApplicationWorkbenchContentSourceResolver` component has the following properties:

- `defaultComponentPath`: The Nucleus path of the `WorkbenchContentSource` component to default to if an EAC application-specific version cannot be resolved. Defaults to `/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource`.
- `componentBasePath`: The base path for the application-specific `WorkbenchContentSource` components. `PerApplicationWorkbenchContentSourceResolver` adds the EAC application keys, such as `_en` and

_es, as suffixes to this path to resolve the correct WorkbenchContentSource to reference. Defaults to /atg/endeca/assembler/cartridge/manager/WorkbenchContentSource.

- assemblerApplicationConfiguration: The Nucleus path to the AssemblerApplicationConfiguration component, where the PerApplicationWorkbenchContentSourceResolver gets the application keys. Defaults to ../../AssemblerApplicationConfiguration.
- useDefaultIfSingleApplication: Indicates that the PerApplicationWorkbenchContentSourceResolver should use the DefaultWorkbenchContentSource if there is only one EAC application and avoid resolving an application-specific WorkbenchContentSource.

Manually Adding Application-specific WorkbenchContentSource Components

It is a requirement that the WorkbenchContentSource component used to communicate with any given EAC application be globally scoped and started up front, before any requests are made. To accommodate this requirement, the ApplicationAssemblerConfiguration component automatically creates corresponding WorkbenchContentSource components for each EAC application on start up.

If the automatically-created WorkbenchContentSource components are not sufficient for your needs, you can manually create .properties files for other application-specific WorkbenchContentSource components, for example:

```
$basedOn=DefaultWorkbenchContentSource

# EAC application name
appName=EAC-application-name

# Workbench host
host=Workbench-host-name

# Workbench content retrieval port, defaults to 8007
serverPort=8007
```

Note that the serverPort property refers to the port used for content retrieval (8007, by default), as opposed to the port used to connect to the Workbench (8006, by default).

After creating the EAC application-specific WorkbenchContentSource components, you must add them to the intialServices property of the /initial component so that they are started on application start-up, for example:

```
initialServices+=\
/atg/endeca/assembler/cartridge/manager/WorkbenchContentSource_EAC-application-key
```

Querying the Assembler

The atg.endeca.assembler.NucleusAssemblerFactory class is responsible for creating the atg.endeca.assembler.NucleusAssembler instance that retrieves and organizes content. The

NucleusAssemblerFactory class implements the com.endeca.infront.assembler.AssemblerFactory interface and defines a createAssembler() method that the AssemblerTools component invokes to get a NucleusAssembler instance. NucleusAssembler is an inner class of NucleusAssemblerFactory. It implements the com.endeca.infront.assembler.Assembler interface and defines an assemble() method that the AssemblerTools component invokes to begin a query. The following code excerpt from AssemblerTools.java shows the use of these two methods:

```
// Get the assembler factory and create an Assembler
Assembler assembler = getAssemblerFactory().createAssembler();
assembler.addAssemblerEventListener(new AssemblerEventAdapter());
    // Assemble the content
ContentItem responseContentItem = assembler.assemble(pContentItem);
```

In addition to retrieving the base content from the cartridge XML configuration files, the NucleusAssembler class also modifies that content as necessary using CartridgeHandler components. The NucleusAssemblerFactory component provides the NucleusAssembler class with the configuration it needs to find the correct CartridgeHandler components. CartridgeHandlers can be found either by using a default naming strategy (that is, looking for a Nucleus component named after the cartridgeType in one of the NucleusAssemblerFactory component's path properties), or via an explicit mapping. To support these strategies, the NucleusAssemblerFactory component provides the following properties:

- experienceManagerHandlerPath: Defaults to the /atg/endeca/assembler/cartridge/handler/experiencemanager folder.
- guidedSearchHandlerPath: Defaults to the /atg/endeca/assembler/cartridge/handler/guidedsearch folder.
- defaultHandlerPath: Defaults to the /atg/endeca/assembler/cartridge/handler folder.
- handlerMapping: A Map<String, String> property that provides a map from the cartridgeType to the Nucleus path of the corresponding CartridgeHandler component. This property can be used to override the default mapping specified in path properties.

When looking for a cartridge handler, the NucleusAssembler class first invokes the AssemblerTools.isExperienceManager() method to determine if Experience Manager is present or not. If isExperienceManager() returns true, the NucleusAssembler class tries to locate the correct handler in the path specified by the NucleusAssemblerFactory.experienceManagerHandlerPath property. For example, for the MyCartridge cartridge, the NucleusAssembler class would look for the handler called /atg/endeca/assembler/cartridge/handler/experiencemanager/MyCartridge. If isExperienceManager() returns false, the NucleusAssembler class looks for the handler in the path specified by the NucleusAssemblerFactory.guidedSearchHandlerPath property. If neither path resolves successfully, the NucleusAssembler class looks for the handler in the path specified by the NucleusAssemblerFactory.defaultHandlerPath. Finally, if the NucleusAssembler class still cannot find the correct handler, it looks at the explicit mappings defined in the NucleusAssemblerFactory.handlerMapping property.

Cartridge Handlers and Their Supporting Components

To use cartridges, the Oracle Commerce Core Platform must create Nucleus components for the cartridge handler classes and any classes that the handlers depend on. The DAF.Endeca.Assembler module includes

Nucleus component configuration for Platform-level cartridge handlers in the /atg/endeca/assembler/cartridge/handler and /atg/endeca/assembler/cartridge/handler/config Nucleus paths. The Store.Endeca.Assembler module includes component configuration for Commerce Reference Store-specific cartridges in those same Nucleus path locations.

In addition to the handler classes, cartridges rely on Experience Manager configuration and application rendering code. Because Experience Manager configuration and page rendering are both application-specific, the files that support all cartridges are included in the Commerce Reference Store modules.

The default folder that Nucleus will try to resolve cartridge handlers in is /atg/endeca/assembler/cartridge/handler. The /config subdirectory in that same location contains configuration components associated with the CartridgeHandler components. Similarly, /atg/endeca/assembler/cartridge/handler/xmgr and /atg/endeca/assembler/cartridge/handler/guidedsearch folders contain cartridge handlers that are specific to Experience Manager and Guided Search, respectively, and they also have their own /config sub-paths.

The components in the /atg/endeca/assembler/cartridge/manager Nucleus folder provide additional cartridge support outside of what can be found in the cartridge handlers themselves. For example, the NavigationStateBuilder and NavigationState components build and represent the current navigation state, respectively; the DefaultFilterState component represents the state of any filters; and the MdexRequestBuilder component builds MDEX requests.

Note: Currently, the /atg/endeca/assembler/cartridge/handler/xmgr and /atg/endeca/assembler/cartridge/handler/guidedsearch folders are empty and function only as placeholders for future components.

Providing Access to the HTTP Request to the Cartridges

The /atg/endeca/servlet/request/NucleusHttpServletRequestProvider component, which is of class atg.endeca.servlet.request.NucleusHttpServletRequestProvider, provides access to the current request to various components in both the /atg/endeca/assembler/cartridge/handler and /atg/endeca/assembler/cartridge/manager Nucleus folders.

Controlling How Cartridges Generate Link URLs

If a cartridge needs to provide links to another navigation or record state, the handler for that cartridge store the necessary information to build those links in com.endeca.infront.cartridge.model.NavigationAction objects that are returned as part of the response ContentItem. The NavigationAction objects have properties that define:

- The site context for the link, in the form of a reference to the SiteState object
- The site root for the link, for example, /pages
- The content path for the link, for example, /browse
- The navigation state query parameters for the link, for example, ?N=4294967263

For example, this NavigationAction object contains the information necessary to create a Remove All Breadcrumbs link on a page:

```
removeAllAction: {@class:
 "com.endeca.infront.cartridge.model.NavigationAction",navigationState: "?
format=json",contentPath: "/browse",siteRootPath: "/pages",siteState: {@class:
 "com.endeca.infront.site.model.SiteState",contentPath: "/browse",siteId: "/
storeSiteUS",properties: { },matchedUrlPattern: ""}},
```

The com.endeca.infront.cartridge.NavigationCartridgeHandler class, and its subclasses, has a reference to the SiteState object, allowing those classes to add a SiteState reference to the NavigationAction objects they create. For the other link-related properties, the NavigationCartridgeHandler classes rely on two additional components, BasicUrlFormatter and DefaultActionPathProvider, described in the sections below.

It is the responsibility of the page code that renders the links to use the NavigationObject properties to build appropriate link URLs.

BasicUrlFormatter

The /atg/endeca/url/basic/BasicUrlFormatter component is of class com.endeca.soleng.urlformatter.basic.BasicUrlFormatter. This class is responsible for serializing query parameters from a navigation state, for example, ?N=4294967263. It includes properties such as defaultEncoding and prependQuestionMarks that control how the query parameters are generated. Out of the box these properties are set to UTF-8 and true, respectively.

For more information on the BasicUrlFormatter class, refer to the *Oracle Commerce Guided Search Assembler Application Developer's Guide*.

DefaultActionPathProvider

The /atg/endeca/assembler/cartridge/manager/DefaultActionPathProvider component, of class atg.endeca.assembler.navigation.DefaultActionPathProvider, calculates the site root path and the content path for follow-on links. For example, in the link below, the site root path is /pages and the content path is /browse, while the remainder of the URL represents the query parameters that define the request.

```
/pages/browse?N=4294967263
```

The combination of the site root path and the content path is called the *action path*.

To calculate the site root path and the content path, the DefaultActionPathProvider class implements the com.endeca.infront.navigation.url.ActionPathProvider interface and its four methods:

- getDefaultNavigationActionContentPath(): Returns the content path for a navigation action.
- getDefaultNavigationActionSiteRootPath(): Returns the site root path for a navigation action.
- getDefaultRecordActionContentPath(): Returns the content path for a record action.
- getDefaultRecordActionSiteRootPath(): Returns the site root path for a record action.

The DefaultActionPathProvider component also has the following properties that support site root and content path generation:

- `defaultExperienceManagerNavigationActionPath`: The content path to use for navigation requests when Experience Manager is installed and no other content path can be resolved, defaults to `/browse`.
- `defaultExperienceManagerRecordActionPath`: The content path to use for record requests when Experience Manager is installed and no other path can be resolved, defaults to `/product`.
- `defaultGuidedSearchNavigationActionPath`: The content path to use for navigation requests when Guided Search is installed, defaults to `/guidedsearch`.
- `defaultGuidedSearchRecordActionPath`: The content path to use for record requests when Guided Search is installed, defaults to `/recorddetails`.
- `navigationActionUriMap`: A map whose keys are navigation request action paths and whose values are replacement action paths that should be substituted for the key action paths. For example, a `/pages/[site]/brand` action path can be replaced with a `/pages/[site]/browse` action path. This map can be used when overriding the action path of the current request is necessary. The keys are in regular expression form, so things such as query parameters are ignored.
- `recordActionUriMap`: Analogous to `navigationActionUriMap`, this is a map whose keys represent record request action paths and whose values are replacement action paths that should be substituted for the key action paths. The keys are in regular expression form.
- `assemblerTools`: A reference to the `AssemblerTools` component. The `AssemblerTools` component provides a reference to the `AssemblerSettings` component, where the default site root paths are defined. Defaults to `/atg/endeca/assembler/AssemblerTools`.
- `currentRequest`: Provides access to the current request's details. Defaults to `/OriginatingRequest`.
- `contentSource`: A reference to the `WorkbenchContentSource` component used to connect with the correct Workbench server and application. Defaults to `/atg/endeca/assembler/cartridge/manager/WorkbenchContentSource`. See Connecting to the Workbench and MDEX (page 100) for details on this component.

Calculating the Content Path

To calculate the content path for a navigation action, the `DefaultActionPathProvider.getDefaultNavigationActionContentPath()` method is invoked. This method calls the `AssemblerTools.isExperienceManager()` method to determine if Experience Manager is in use. If so, the `DefaultActionPathProvider` component calculates the content path to return using the process described in the next paragraph. If Experience Manager is not in use, the `DefaultActionPathProvider` component returns the value of its `defaultGuidedSearchNavigationActionPath` property, which defaults to `/guidedsearch`.

To calculate the content path for navigation actions when Experience Manager is in use, the `DefaultActionPathProvider` component retrieves the value of the `/atg/endeca/assembler/multisite/SiteState.contentPath` property and looks for a match in the keys of its `navigationActionUriMap` property. If a match is found, the `DefaultActionPathProvider` component returns the content path portion of the matching entry's value. If no match is found, the `DefaultActionPathProvider` component returns the content path it retrieved from the `SiteState` object. If it cannot resolve a content path from either the `SiteState` object or the `navigationActionUriMap`, the `DefaultActionPathProvider` component returns the value specified in its `defaultExperienceManagerNavigationActionPath` property, which defaults to `/browse`.

The process for calculating the content path for record actions when Experience Manager is in use is very similar to that for navigation actions. The `DefaultActionPathProvider` component retrieves the value of the `SiteState.contentPath` property, however, it uses the `recordActionUriMap` property for the lookup instead. Also, if a content path cannot be resolved from either the

SiteState object or the recordActionUriMap, this method returns the value specified in the DefaultActionPathProvider.defaultExperienceManagerRecordActionPath property, which defaults to /product.

Calculating the Site Root Path

To calculate the site root path for a navigation action, the DefaultActionPathProvider.getDefaultNavigationActionSiteRootPath() method uses a combination of the /atg/endeca/assembler/cartridge/manager/WorkbenchContentSource component and com.endeca.infront.content.source.ContentLocator objects. The WorkbenchContentSource component provides access to the ContentLocator objects, and the ContentLocator objects provide the site root path. Specifically, the DefaultActionPathProvider does the following:

1. Passes the SiteState.contentPath value to the WorkbenchContentSource component to get a ContentLocator object for that path. This ContentLocator object has two properties:
   - A contentPath property that contains the absolute path for the associated content, for example, /pages/[site]/browse.
   - A siteRootPath property that contains the site root for the associated content.
2. DefaultActionPathProvider compares the value of the ContentLocator.contentPath property to the keys in its navigationActionUriMap map to determine if a replacement is needed.

   If a match is found, DefaultActionPathProvider makes another request to the WorkbenchContentSource for a new ContentLocator, this time using the value for the matching key. This new ContentLocator object's siteRootPath is then used as the SiteRootPath for the follow-on links.

   If no match is found, DefaultActionPathProvider uses the siteRootPath from the ContentLocator retrieved in step 1.

   If step 1 did not return a ContentLocator object, the DefaultActionPathProvider component calls the AssemblerTools.isExperienceManager() method to determine if Experience Manager is in use. If so, the DefaultActionPathProvider component invokes the AssemblerTools.assemblerSettings() method to retrieve the default site root prefix. This prefix is dependent on whether or not Experience Manager or Guided Search is installed and defaults to /pages and /service, respectively.

The process for calculating the site root path for record actions is very similar to that for navigation actions. The getDefaultRecordActionSiteRootPath() method is invoked. This method performs similarly to the getDefaultNavigationActionSiteRootPath() method, however, it uses the recordActionUriMap property for the lookup instead. The process for retrieving a default site root in cases where one cannot be resolved from a ContentLocator object is the same; a call is made to the AssemblerTools.assemblerSettings() method to retrieve the default site root prefix.

DefaultActionPathProvider and the InvokeAssembler Servlet Bean

When using the /atg/endeca/assembler/droplet/InvokeAssembler servlet bean to retrieve content from the Assembler, there is no concept of a "current request." Because the DefaultActionPathProvider logic uses the current request's site root and content path values to do its calculations, the InvokeAssembler servlet bean provides navActionContentPath and recordActionContentPath parameters for passing in a value that can function as the current request's site root and content path. These parameters are used for navigation requests and record requests, respectively. The code sample below shows the use of the navActionContentPath.

```
<dsp:droplet name="InvokeAssembler">
```

```
    <dsp:param name="contentCollection" value="/content/Shared/Guided
                                               Navigation"/>
    <dsp:param name="navActionContentPath" value="/browse"/>
    <dsp:oparam name="output">

      <dsp:getvalueof var="contentItem"
                      vartype="com.endeca.infront.assembler.ContentItem"
                      param="contentItem" />

    </dsp:oparam>

  </dsp:droplet>
```

Retrieving Renderers

The Oracle Commerce Core Platform includes one component, `ContentItemToRendererPath`, and one `dsp` tag, `dsp:renderContentItem`, for retrieving the correct renderer for a content item.

ContentItemToRendererPath

The `/atg/endeca/assembler/cartridge/renderer/ContentItemToRendererPath` component is responsible for locating the correct renderer for the `ContentItem` that has been return by the Assembler in response to a request. The `ContentItemToRendererPath` component is an instance of the class `atg.endeca.assembler.cartridge.renderer.CartridgeRenderingPathMapperImpl`, which implements the `atg.endeca.assembler.cartridge.renderer.CartridgeRenderingMapper` interface. The core method of the `CartridgeRenderingMapper` interface is:

```
public String getRendererPathForContentItem(ContentItem pItem);
```

The `getRendererPathForContentItem()` method returns the web-app relative path of the JSP file used to render the `ContentItem`.

Creating the Path

The `ContentItemToRendererPath` component provides some configurable properties that control how a `ContentItem` is mapped to a JSP path:

- `formatString`: The string that defines the relative path of the JSP file. Defaults to `/cartridges/{cartridgeType}/{cartridgeType}{selectorSuffix}.jsp`. `{cartridgeType}` is replaced by the type of the current `ContentItem`, which is determined using the `cartridgeTypePropertyName` property, described below. `{selectorSuffix}` is provided by the `SelectorReplacementValueProducer`, also described below.
- `cartridgeTypePropertyName`: The name of the `ContentItem` property that contains the `cartridgeType`. Defaults to `cartridgeType`.
- `contentItemToReplacementPropertyNames`: A map that creates a relationship between a source `ContentItem` attribute's name and a `formatString` property name. You can use this map to make `ContentItem` properties available for use in the `formatString`.

- `replacementValueProducers`: An array of `ReplacementValueProducers`, described below, that makes additional values available for use in the `formatString`.

To create the path, `getRendererPathForContentItem()` creates a replacement map that gets populated with values calculated by other components or retrieved from other contexts. The replacement map values are then used to replace placeholders in the `ContentItemToRendererPath.formatString` property, resulting in a string that defines the relative path of the JSP file.

ReplacementValueProducer and SelectorReplacementValueProducer

The `atg.endeca.assembler.cartridge.renderer.ReplacementValueProducer` interface can be implemented by components that need to make new, perhaps dynamically-generated, values available for use in the replacement map and, by extension, the `formatString`. It contains one method that adds values to the replacement map.

```
/** Add any replacement values to pMap. Note that a given
 * instance may add a single value, multiple values, or none.
 *
 * @param pMap--The map to add parameters to.
 * @param pContentItem--The ContentItem (available for reference
 *   and calculating replacement values based on the content item)
 *   ContentItem should not be modified.
 * @param pRequest--The current request. May be null, if invoked
 *   outside of a request.
 */
public void addReplacementValues(Map<String, String> pMap,
                                 ContentItem pContentItem,
                                 HttpServletRequest pRequest);
```

Out of the box, the Core Platform includes one replacement value producer, the `/atg/endeca/assembler/cartridge/renderer/SelectorReplacementValueProducer`. This component adds a `selector` and `selectorSuffix` to the replacement map, if needed. A `selector` represents the type of device being used to view the web page, for example, a mobile device. The `selectorSuffix` is a corresponding suffix—for example, "_mobile"—that gets added to the end of the JSP renderer path, so that the correct JSP is rendered for that type of device.

The `SelectorReplacementValueProducer` component is of class `atg.endeca.assembler.cartridge.renderer` and its primary configurable properties are:

- `browserTypeToSelectorName`: A map where the key is the browser type and the value is the corresponding type of device (the "selector"). Out of the box, this property is configured to include the entry `iOSMobile=mobile`, which declares that when the browser type is `iOSMobile`, the value in the replacement map for `selector` is `mobile`. The `selectorSuffix` always has the same value as the `selector` with a preceding underscore, making the `selectorSuffix` in this case `_mobile`. If no matching browser type is found, `selector` and `selectorSuffix` are not set.

- `selectorKeyName`: The name of the key to use when putting the selector value into the replacement map. Defaults to `selector`.

- `selectorSuffixKeyName`: The name of the key to use when putting the selector suffix value into the replacement map. Defaults to `selectorSuffix`.

- `selectorOverrideParameterName`: The name of a request query parameter that can be used to override the selector setting in the replacement map. Defaults to `ciSelector`. This property allows you to force a selector value of `mobile` by having a `ciSelector` query parameter value of `mobile`.

dsp:renderContentItem

The dsp:renderContentItem JSP tag has two responsibilities:

- For a JSP response, it locates and dispatches to a rendering JSP page. The dsp:renderContentItem tag uses the ContentItemToRendererPath component to determine the path of the JSP page to include.
- It sets an HttpServletRequest.contentItem attribute to the specified contentItem. This provides a well-known attribute for rendering pages to pull data from; however, this attribute is set for the duration of the include only.

The dsp:renderContentItem tag supports the following tag attributes:

- contentItem (required) - The ContentItem to locate a rendering JSP page for. The value of the contentItem request attribute is also set to this ContentItem, for the duration of the include.
- format (optional) – Specifies whether the response should be serialized into JSON or XML. Acceptable values are json or xml.
- webApp (optional) - The web application that the include is relative to. By default, the current web application is used, but by passing another value in the webApp attribute, you can specify an include that is relative to a different web application. The value of webApp may either be the content root of the target web application (in which case, it must begin with a slash) or the display name of webApp (in which case, it is located via Oracle Commerce's WebAppRegistry; see the *Platform Programming Guide* for more information on the WebAppRegistry).
- var (optional) – The name of the request attribute to set. You can use var to override the default request attribute name of contentItem.

Similar to dsp:include, dsp:renderContentItem supports either nested dsp:param tags or dynamic attributes for setting additional parameters.

Configuring Keyword Redirects

In order for keyword redirects that have been defined in the Workbench to work in an environment that includes the Guided Search integration, you may have to do some additional configuration on the Oracle Commerce Platform side. Specifically, keyword redirects that point to servers other than the one where the Oracle Commerce Platform application is running require additional configuration. To add this additional configuration, modify the allowedHostNames property of the /atg/dynamo/servlet/pipeline/RedirectURLValidator component to include the host for the redirected URL. For example, for a keyword redirect that uses oracle as its term and http://oracle.com as its link, you must add the host oracle.com to the allowedHostNames property.

10 Retrieving Promoted Content

Each `WorkbenchContentSource` component uses a *store factory* of class `atg.endeca.assembler.content.ExtendedFileStoreFactory` to retrieve promoted content. This class extends the Guided Search `com.endeca.infront.content.source.FileStoreFactory` class.

This chapter describes how to configure store factory components and related components to retrieve promoted content. The configuration required differs depending on whether your environment uses a single MDEX engine or multiple MDEX engines.

For more information about content promotion, see the *Oracle Commerce Guided Search Administrator's Guide*.

Single-MDEX Environment

If your environment includes a single MDEX engine (for example, if all indexed content is in one language, or content is in multiple languages but all languages are indexed in the same MDEX), the `/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource` component uses the `/atg/endeca/assembler/cartridge/manager/DefaultFileStoreFactory` component to retrieve promoted content. (See the Connecting to the Workbench Server (page 103) section of the *Query Integration* (page 85) chapter for information about the `DefaultWorkbenchContentSource` component.)

The `DefaultFileStoreFactory` component includes the following properties:

- `isAuthoring`: A boolean that specifies whether or not the component is running in an authoring environment. The value of this property determines which other properties are taken into account. (For example, connection information such as the `serverPort` property applies only to an authoring environment.) Defaults to the value of the `previewEnabled` property of the `AssemblerSettings` component:

  ```
  isAuthoring^=AssemblerSettings.previewEnabled
  ```

- `appName`: The EAC application name. Defaults to the value of the `defaultApplicationName` property of the `/atg/endeca/ApplicationConfiguration` component:

  ```
  appName^=/atg/endeca/ApplicationConfiguration.defaultApplicationName
  ```

- `host`: The fully qualified host name of the machine running the Oracle Commerce Workbench. Defaults to the value of the `workbenchHostName` property of the `/atg/endeca/ApplicationConfiguration` component:

  ```
  host^=/atg/endeca/ApplicationConfiguration.workbenchHostName
  ```

- `serverPort`: The port on the host machine that the Workbench uses to publish Experience Manager content. Set by default to:

  ```
  serverPort=8007
  ```

- `clientPort`: The port on the client machine that is used to retrieve Experience Management content from the Workbench. If `clientPort` is set to -1 (the default), a port is assigned automatically.

  ```
  clientPort=-1
  ```

- `configurationPath`: Set this property to the file-system pathname of the directory to retrieve promoted content from. For example:

  ```
  configurationPath=\
  ToolsAndFrameworks/version/server/workspace/state/repository/ATG
  ```

In addition, the `/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource` component and the `/atg/endeca/assembler/admin/EndecaAdministrationService` component each have a `storeFactory` property that is configured by default to point to the `DefaultFileStoreFactory` component:

```
storeFactory=\
  /atg/endeca/assembler/cartridge/manager/DefaultFileStoreFactory
```

Multiple-MDEX Environment

If your environment includes multiple MDEX engines (for example, multiple languages with a separate MDEX per language), configuring the retrieval of content requires a few more steps than in a single-MDEX environment.

As discussed in the *Query Integration* (page 85) chapter, in a multiple-MDEX environment, the Guided Search integration uses a separate `WorkbenchContentSource` for each EAC application. Each `WorkbenchContentSource` requires a separate instance of `FileStoreFactory` to retrieve the appropriate content for the corresponding EAC application.

The `/atg/endeca/assembler/AssemblerApplicationConfiguration` component can automatically create a `FileStoreFactory` for each `WorkbenchContentSource` it creates and set a reference to that `FileStoreFactory` on the `WorkbenchContentSource`. This behavior is enabled by setting the value of the `useFileStoreFactory` property of the `AssemblerApplicationConfiguration` component to `true`.

In addition, the `/atg/endeca/assembler/admin/EndecaAdministrationService` component can manage the updates to the store factories. This behavior is enabled by changing the class of the component from `com.endeca.infront.assembler.servlet.admin.AdministrationService` to `atg.endeca.assembler.MultiAppAdministrationService`. The `MultiAppAdministrationService` class extends `AdministrationService` to enable handling of updates to multiple store factory instances.

Two options for configuring content retrieval in a multiple-MDEX environment are described below:

- Creating `FileStoreFactory` instances automatically from a prototype-scoped component.
- Creating `FileStoreFactory` instances from properties files.

Note that both sets of instructions assume you have already created all of the EAC applications and configured the various Oracle Commerce Platform routing components, as discussed in the *Routing* (page 9)chapter.

Creating FileStoreFactory Instances from a Prototype-Scoped Component

To create FileStoreFactory instances from a prototype-scoped component:

1. Modify the /atg/endeca/assembler/AssemblerApplicationConfiguration component in the local server configuration. Set the useFileStoreFactory property to true to automatically create a corresponding FileStoreFactory for each EAC application and set a reference to that FileStoreFactory on the application's WorkbenchContentSource:

   ```
   useFileStoreFactory=true
   ```

   The AssemblerApplicationConfiguration component has a prototypeFileStoreFactory property that points to the /atg/endeca/assembler/cartridge/manager/PrototypeFileStoreFactory component. (The PrototypeFileStoreFactory component's $basedOn property is set to the DefaultFileStoreFactory component described in the Single-MDEX Environment (page 115) section.) The FileStoreFactory instances are created from the PrototypeFileStoreFactory component.

2. Set the assemblerContentBaseDirectory property of the AssemblerApplicationConfiguration component to the file-system pathname of the directory to retrieve promoted content from. For example:

   ```
   assemblerContentBaseDirectory=\
   ToolsAndFrameworks/version/server/workspace/state/repository
   ```

   For each FileStoreFactory created, the corresponding EAC application name is appended to the assemblerContentBaseDirectory value to set the FileStoreFactory component's configurationPath property. For example, if assemblerContentBaseDirectory is set as shown above, the configurationPath property for an application named ATGes would be ToolsAndFrameworks/*version*/server/workspace/state/repository/ATGes.

3. Modify the /atg/endeca/assembler/admin/EndecaAdministrationService component in the local server configuration. Set the $class property to atg.endeca.assembler.MultiAppAdministrationService:

   ```
   $class=atg.endeca.assembler.MultiAppAdministrationService
   ```

   The MultiAppAdministrationService class is able to handle updates to multiple store factory instances.

4. Set the storeFactory property of the EndecaAdministrationService component to null:

   ```
   storeFactory^=/Constants.NULL
   ```

Creating FileStoreFactory Instances from Properties Files

To create FileStoreFactory instances from properties files:

1. For each EAC application, create a properties file for the corresponding FileStoreFactory component. Set the $class property to atg.endeca.assembler.content.ExtendedFileStoreFactory:

   ```
   $class=atg.endeca.assembler.content.ExtendedFileStoreFactory
   ```

2. Set the configurationPath property of each FileStoreFactory component to the file-system pathname of the directory to retrieve promoted content from. For example, the configurationPath property for the FileStoreFactory component associated with an EAC application named ATGde might be:

   ```
   configurationPath=\
   ToolsAndFrameworks/version/server/workspace/state/repository/ATGde
   ```

3. Set the `appName` property of each `FileStoreFactory` component to the name of the associated EAC application. For example:

   ```
   appName=ATGde
   ```

4. Modify the `/atg/endeca/assembler/AssemblerApplicationConfiguration` component in the local server configuration. Set the `useFileStoreFactory` property to `true` to automatically set a reference to the corresponding `FileStoreFactory` on the application's `WorkbenchContentSource`:

   ```
   useFileStoreFactory=true
   ```

5. Set the `applicationKeyToStoreFactory` property of the `AssemblerApplicationConfiguration` component to map application keys to the `FileStoreFactory` components you created. For example:

   ```
   applicationKeyToStoreFactory=\
   en=/atg/endeca/assembler/cartridge/manager/FileStoreFactory_en,\
   es=/atg/endeca/assembler/cartridge/manager/FileStoreFactory_es,\
   de=/atg/endeca/assembler/cartridge/manager/FileStoreFactory_de
   ```

6. Modify the `/atg/endeca/assembler/admin/EndecaAdministrationService` component in the local server configuration. Set the `$class` property to `atg.endeca.assembler.MultiAppAdministrationService`:

   ```
   $class=atg.endeca.assembler.MultiAppAdministrationService
   ```

   The `MultiAppAdministrationService` class is able to handle updates to multiple store factory instances.

7. Set the `storeFactory` property of the `EndecaAdministrationService` component to null:

   ```
   storeFactory^=/Constants.NULL
   ```

11 Record Filtering

Oracle Commerce Guided Search provides a mechanism for filtering the records returned by a query, based on the values of record properties. For example, for a multi-language application, you can use record filters to restrict the set of records returned to only those in the current language.

This chapter discusses Oracle Commerce classes you can use to build and apply Guided Search record filters. It includes these sections:

RecordFilterBuilder Interface and Implementing Classes

The Guided Search integration includes the `atg.endeca.assembler.navigation.filter.RecordFilterBuilder` interface. Classes that build Guided Search record filters implement this interface. The `RecordFilterBuilder` interface includes a `buildRecordFilter()` method that is responsible for building the actual record filter.

The Guided Search integration includes a number of classes that implement the `RecordFilterBuilder` interface, for example:

- `LanguageFilterBuilder`
- `CatalogFilterBuilder`
- `SiteFilterBuilder`
- `PriceListPairFilterBuilder`

The first three of these classes are described below. See the *Handling Price Lists* (page 125) chapter for information about the `PriceListPairFilterBuilder` class.

LanguageFilterBuilder

The `atg.endeca.assembler.navigation.filter.LanguageFilterBuilder` class constructs a filter that restricts the set of records returned to only those in the current language. `LanguageFilterBuilder`

determines the current customer's locale, and based on this, constructs a filter that excludes records that are not in the locale's language.

languagePropertyName

The name of the language property in Guided Search records to use for filtering. This is typically set to:

```
languagePropertyName=product.language
```

Note that the filter assumes that the value of this property was set in the records by the `LanguageNameAccessor` property accessor. See the LanguageNameAccessor (page 62) section for more information.

CatalogFilterBuilder

The `atg.commerce.endeca.assembler.navigation.filter.CatalogFilterBuilder` class constructs a filter that restricts the set of records returned to only those associated with the appropriate catalogs.

catalogTools

The component of class `atg.commerce.catalog.custom.CustomCatalogTools` used to determine the catalogs to include. By default, this property is set to:

```
catalogTools=/atg/commerce/catalog/CatalogTools
```

Note that a record associated with an excluded catalog might still be returned if it is also associated with an included catalog.

catalogIdPropertyName

The name of the catalog ID property in Guided Search records to use for filtering. This is typically set to:

```
catalogIdPropertyName=product.catalogId
```

SiteFilterBuilder

The `atg.endeca.assembler.navigation.filter.SiteFilterBuilder` class constructs a filter that restricts the set of records returned to only those associated with specified sites. For example, if there are three sites, site A, site B, and site C, the filter might specify that only records associated with site A or site C should be returned. (Note that a record associated with site B may still be returned if it is also associated with site A or site C.)

Note that the `SiteFilterBuilder` class offers similar functionality to the site filters you can create when you define a site for an EAC application. A site filter limits the records that are returned for a site to only those that are specified in the filter. This filtering is applied to all queries made to the site. By contrast, the `SiteFilterBuilder` class provides you with an additional measure of control over the site data that gets filtered out. For example, your application may need to support off-site spotlights or off-site searches. In this case, your application needs to query for results from the current site but also from other sites. To support

this scenario, your application can request all records from the MDEX and then filter out the records for the unneeded sites using the `SiteFilterBuilder` class.

A second distinction between a site filter and the filters provided by the `SiteFilterBuilder` class is that the site filter is an application filter, while the filters created by the `SiteFilterBuilder` class are added to the request as URL filters. Application filters happen implicitly, at an application level, every time a request is made, while URL filters are passed along with the request URL and are only executed for the current request. URL filters are combined with application filters before the results are returned from the MDEX.

Note: For more information on site filters and defining sites, see the *Oracle Commerce Guided Search Administrator's Guide*.

`SiteFilterBuilder` has a number of properties that it uses to determine which sites to include when it constructs the filter, described below.

siteIds

An array of the site IDs of the sites to include. Typically the value of this property is set through a form handler in a JSP, based on user interface elements, such as a set of checkboxes that the customer selects to indicate the sites to search.

siteScope

If `siteIds` is null, the `siteScope` property is used to determine the set of sites to include. It can be any of the following values:

- If `siteScope` is null or is set to `current`, only records associated with the current site are returned.
- If `siteScope` is set to `any`, all records that are associated with any site are returned.
- If `siteScope` is set to `all`, all records are returned, including ones not associated with any site.
- If `siteScope` is set to `none`, only records that are not associated with any site are returned.
- If `siteScope` is set to a shareable type ID, records are returned for any sites that are in a sharing group that shares the shareable type with the current site.

includeInactiveSites

If `true`, any inactive sites specified in the `siteIds` property or determined via the `siteScope` property are included. If `false` (the default), inactive sites are omitted.

includeDisabledSites

If `true`, any disabled sites specified in the `siteIds` property or determined via the `siteScope` property are included. If `false` (the default), disabled sites are omitted.

sitePropertyName

The name of the site ID property in Guided Search records to use for filtering. This is typically set to:

```
sitePropertyName=product.siteId
```

siteManager

The component of class `atg.multisite.SiteManager` used to determine which sites are enabled and active. This is typically set to `/atg/multisite/SiteManager`.

siteGroupManager

The component of class `atg.multisite.SiteGroupManager` used to determine which sites share with the current site the shareable type specified in the `siteScope` property. This is typically set to `/atg/multisite/SiteGroupManager`.

Enabling Record Filter Builder Components

The Guided Search integration includes a number of record filter builder components, such as:

```
/atg/endeca/assembler/cartridge/manager/filter/LanguageFilterBuilder
/atg/endeca/assembler/cartridge/manager/filter/CatalogFilterBuilder
/atg/endeca/assembler/cartridge/manager/filter/PriceListPairFilterBuilder
/atg/endeca/assembler/cartridge/manager/filter/SiteFilterBuilder
```

To enable a specific record filter builder component, you add it to the `recordFilterBuilders` property of the `/atg/endeca/assembler/cartridge/manager/NavigationStateBuilder` component. This property is an array of components of classes that implement the `RecordFilterBuilder` interface. For example:

```
recordFilterBuilders+=\
/atg/endeca/assembler/cartridge/manager/filter/PriceListPairFilterBuilder,
/atg/endeca/assembler/cartridge/manager/filter/CatalogFilterBuilder
```

DateRangeFilterBuilder

In addition to the `recordFilterBuilders` property, the `NavigationStateBuilder` component has a `rangeFilterBuilders` property that can be set to an array of components of classes that implement the `atg.endeca.assembler.navigation.filter.RangeFilterBuilder` interface. Classes that implement this interface construct range filters that are applied to results returned from MDEX queries.

Commerce Reference Store configures the `rangeFilterBuilders` property as follows:

```
rangeFilterBuilders+=\
  /atg/endeca/assembler/cartridge/manager/filter/DateRangeFilterBuilder
```

The `DateRangeFilterBuilder` component, which is of class `atg.endeca.assembler.navigation.filter.DateRangeFilterBuilder`, builds range filters based on the `startDate` and `endDate` properties of products and SKUs. It has `startDatePropertyNames` and `endDatePropertyNames` properties that are configured like this:

```
startDatePropertyNames=\
  product.startDate,\
  sku.startDate
```

```
endDatePropertyNames=\
  product.endDate,\
  sku.endDate
```

By default, `DateRangeFilterBuilder` uses only the day portion of the `startDate` and `endDate` timestamp values in constructing filters. The granularity of the filters is controlled by the `unitOfTime` property, which is set by default to `DAYS`. You can make the time period more granular by changing the value of this property to `HOURS` or `MINUTES`. Note, however, that this can degrade filtering performance, because query caching becomes less effective.

12 Handling Price Lists

If your application stores prices in product or SKU properties in the catalog repository, indexing price data and accessing it on site pages is handled much like it is for other properties, such as color or brand. If your application uses price lists, however, the prices are stored in a separate price list repository (`/atg/commerce/pricing/priceLists/PriceLists`), so additional mechanisms are required to index the price data and access it on sites.

This chapter describes how the Guided Search integration handles price data in price lists. It includes these sections:

For more information about price lists, see the *Core Commerce Programming Guide*.

Price List Pairs

A common configuration used on Commerce sites involves assigning a pair of price lists to each customer, with one price list containing the list prices for all SKUs in the catalog, and the other price list containing sale prices for the SKUs that are currently on sale (and empty values for SKUs that are not on sale). The customer profile's `priceList` property is set to the price list holding the list prices, and the profile's `salePriceList` property is set to the price list holding the sale prices.

When the application looks up the price of an individual SKU, the following logic is applied:

- If the price list specified in the `salePriceList` property has a price for the SKU, use that price.
- If the price list specified in the `salePriceList` property does *not* have a price for the SKU, use the price from the price list specified in the `priceList` property.

In other words, use the sale price if there is one, and if there isn't, use the list price. The resulting value is referred to as the *active price*.

The Guided Search integration includes classes that support this configuration. These classes assume each customer is assigned a price list pair. There may be only one price list pair that is assigned to all

customers, or there may be different price list pairs for each site in a multisite environment. For example, a multisite environment with multiple country stores might have a different price list pair for each country store, to handle different currency, catalogs, or pricing; the customer is assigned price lists based on the `defaultListPriceList` and `defaultSalePriceList` site properties for the current site.

When the Guided Search integration generates records for a given SKU, various classes are used to retrieve the data associated with specific price list pairs:

- The `PriceListPairVariantProducer` class produces a separate record for each price list pair.
- In each record, the `PriceListPairAccessor` class sets the value of the `product.priceListPair` property to the price list pair the record applies to.
- In each record, the `ActivePriceAccessor` class sets the value of the `sku.activePrice` property based on the price values in the price list pair.
- After the records are generated and indexed, the `PriceListPairFilterBuilder` is used during querying to construct a filter that returns only the records associated with the price list pair for the current customer.

Note that if your application uses only a single price list pair, the `PriceListPairVariantProducer` and the `PriceListPairFilterBuilder` are not needed and can be disabled. If your application assigns price lists based on criteria other than site, you may need to write alternative classes (e.g., a different variant producer) to implement price-handling logic.

Indexing Price List Data

This section describes the variant producer and property accessors used by the Guided Search integration to index price list data.

PriceListPairVariantProducer

The `atg.commerce.endeca.index.producer.PriceListPairVariantProducer` class produces a separate record for each price list pair. It obtains the price list pair for each site from the values of the `defaultListPriceList` and `defaultSalePriceList` properties of the site's `siteConfiguration` item.

The Guided Search integration includes a component of this class, `/atg/commerce/search/PriceListPairVariantProducer`, which is added to the `ProductCatalogOutputConfig.variantProducers` property by the `DCS.Endeca.Index` module. The following are key properties of `PriceListPairVariantProducer`.

priceListPairUniqueParamName

The name of the query parameter used to specify the price list pair in the URL identifying a product or SKU. By default, this property is set to `priceListPair`. For example, the value of the `product.url` property in a record might be:

```
atgrep:/ProductCatalog/sku/xsku2099?_product=xprod2099&catalog=
  homeStoreCatalog&locale=en_US&priceListPair=plist3080003_plist3080002
```

languagesPropertyName

The name of the property of the `siteConfiguration` item that specifies the languages for the site. By default, this property is null. If this property is set, `PriceListPairVariantProducer` uses the value of the specified `siteConfiguration` property to exclude unneeded variants.

For example, in Commerce Reference Store, the CRS Store US and CRS Home sites use the same price list pair (representing prices in dollars), while CRS Store Germany uses a separate price list pair (representing prices in euros). Commerce Reference Store sets the value of the `languagesPropertyName` property to `languages`. For the CRS Store US and CRS Home sites, the `siteConfiguration` item's `languages` property is set to `en, es`. So when generating the records for the price list pair used for CRS Store US and CRS Home, `PriceListPairVariantProducer` excludes the German language variants, since these price lists aren't used on any sites that support German.

Note that by default the Oracle Commerce Platform does not have a property for languages on the `siteConfiguration` item. If the `languagesPropertyName` is not set to a valid `siteConfiguration` property, records are generated for all possible combinations of language and price list pair.

PriceListPairAccessor

The `atg.endeca.index.accessor.PriceListPairAccessor` class sets the value of the `product.priceListPair` property of a record to the record's price list pair, which is obtained from the `PriceListPairVariantProducer`. The `product.priceListPair` property is specified in the `ProductCatalogOutputConfig` definition file like this:

```
<property name="priceListPair" is-dimension="true" type="string"
  property-accessor="/atg/endeca/index/accessor/PriceListPairAccessor"
  output-name="product.priceListPair" is-non-repository-property="true"/>
```

The resulting value has the following format:

salePriceList_listPriceList

For example:

```
<PROP NAME="product.priceListPair">
  <PVAL>
    plist3080003_plist3080002
  </PVAL>
</PROP>
```

ActivePriceAccessor

The `atg.endeca.index.accessor.ActivePriceAccessor` class sets the value of a record's `sku.activePrice` property based on the prices in the record's price list pair. The `sku.activePrice` property is specified in the `ProductCatalogOutputConfig` definition file like this:

```
<property name="price" type="float"
  property-accessor="/atg/commerce/endeca/index/accessor/ActivePriceAccessor"
```

```
  output-name="sku.activePrice" is-non-repository-property="true"/>
```

The actual calculation of the price is performed by a component of class `atg.commerce.endeca.index.ActivePriceCalculator`. This class looks up the prices in the price lists and uses the sale price if there is one, or uses the list price if there is no sale price. The `ActivePriceCalculator` component is specified through the `activePriceCalculator` property of the `ActivePriceAccessor` component. By default, this property is set to:

```
activePriceCalculator=/atg/commerce/endeca/index/ActivePriceCalculator
```

QueueingPropertiesChangeListener

The `IncrementalLoader` component monitors when changes are made to the values of properties specified in an `EndecaIndexingOutputConfig` component's definition file, and adds the modified items to the incremental item queue. So, for example, if the `color` property of a SKU in the product catalog is modified, that SKU is added to the incremental item queue for reindexing.

Prices in price lists, however, are not referenced directly by catalog items; instead, `price` items in the price list repository have references to the products or SKUs they apply to. So changes to `price` items do not automatically trigger reindexing of the corresponding product or SKU.

The `atg.repository.search.indexing.listener.QueueingPropertiesChangeListener` class addresses this issue by providing a mechanism for triggering reindexing of items in one repository based on changes to items in another repository. Oracle Commerce includes a component of this class, `/atg/commerce/search/PriceListPropertiesChangedListener`, that is configured to monitor changes to `price` items in the price list repository and add products and SKUs that they reference to the incremental item queue.

The following describes key properties of the `QueueingPropertiesChangeListener` class, and their default settings in the `PriceListPropertiesChangedListener` component.

incrementalLoader

The `IncrementalLoader` component to use. This component is responsible for queueing changes in the incremental item queue. This property is set to:

```
incrementalLoader=/atg/search/repository/IncrementalLoader
```

repository

The repository whose items are monitored for changes. This property is set to the price list repository:

```
repository=/atg/commerce/pricing/priceLists/PriceLists
```

itemDescriptorName

The item type of the repository items to monitor for changes. This property is set to:

```
itemDescriptorName=price
```

referencingPropertyToIndexedRepositoryAndType

A Map in which each key is the name of a property of the monitored item type, and the corresponding value is the Nucleus pathname and item type of the item descriptor in the indexed repository that the monitored properties reference. The values are of the form *repositoryPathName*:*itemDescriptorName*. For PriceListPropertiesChangedListener, the keys are productId and skuId properties of price items in the price list repository, and the values represent product and sku item descriptors in the catalog repository:

```
referencingPropertyToIndexedRepositoryAndType=\
  productId=/atg/commerce/catalog/ProductCatalog:product,\
  skuId=/atg/commerce/catalog/ProductCatalog:sku
```

monitoredPropertyNames

A list of properties of the item type specified by the itemDescriptorName property. If the value of monitoredPropertyNames is null (the default), all properties of the item type are monitored, and changes to the values of any of them triggers reindexing of the associated products or SKUs. If the value of monitoredPropertyNames is not null, only the listed properties are monitored.

Indexing Time-Based Prices

If your sites use time-based pricing, a price list may have multiple prices for a given product or SKU, with the active price differing depending on when the item is purchased. For example, a product may sell for $100.00 until December 25, and then sell for $50.00 after that.

To control which price values appear in indexed records, the ActivePriceCalculator class enables you to specify a time in the future to use as the effective time for determining prices. For instance, in the example mentioned above, if you run an indexing job on December 24 that uses an effective time of noon on December 26 for pricing, the record generated for the product will include a price value of $50.00.

You can specify the effective time for determining prices either as an explicit time or as an offset from the indexing start time. The /atg/search/repository/BulkLoaderand /atg/search/repository/IncrementalLoader components determine the time when an indexing job is started, and store this value in the atg.repository.search.indexing.Context object. The effective time for determining prices is calculated relative to this indexing start time. The indexing start time remains unchanged throughout the entire indexing job, to ensure that the prices in all of the generated records reflect the same effective time, regardless of how long the indexing job takes.

ActivePriceCalculator provides two properties for specifying the effective time for determining the prices in an indexing job:

indexingTimeOffsetInHours
Specifies the effective time as a number of hours after the time when the indexing job is started. For example, if the value of this property is 53.5, the effective time for pricing will be two days plus five and a half hours later than the start time of the indexing job.

indexingTimeCalendarString
Specifies the effective time for pricing as an explicit time (for example, January 6, 2056, at 3:00 am) or series of times (for example, the 1st and 15th of each month, at 5:00 pm). The value of this property is a string that uses the CalendarSchedule syntax described in the

Scheduler Services section of the *Platform Programming Guide*. For example, the following specifies the effective times as the 1st and 15th of each month, at 3:05 pm:

```
indexingTimeCalendarString=* 1,15 . . 15 5
```

Note that if `indexingTimeCalendarString` is set to a series of times, the effective time used for pricing for an individual indexing job is the first time in the series after the indexing start time. For example, if you use the `indexingTimeCalendarString` value above and you start an indexing job on the 8th of a month, the effective time will be the 15th of that month at 3:05 pm. If you start an indexing job on the 16th of a month, the effective time will be the 1st of the following month at 3:05 pm.

The `indexingTimeOffsetInHours` and `indexingTimeCalendarString` properties are mutually exclusive. If both properties are set, the value of `indexingTimeCalendarString` is used, and `indexingTimeOffsetInHours` is ignored. If neither property is set, the indexing start time is used as the effective time for determining prices.

For more information about time-based pricing, see the *Core Commerce Programming Guide*.

Filtering Records by Price List

The `atg.commerce.endeca.assembler.navigation.filter.PriceListPairFilterBuilder` class constructs a filter that restricts the set of records returned to only those associated with the price list pair used for the current customer. The Guided Search integration includes a component of this class, `/atg/endeca/assembler/cartridge/manager/filter/PriceListPairFilterBuilder`.

The name of the price list pair property in Guided Search records to use for filtering is specified through the `priceListPairPropertyName` property. This is typically set to:

```
priceListPairPropertyName=product.priceListPair
```

See the *Record Filtering* (page 119) chapter for more information about configuring and using record filters.

13 Dimension Value Caching

This chapter discusses dimension value caching, which the Guided Search integration uses to map GSA repository items to the Guided Search dimension values that represent them in the MDEX. The discussion in this chapter focuses on categories, but the feature is implemented in a general way so it can work with other repository items.

This chapter includes the following sections:

Mapping Categories to Dimension Values

A key aspect of the Guided Search integration involves treating product categories both as `category` items in the product catalog repository and as Guided Search dimension values. In some contexts categories are accessed via their category IDs, while in other contexts they are accessed via their dimension value IDs.

To manage the relationship between categories and dimension values, the Guided Search integration maintains a cache that maps each Oracle Commerce Platform category ID to the equivalent Guided Search `product.category` dimension value ID. The cache supports two-way lookup, so either value can be obtained if the other one is known. Commerce Reference Store makes extensive use of this cache in both directions. For example, to create a link from an Nucleus-driven page to an Assembler-driven category page, it can use the cache to obtain the dimension value ID from the category ID; to provide the current category context to an Oracle Commerce Platform scenario running in a cartridge on a category page, it can use the cache to find the category ID associated with the current category dimension value.

If your Guided Search environment includes multiple MDEX engines (for example, if you use a separate MDEX for each language), a separate dimension value cache is maintained for each MDEX. This avoids any collisions that might be caused by multiple MDEX engines using the same dimension value IDs.

DimensionValueCache and DimensionValueCacheObject

The dimension value cache is implemented by the `atg.commerce.endeca.cache.DimensionValueCache` class. This class uses objects of class `atg.commerce.endeca.cache.DimensionValueCacheObject` for storing cache entries. The cache is a `ConcurrentHashMap`, where each key is an category ID, and the corresponding map value is an instance of `DimensionValueCacheObject`.

The `DimensionValueCacheObject` class stores the following information about a dimension value:

- `dimvalId` – the dimension value ID for the category; e.g., `1245`
- `repositoryId` – the GSA repository ID for the category; e.g., `cat50087`
- `url` -- the Guided Search URL for the dimension value; e.g., `/browse?N=1245`
- `ancestorRepositoryIds` – a List of repository IDs for the category's ancestor categories; e.g., `cat10016,cat10014`

Note that a single key can be associated with multiple `DimensionValueCacheObject` instances, because a category can have multiple parent categories. Therefore when a `DimensionValueCache` is used to look up the dimension value for a specific repository ID, the results are returned as a `List` of `DimensionValueCacheObject` instances (although in many cases the `List` may have only one entry).

Managing the Cache

The `/atg/commerce/endeca/cache/DimensionValueCacheTools` component (of class `atg.commerce.endeca.cache.DimensionValueCacheTools`) provides methods used to access the caches. These include methods for:

- Retrieving a `List` of `DimensionValueCacheObject` instances that correspond to a particular category ID.
- Retrieving the `DimensionValueCacheObject` associated with a particular dimension value ID.
- Creating a new cache.
- Refreshing an existing cache.

In an environment with multiple MDEX engines, a single `DimensionValueCacheTools` component performs these operations on all caches. `DimensionValueCacheTools` has a `getCache()` method which retrieves the appropriate cache to access for a given request, based on the value returned by the `getCurrentApplicationKey()` method of the `AssemblerApplicationConfiguration` component.

Populating and Refreshing the Cache

The `/atg/endeca/assembler/cartridge/handler/DimensionValueCacheRefresh` component (of class `atg.commerce.endeca.assembler.cartridge.handler.DimensionValueCacheRefreshHandler`) is responsible for accessing the MDEX to populate the associated cache. If an attempt is made to access a cache that does not exist, the `DimensionValueCacheTools.createEmptyCache()` method is invoked to create an empty `DimensionValueCache`. The `DimensionValueCacheRefresh` component then accesses the MDEX to populate the cache. For each dimension value of the specified dimension, `DimensionValueCacheRefresh` creates a new `DimensionValueCacheObject` that stores the dimension value ID, the repository ID, the URL, and the repository IDs of the item's ancestor items.

If a cache lookup fails to find an entry, this may be because the cache is out of date. When this happens, `DimensionValueCacheRefresh` attempts to refresh the cache by recreating all of the entries. However, to prevent unnecessary refreshes (such as when an entry is not found because it has not been indexed, which means a refresh will not fix the failed lookup), the cache is not refreshed if any of the following conditions exist:

- The number of seconds since the last refresh is less than the value of the `DimensionValueCacheTools.minimumCacheRefreshIntervalSecs` property (default value is 600).
- A refresh is already in progress.

- The MDEX has not been updated since the last time the cache was refreshed.

To ensure that the caches do not become stale, `DimensionValueCacheTools` has a property, `checkMDEXUpdatedEveryNHours`, for specifying a time interval in hours. (The default value is 24.) When an attempt is made to access a cache, if the number of hours since the last refresh of the cache is greater than this value, `DimensionValueCacheRefresh` attempts to refresh the cache. However, the cache is not refreshed if any of the conditions listed above exist.

Key properties of the `DimensionValueCacheRefresh` component include:

dimensionName
The name of the dimension in the MDEX. Set by default to `product.category`.

repositoryIdProperty
The name of the property in the MDEX that represents the repository ID of the category. Set by default to `category.repositoryId`.

dimensionValueCacheTools
The `DimensionValueCacheTools` component used to access the cache. Set by default to `/atg/commerce/endeca/cache/DimensionValueCacheTools`.

navigationState
The component representing the Guided Search `NavigationState` to use to access the MDEX. By default, this is set to the `/atg/endeca/assembler/cartridge/manager/UnfilteredNavigationState` component, which creates a `NavigationState` object without any refinements or filters applied. This is done so that the set of dimension values returned is not restricted based on the navigational context.

Populating the DimensionValueCacheObject.url Property

To populate the `url` property of a `DimensionValueCacheObject` with an appropriate link, the `DimensionValueCacheTools` component invokes the Assembler. These links must always begin with the `/browse` content path and, as such, they require the `DimensionValueCacheTools` component to perform an extra step. Specifically, before the invocation, the `DimensionValueCacheTools` component modifies the request it passes to the Assembler to add a new request attribute, `DefaultActionPathProvider.ALWAYS_USE_DEFAULT_NAVIGATION_CONTENT_PATH`, and sets it to `true`. This request attribute forces the Assembler to use the `DefaultActionPathProvider` component's `defaultExperienceManagerNavigationActionPath` property when setting the content path for the `url`, instead of going through the normal `DefaultActionPathProvider` calculations to derive the content path. Because this property is set to `/browse` by default, forcing the Assembler to use it ensures that the links returned to the `DimensionValueCacheTools` component are correct. The `DimensionValueCacheTools` object subsequently removes the additional request attribute after the links are retrieved, so any other invocations of the Assembler proceed as normal.

Note: For more details on Assembler invocation and the `DefaultActionPathProvider` component, see the *Query Integration* (page 85) chapter.

DimensionValueCacheDroplet

On a JSP page, you can use the `/atg/commerce/endeca/cache/DimensionValueCacheDroplet` component (of class `atg.commerce.endeca.cache.DimensionValueCacheDroplet`) to obtain the dimension value associated with a specific category. This servlet bean takes the following input parameters:

repositoryId
The repository ID of the category to retrieve the corresponding `DimensionValueCacheObject` for.

ancestors
A list of the repository IDs of the category's ancestor categories, delimited by colons. This value helps determine the correct `DimensionValueCacheObject` to retrieve for a category that has more than one path in the catalog hierarchy.

`DimensionValueCacheDroplet` returns the `DimensionValueCacheObject` entry that matches these parameters. For example:

```
<dsp:droplet name="DimensionValueCacheDroplet">
  <dsp:param name="repositoryId" value="${categoryId}"/>
  <dsp:param name="ancestors" value="${topLevelCategoryId}"/>
  <dsp:oparam name="output">
    <dsp:getvalueof var="categoryCacheEntry" param="dimensionValueCacheEntry" />
  </dsp:oparam>
</dsp:droplet>
```

The `url` property of the `DimensionValueCacheObject` can be used to render a link to an Assembler-driven category page. For example:

```
<dsp:a page="${categoryCacheEntry.url}">
  <dsp:valueof value="${categoryDisplayName}">
    <fmt:message key="common.categoryNameDefault" />
  </dsp:valueof>
</dsp:a>
```

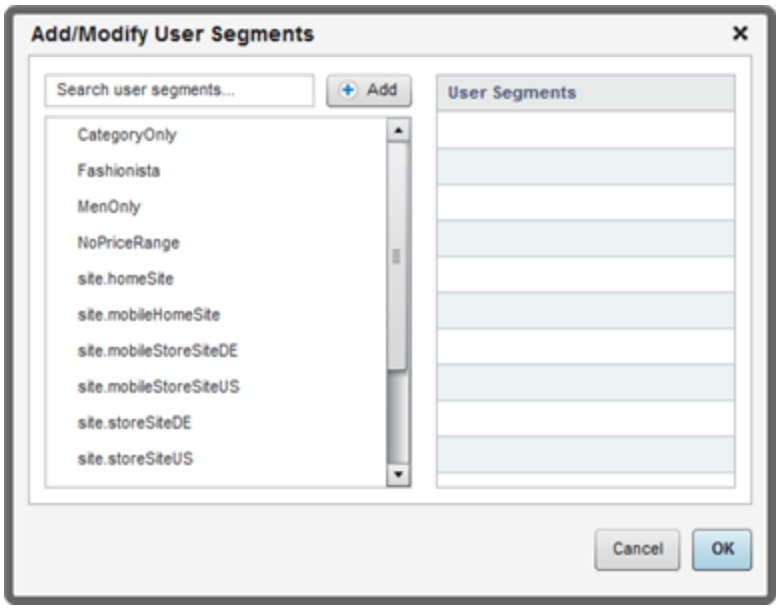
14 User Segment Sharing

This chapter discusses the user segment sharing feature that allows a content administrator to choose a user segment that has been defined in the Business Control Center as a trigger for a cartridge in Experience Manager.

This chapter includes the following sections:

About User Segment Sharing

The user segment sharing feature automatically populates the Add/Modify User Segments dialog box in Experience Manager with any user segments that have been defined in the Business Control Center. This is the dialog box that is used to define triggers for a cartridge, an example of which is shown below:



User segments can be created in both the Business Control Center and in the Workbench, and Experience Manager will show both in the Add/Modify User Segments dialog box. In general, to reduce the possibility for duplication, user segments should be defined in the Business Control Center and then automatically populated in the Add/Modify User Segments dialog box.

User segment sharing is a one-way relationship. When configured to do so, user segments created in the Business Control Center are shared with Experience Manager. However, user segments created in Workbench are not shared with the Business Control Center.

Configuring User Segment Sharing

When configuring the user segment sharing feature, you must specify an Oracle Commerce Platform server to act as the user segment server. This server responds to Workbench requests for the list of user segments defined in the Business Control Center. In general, Oracle recommends using the Content Administration server as the user segment server.

If your environment does not have a Content Administration server, you can use the Production server instead but additional configuration is required. Note that using the Production server as the user segment server is less than ideal because typically the most up-to-date user segment data resides on the Content Administration server, and you want a merchandiser to have access to that up-to-date data in Experience Manager. Also, calls made to a live, customer-accessible Production server will typically have to go through a firewall. For these reasons, you should only use the Production server as your user segment server in development environments that do not use a Content Administration server.

Note: The Content Administration server is also referred to as the Publishing server in CIM.

To query for Business Control Center user segments, the Workbench sends a call to the user segment server using the REST Service provided by the REST module. This REST call must be secure to prevent unwanted access to the user segment data. By default, user segment security is enabled via the `RequestCredentialAccessController` component that is included with the REST module. However, you must add security credentials to both the user segment server and the Workbench to complete the security configuration. Also, you must configure each EAC application with the correct URL for the REST request. To request user segments, the Workbench sends its security credential in a header with the EAC application's REST request to the user segment server. On the user segment server side, the `RequestCredentialAccessController` component compares the security credential in the request to the security credentials configured on the user segment server. If a match is found, the request is allowed. If not, it is denied.

Additional Configuration Required for the Production Server

If your environment does not have a Content Administration server, you can use the Production server as the user segment server instead. Note that this configuration is recommended for development environments only (see Configuring User Segment Sharing (page 136) for more information on why). To use the Production server as the user segment server, you must make sure it includes the REST module. You can do this in CIM by choosing the Oracle Commerce REST - RESTful Web Services option in the Product Selection menu. Alternatively, you can include the REST module when running the `runAssembler` command to assemble your Production server's EAR file:

```
runAssembler earfilename –m REST other-modules
```

In addition to including the REST module, you must also add the GetAllProfileGroups URL to the /atg/rest/registry/ActorChainRestRegistry.registeredUrls property. To do this, create an /atg/rest/registry/ActorChainRestRegistry.properties file in the Production server's localconfig directory with the following property, then restart the Production server:

```
registeredUrls+=/atg/userprofiling/ProfileGroupsActor/getAllProfileGroups
```

About the RequestCredentialAccessController Component

In order to make the REST calls for user segments secure, the REST module includes a component, /atg/rest/security/RequestCredentialAccessController, that enables and enforces access control for these calls. Out of the box, the RequestCredentialAccessController component's enable property is set to true. If you need to disable security for the REST calls, you can set this value to false, although this is not a configuration that Oracle recommends.

To determine if a user segment request should be fulfilled, the RequestCredentialAccessController component compares the security credential passed in an HTTP header of the request with the credentials stored in a credential store map. If a matching credential exists in the credential store map, the request is fulfilled. If no match exists, access to the user segment data is denied. To support this functionality, the RequestCredentialAccessController component includes the properties listed below, in addition to the enabled property. Note that these properties must not be changed or user segment security will cease to work:

- credentialStoreMap: The credential store map under which valid REST security credentials are stored. User segment server requests must include a credential that matches a credential stored in this map in order to be fulfilled. The default value for this property is requestCredentialMap and must not be changed.
- fieldName: The name of the HTTP header that contains the credential for user segment server REST requests. This setting defaults to Request-Credential, which is the field that the Workbench uses to pass the credential header, and it must not be changed.

Managing Credentials

In order for user segment security to work, you must add credentials in two places:

- To the credentialStoreMap. The RequestCredentialAccessController component references this map when determining if a request includes a valid credential.
- To the Workbench so that it can pass a valid credential along with the user segment request.

Modifications to REST security credentials stored in the credentialStoreMap are effective immediately after they are saved. Modifications to the Workbench security credential require a restart before those changes become available for use.

Managing Credentials in the credentialStoreMap

You can add a credential to the credentialStoreMap using either CIM or Dynamo Server Admin. Follow the instructions below to add a security credential to the credentialStoreMap using CIM.

1. In the CIM MAIN MENU, select [2] Configure OPSS Security.
2. In the SECURITY DEPLOYMENT MENU, choose [1] Enter the location to deploy OPSS files.
3. Press Enter to accept the default location for OPSS files.
4. In the SECURITY DEPLOYMENT MENU, choose [2] Enter the security credential for REST Services.

5. Enter the new credential at the prompt. The credential can be any text, similar to a password, however it should correspond to your organization's OPSS security platform requirements.

6. Re-enter the credential to confirm it.

7. In the SECURITY DEPLOYMENT MENU, choose [3] Deploy configuration files.

8. In the COPY CREDENTIALS TO SHARED DIRECTORY menu, choose [D] Deploy to `/<ATG11dir>/home/../home/security`.

9. In the VERIFY WHETHER TO OVERWRITE CURRENT DIRECTORY CONTENTS menu, choose [D] Deploy OPSS configuration files.

10. In the SECURITY DEPLOYMENT MENU, choose [D] Done.

Alternatively, you can add or delete security credentials using Dynamo Server Admin.

To enter security credentials in Dynamo Server Admin:

1. In a browser, navigate to the instance of Dynamo Server Admin that is running on the user segment server:

   `http://<user_segment_server_host>:<user_segment_server_HTTP_port>/dyn/admin`

2. In the authentication dialog box, enter the Dynamo Server Admin username and password click OK.

3. **(WebLogic only)** Depending on how you configured your environment, WebLogic may require an additional login for the WebLogic server. If necessary, enter your WebLogic username and password, and then click OK.

   You see the Administration home page.

4. Click the Component Browser link.

5. Navigate to `/atg/dynamo/security/opss/csf/CredentialStoreManager`.

6. From the Action drop-down menu, choose Create Generic Credential and then click Select.

7. In the Map Name field, enter `requestCredentialMap`.

8. Enter a key name in the Credential Key Name field, for example, `key1`. Use a unique key name to enter a new credential. Use an existing key name to replace the credential for that key name.

9. Enter the new credential in the Enter Credential area. The credential can be any text, similar to a password, however it should correspond to your organization's OPSS security platform requirements.

10. Click Submit Credentials.

To delete an existing REST security credential:

1. In a browser, navigate to the instance of Dynamo Server Admin that is running on the user segment server. See the previous section for detailed instructions on how to do this.

2. Click the Component Browser link.

3. Navigate to `/atg/dynamo/security/opss/csf/CredentialStoreManager`.

4. From the Action drop-down menu, choose Delete Credential and then click Select.

5. Select the credential you want to delete.

6. Click Delete Credential.

Managing Credentials in the Workbench

To manage credentials in the Workbench, you use the `manage_credentials` script in the `/credential_store/bin` directory under `ToolsAndFrameworks`.

To add a credential to the Workbench:

1. In a UNIX shell or command prompt, navigate to the `ENDECA_TOOLS_ROOT/credential_store/bin` directory, for example, `/usr/local/endeca/ToolsAndFrameworks/`*`version`*`/credential_store/bin` or `C:\Endeca\ToolsAndFrameworks\`*`version`*`\credential_store\bin`.
2. Enter one of the following commands.

   On UNIX, enter:

   ```
   ./manage_credentials.sh add --user admin --config [path to jps-config.xml] --type
   generic --mapName restService --key clientCredential
   ```

   For example:

   ```
   ./manage_credentials.sh add --user admin --config $ENDECA_TOOLS_ROOT/server/
   workspace/credential_store/jps-config.xml --type generic --mapName restService --key
   clientCredential
   ```

   On Windows, enter:

   ```
   manage_credentials.bat add --user admin --config [path to jps-config.xml] --type
   generic --mapName restService --key clientCredential
   ```

   For example:

   ```
   manage_credentials.bat add --user admin --config %ENDECA_TOOLS_ROOT%\server
   \workspace\credential_store\jps-config.xml --type generic --mapName restService --
   key clientCredential
   ```

3. Enter the new credential at the prompt.
4. Re-enter the credential to confirm the addition.
5. Follow the instructions below to restart the ToolsAndFrameworks service.

To restart the ToolsAndFrameworks service:

1. In a UNIX shell or command prompt, navigate to the `ENDECA_TOOLS_ROOT/server/bin` directory, for example, `/usr/local/endeca/ToolsAndFrameworks/`*`version`*`/server/bin` or `C:\Endeca\ToolsAndFrameworks\`*`version`*`\server\bin`.
2. Execute the `shutdown` script.

   On UNIX, enter:

   ```
   ./shutdown.sh
   ```

   On Windows, enter:

   ```
   shutdown.bat
   ```

3. Execute the `startup` script.

   On UNIX, enter:

```
./startup.sh
```

On Windows, enter:

```
startup.bat
```

To delete a credential:

1. In a UNIX shell or command prompt, navigate to the `ENDECA_TOOLS_ROOT/credential_store/bin` directory, for example, `/usr/local/endeca/ToolsAndFrameworks/`*`version`*`/credential_store/bin` or `C:\Endeca\ToolsAndFrameworks\`*`version`*`\credential_store\bin`.
2. Enter one of the following commands.

   On UNIX, enter:

   ```
   ./manage_credentials delete --user admin --config [path to jps-config.xml] --mapName
   restService --key clientCredential
   ```

   For example:

   ```
   ./manage_credentials delete --user admin --config $ENDECA_TOOLS_ROOT/server/
   workspace/credential_store/jps-config.xml --mapName restService --key
   clientCredential
   ```

   On Windows, enter:

   ```
   manage_credentials.bat delete --user admin --config [path to jps-config.xml] --
   mapName restService --key clientCredential
   ```

   For example:

   ```
   manage_credentials.bat delete --user admin --config %ENDECA_TOOLS_ROOT%
   \server\workspace\credential_store\jps-config.xml --mapName restService --key
   clientCredential
   ```

   You are notified when the credential is successfully deleted.

Configuring the EAC Application

For the EAC application, you must specify the hostname and port of the URL that is used to connect to the REST Service (the remainder of the URL after the port is well known and cannot be changed). You can create this configuration in either of the following ways:

- By directly modifying the `atgServices.json` file for the EAC application that needs access to Business Control Center user segment data. With this approach, you are making a one-time change that will have to be repeated if you re-create the EAC application in the future.
- By modifying the deployment template used to create your EAC application to add the necessary hostname and port prompts and then store the responses in the application's configuration. With this approach, every EAC application that is created using the modified deployment template will include the proper user segment sharing configuration.

Modifying the EAC Application Directly

To specify the REST Service hostname and port directly in the EAC application:

1. Navigate to the `config/ifcr/configuration/tools/xmgr` directory of your deployed EAC application on disk, for example, `/usr/local/endeca/Apps/CRS/config/ifcr/configuration/tools/xmgr`.

2. Open `atgServices.json` in a text editor.

3. Set the `profileGroupsConnectionUrl` property to the JSON response for the `getAllProfileGroups` operation on the Content Administration server.

   For example:

   ```
   {
   profileGroupsConnectionUrl:"http://<Content Administration
   host>:<Content Administration server HTTP port>
   /rest/model/atg/userprofiling/ProfileGroupsActor/
   getAllProfileGroups?atg-rest-output=json&atg-preview=false"
   }
   ```

4. Save and close the file.

5. Navigate to the `control` directory of your deployed EAC application on disk, for example, `/usr/local/endeca/Apps/CRS/control`.

6. Run the `set_editors_config` script, for example:

   ```
   ./set_editors_config.sh
   ```

Modifying the Deployment Template

To modify the deployment template:

1. Locate the `deploy.xml` file you will use to configure your EAC application.

2. Add two `<token>` elements to the `<custom-tokens>` element with the following configuration:

   ```
   <custom-tokens>
   <!-- Other tokens -->
   <token name="USER_SEGMENTS_HOST">
   <prompt-question>Enter the hostname of user segment server.
   This server will respond to user segment requests from the
   Workbench. [Default:localhost]</prompt-question>
   <install-config-option>userSegmentsHost</install-config-option>
   <default-value>localhost</default-value>
   </token>
   <token name="USER_SEGMENTS_PORT">
   <prompt-question>Enter the HTTP port of the user segment server.
   This server will respond to user segment requests from the
   Workbench.</prompt-question>
   <install-config-option>userSegmentsPort</install-config-option>
   </token>
   </custom-tokens>
   ```

3. In the same directory as the `deploy.xml` file, add an `/ifcr/configuration/tools/xmgr/atgServices.json` file with the following content:

   ```
   {
   profileGroupsConnectionUrl=
   "http://@@USER_SEGMENTS_HOST@@:@@USER_SEGMENTS_PORT@@/
   ```

```
rest/model/atg/userprofiling/ProfileGroupsActor/getAllProfileGroups?
atg-rest-output=json&atg-preview=false"
}
```

4. Save and close the file.

Note about Configuring Commerce Reference Store

If you are installing and configuring Commerce Reference Store, the URL connection prompts are incorporated into the CIM script that Commerce Reference Store uses, via the following two options in the DEPLOY CRS EAC APP menu:

```
Enter the hostname of the user segment server. Oracle recommends using the
Publishing server for this purpose. If your environment does not have a Publishing
server, enter the Production server host name and refer to the Guided Search
Integration Guide for additional configuration requirements. [[localhost]] >

Enter the hostname of the user segment server. Oracle recommends using the
Publishing server for this purpose. If your environment does not have a Publishing
server, enter the Production server host name and refer to the Guided Search
Integration Guide for additional configuration requirements. [[7003]]>
```

When CIM executes the deployment template, it uses the values specified for these two menu options to populate the deployment template prompts.

Avoiding Duplicate User Segment Names in the Business Control Center

The Business Control Center allows you to create folders for user segments. The user segment sharing feature in Experience Manager, however, does not differentiate by folder, so the following two segments would both appear as YoungMales in the Add/Modify User Segments dialog box:

`/RepositoryGroups/YoungMales`

`/RepositoryGroups/mySegments/YoungMales`

For this reason, it is important to use a unique name for every user segment you create in the Business Control Center, regardless of its location in the user segment folder structure.

Renaming a User Segment in the Business Control Center

If the name of a user segment is changed in the Business Control Center, it is treated in Experience Manager as if a new segment with a new name has been added. For example, assume you have created a user segment in the Business Control Center called `MySegment` and, in Experience Manager, you have configured a cartridge

called `MyCartridge` to be triggered when `MySegment` is part of the current query. If you subsequently change the name of `MySegment` to `RenamedSegment`, when you return to Experience Manager, the trigger for `MyCartridge` remains `MySegment` but this cartridge will never be triggered because `MySegment` no longer exists.

Also, if you look at the Add/Modify User Segments dialog box for `MyCartridge`, it will show `RenamedSegment` in the left hand pane, where segments that are available for selection are displayed, but it will continue to show `MySegment` in the right pane, where the currently selected segments are displayed. To reconfigure `MyCartridge` to use `RenamedSegment` as a trigger, you must remove `MySegment` from the currently selected segments list and add `RenamedSegment` to the list of triggers.

15 Using Sites and Site Groups as Content Item Triggers

In Experience Manager, content folders, unlike pages, are not site-specific, meaning a content folder's items may be shared by multiple sites. For organizational reasons, you may want to group a set of content items under the same content folder but have some of those items only trigger for particular sites or site groups. For example, you may want to organize all of your promotional banner content items in the same content folder, keeping them together and easy to locate, but one of those promotional banners would only be triggered when a particular site is the current site. For this site-specific promotional banner, you would specify the site as a trigger when you add the banner's content item to the content folder in Experience Manager. Note that a content item can have one or more sites or site groups as a trigger.

Enabling the use of sites and site groups as triggers in Experience Manager requires some additional configuration on the part of the Workbench administrator. Specifically, to incorporate sites and site groups in content item triggers, the following happens:

- An administrator or business user manually adds a set of user segments to Workbench that correspond to the sites and site groups that have been defined in Site Administration. This allows the sites and site groups to be used in the user segment triggers configured by business users.
- Experience Manager business users configure triggering rules for the content items using the manually added user segments.
- Every time the Core Platform calls the Assembler, it passes site context information (the current site and its site groups) for the current request and shopper.
- The passed information allows the Assembler to return the correct content for the request, based on the configuration set in Experience Manager.

The following sections provide more detail on these general steps.

Adding Sites and Site Groups to Experience Manager

An administrator or business user must manually add user segments to Workbench that correspond to the sites and site groups that are defined in Site Administration before they can be referred to in triggering rules. When adding the new Workbench user segments, business users must use the correct syntax. Site and site group IDs are unique within a type but can, in theory, collide across types. Because these names are passed as a single list to Workbench, a prefix scheme is used to ensure a unique identifier for each site and site group that is passed. For more details on these prefixes, see the next section, Constructing the Segment List (page 146).

Constructing the Segment List

The DAF module is responsible for adding sites and site groups to the user segment list that is sent to the Assembler. The DAF module includes the request-scoped `/atg/endeca/assembler/cartridge/manager/user/LiveUserState` component. This component is of class `atg.endeca.assembler.navigation.LiveUserState`. The `LiveUserState` class is an extension of the `com.endeca.infront.navigation.UserState` class that overrides the `getUserSegments()` method with a call to a `LiveUserState.computeSegments()` method that computes the list of sites and site groups to be added to the segment list that is sent to the Assembler. Segment names are added to the `userSegments` property via the `addUserSegments()` method.

The DAF module contains the following configuration for the `LiveUserState` component:

```
sitePrefix=site
siteGroupPrefix=sitegroup
prefixDelimiter=.
```

This configuration specifies that, when the `LiveUserState` component adds a site to the segment list, it prefixes the site's ID (as defined in the Site Manager UI) with the word `site` and a period, for example:

```
site.MySite
```

A similar situation exists for site groups. Site group IDs are prefixed with `sitegroup` and a period, for example:

```
sitegroup.MySiteGroup
```

It is important to remember that any segments added in the Workbench must also follow these naming conventions.

16 Commerce Single Sign-On

The Oracle Commerce Business Control Center and the Oracle Commerce Workbench are both used for managing content and its presentation on sites built with the integration of the Oracle Commerce Platform and Oracle Commerce Guided Search. To simplify working in both environments, the Guided Search integration includes the Commerce Single Sign-On (SSO) feature. Commerce Single Sign-On ensures that when a user logs into either the Business Control Center or the Workbench, that user is automatically also logged into the other environment. Logging out of one environment automatically logs the user out of the other one as well.

Commerce SSO consists of three pieces:

- An Oracle Commerce Platform SSO server instance that is responsible for managing the SSO sessions shared by the Business Control Center and the Workbench.
- An Oracle Commerce Platform plug-in that is responsible for communication between the Business Control Center and the SSO server.
- A Oracle Commerce Guided Search plug-in that is responsible for communication between the Workbench and the SSO server.

The Guided Search plug-in is discussed in the *Oracle Commerce Guided Search Administrator's Guide*. This chapter discusses the SSO server and the Oracle Commerce Platform plug-in, and includes the following sections:

Commerce Single Sign-On Server

Commerce SSO is managed by a dedicated Oracle Commerce Platform server instance. When you set up your environment in CIM, it gives you the option of setting up this server. The server includes the SSO module and the `DPS.InternalUsers` module (which the SSO module has a dependency on), and uses the same datasources as the ATG Content Administration server, so it can access the Oracle Commerce Platform internal profile repository.

When an unauthenticated user attempts to access the Business Control Center or the Workbench, he or she is redirected to the SSO server's login page. The login is authenticated against either the internal profile repository

or the LDAP server, depending on which configuration is used. If the login succeeds, the requested application is displayed.

The SSO module includes a web application that manages the single-sign on process. The application, whose context root is `sso`, provides six main functions that can be accessed via plug-ins by client applications: login, validate, keep alive, query, control, and logout.

To perform these tasks, the Commerce SSO makes use of *ticket granting tickets* and *service tickets*. A ticket granting ticket is like a global flag that indicates the user has been successfully authenticated. When a user is authenticated successfully, a service ticket is issued to the user. The service ticket is a short-term object that is used to perform validation. The first time the user attempts to access a URL, the service ticket is passed to the SSO server along with the URL to validate that the user is permitted to access the URL. The SSO server responds either "yes" or "no" to the request based on the status of the ticket.

The SSO application adds the `/atg/sso/servlet/SSODispatcherServlet` component, of class `atg.servlet.pipeline.ServletPathDispatcherPipelineServlet`, to the Oracle Commerce Platform request-handling pipeline on the SSO server. This servlet dispatches requests to other servlets that provide the SSO server functions. The servlet that `SSODispatcherServlet` dispatches the request to depends on the servlet path of the request:

- `/login` – Dispatches the request to the `/atg/sso/servlet/LoginServlet` component, of class `atg.sso.servlet.LoginServlet`. This servlet manages the process of authenticating the user and issuing a service ticket.
- `/validate` -- Dispatches the request to the `/atg/sso/servlet/ValidateServlet` component, of class `atg.sso.servlet.ValidateServlet`. This servlet manages the process of validating requests based on the status of service tickets.
- `/keepAlive` -- Dispatches the request to the `/atg/sso/servlet/KeepAliveServlet` component, of class `atg.sso.servlet.KeepAliveServlet`. This servlet ensures that an SSO session remains active as long as there is activity in either the Business Control Center or the Workbench. For example, if the user logs into Commerce SSO and accesses the Workbench for several hours without accessing the Business Control Center, the keep alive function ensures that subsequent attempts to access the Business Control Center do not require logging in again.
- `/query` -- Dispatches the request to the `/atg/sso/servlet/QueryServlet` component, of class `atg.sso.servlet.QueryServlet`. This servlet is responsible for issuing RQL queries against the internal profile repository. This function is accessed only by the Guided Search plug-in.
- `/control` – Dispatches the request to the `/atg/sso/servlet/ControlServlet` component, of class `atg.sso.servlet.ControlServlet`. This servlet handles configuration of the client logout URL. This function is accessed only by the Guided Search plug-in.
- `/logout` – Dispatches the request to the `/atg/sso/servlet/LogoutServlet` component, of class `atg.sso.servlet.LogoutServlet`. This servlet manages the process of deleting any tickets associated with the session and then redirecting to the login page.

Oracle Commerce Platform Plug-In

The Oracle Commerce Platform plug-in consists of extensions to the Business Control Center that provide access to four of the SSO server functions discussed above: login, validate, keep alive, and logout. (The other two functions, control and query, are accessed only by the Guided Search plug-in.) These extensions

include the /atg/dynamo/servlet/dafpipeline/LightweightSSOServlet component (of class atg.userprofiling.sso.LightweightSSOServlet), which is inserted in the request-handling pipeline on the ATG Content Administration server. This component manages much of the communication between the ATG Content Administration server and the SSO server.

CIM includes options for configuring the SSO server instance. In addition, the CIM Commerce Add-Ons screen has a Single Sign-On option for configuring the ATG Content Administration server with information that enables it to communicate with the SSO server, such as the SSO server's host name and port number.

The login, validate, keep alive, and logout functions are discussed below.

Login

When a user who is not logged in attempts to access the Business Control Center, the user is redirected to the SSO login page, and is prompted to authenticate using Commerce SSO. If the authentication succeeds on the SSO server, the user is then redirected to the ATG Content Administration server, which retrieves the corresponding user profile from the internal profile repository and associates the current session with the profile. If authentication fails, the user remains at the Commerce SSO login page.

The /atg/dynamo/servlet/dafpipeline/AccessControlServlet and /atg/web/assetmanager/userprofiling/NonTransientAccessController components are reconfigured by the plug-in to delegate control of the Business Control Center login process to Commerce SSO. The NonTransientAccessController component is responsible for redirecting the user to the SSO server login URL, which it constructs by invoking methods on the /atg/userprofiling/commercesso/CommerceSSOTools component.

Note: To enable redirection of requests from the ATG Content Administration server to the Commerce SSO server, add the hostname of the SSO server to the allowedHostNames property of the /atg/dynamo/servlet/pipeline/RedirectURLValidator component on the ATG Content Administration server. For example:

```
allowedHostNames+=ssohost.example.com
```

Validation

When a user first attempts to access the Business Control Center, a validation request is sent to the Commerce SSO server. This request contains both the ticket parameter and a service parameter containing the value of the original URL being requested. (The request also contains a logout parameter to enable SSO logout, as discussed below.) If the SSO server indicates it is valid for the user associated with the ticket to access the URL in the service parameter, the ProfileRequestServlet loads the associated user profile.

Keep Alive

The LightweightSSOServlet component handles keep-alive calls to the SSO server. After a user's login is authenticated, LightweightSSOServlet periodically polls the KeepAliveServlet on the SSO server, and continues polling as long as it receives a "yes" reply each time. If a "no" reply is received, the Core Commerce session is terminated. If the SSO server cannot be reached, additional attempts are made to contact the server before ending the Core Commerce session.

The polling behavior of the LightweightSSOServlet component is specified through the following properties:

keepAlivePollingFrequency
The amount of time in minutes between keep-alive calls. Default is 10.

keepAliveAttempts
The number of keep-alive calls to make, if there is no response from the Commerce SSO server, before ending the Core Commerce session. Default is 3.

Logout

The way logout is handled depends on whether it is initiated from the Business Control Center or the Workbench.

If a user logs out from the Business Control Center, the standard Core Commerce logout process is invoked, and the current Core Commerce session is terminated. The user request is then redirected to the Commerce SSO logout URL, so that the Commerce SSO session is also terminated. To accomplish this, the `InternalProfileFormHandler.logoutSuccessURL` and `ControlCenterService.logoutSuccessURL` properties are configured to hold the Commerce SSO logout URL. If the SSO session includes a Workbench session, the Commerce SSO server terminates the Workbench session by sending a callback URL.

If a user logs out of the Workbench, the Commerce SSO session is terminated. The Core Commerce logout process must then be triggered as well. As part of the initial request to validate the service ticket and request URL (see above), the Commerce SSO Server is sent a logout parameter populated with a logout callback URL. This parameter is used by the SSO Server to initiate a logout from the Core Commerce session after the SSO session has been terminated by the user logging out of the Workbench. The `LightweightSSOServlet` detects such logout requests and invokes the `logoutUser()` method of the `/atg/userprofiling/ProfileServices` component to handle logging out the user.

Maintaining User Accounts

User accounts and authentication credentials for Commerce SSO can be implemented in either of two ways:

- The Oracle Commerce Platform internal profile repository (used for storing Business Control Center accounts) can be implemented as a standard SQL repository. In this case, the authentication information for Commerce SSO is stored in the repository database.
- The internal profile repository can be implemented as a composite repository, where the authentication information is stored in an LDAP directory.

The next section discusses using an LDAP server for authentication.

LDAP Authentication

Using LDAP authentication allows Commerce SSO to make use of an existing corporate LDAP server to control access to the Business Control Center and the Workbench. In the Oracle Commerce Platform, the internal profile repository is configured as a composite repository that accesses authentication data stored in an LDAP directory. This configuration is described below.

Setting up a Composite Profile Repository

A composite profile repository is a variant of a standard profile repository in which some user data is stored in a database and accessed through a SQL repository, while authentication information is stored in an LDAP directory and accessed through an LDAP repository. The composite repository provides a unified view of all of the data, regardless of its source. See the *Personalization Programming Guide* for information about composite profile repositories. For more general information about composite repositories, see the *Repository Guide*.

You can use CIM to set up the internal profile repository as a composite repository. As mentioned above, CIM includes options for configuring the SSO server instance, and the CIM Commerce Add-Ons screen has a Single Sign-On selection for configuring SSO. If you select the Single Sign-On option, CIM will also display a Commerce SSO Add-Ons screen which has a selection for configuring LDAP authentication settings.

If you select LDAP authentication, both the ATG Content Administration server and the Commerce SSO server will include the `DPS.InternalUsers.LDAP` module. This module changes the class of the `/atg/userprofiling/InternalProfileRepository` component from `atg.adapter.gsa.GSARepository` to `atg.adapter.composite.MutableCompositeRepository`, and includes a SQL repository component (`/atg/userprofiling/InternalGSAProfileRepository`) to serve as the primary view for the composite repository. It also includes configuration for `/atg/adapter/ldap/LDAPRepository` (the LDAP repository that provides the contributing view) and related components.

CIM prompts you for LDAP connection settings and to provide mappings between repository properties and LDAP attributes. Each `user` item in the internal profile repository is linked to an LDAP user by the `login` property. Repository properties such as `firstName`, `lastName`, and `email` can be mapped to LDAP user attributes such as `givenName`, `sn`, and `mail`. If an attribute value is changed on the LDAP server, the corresponding repository property is immediately updated automatically; no re-login is necessary for the change to take effect.

CIM configures the `user` item type as a composite item with the primary item being in the SQL repository and a contributing item being in the LDAP repository. Based on the information you provide, it creates or modifies the following configuration files:

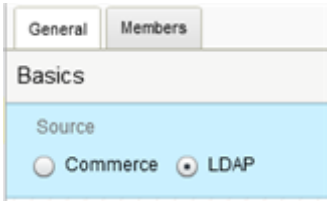
- `/atg/userprofiling/composite.xml` – configuration file for the composite repository (`InternalProfileRepository`)
- `/atg/userprofiling/internalUserProfile.xml` – definition file for the SQL repository (`InternalGSAProfileRepository`)
- `/atg/adapter/ldap/ldapUserProfile.xml` – definition file for the LDAP repository (`LDAPRepository`)
- `/atg/adapter/ldap/InitialContextEnvironment.properties` – properties file for the component that specifies the environment settings for the JNDI initial context for the LDAP server

Note that in this configuration, the LDAP directory is not writable by the Oracle Commerce Platform. The LDAP data should be maintained through your LDAP software, and be available to other systems for reading but not modifying. Therefore, the LDAP repository is configured as read-only. This means that for Commerce SSO, unlike other uses of LDAP by the Oracle Commerce Platform, you do not need to set up a password hasher component. Password hashing should be handled through the LDAP software itself.

Mapping Organizations to LDAP Groups

After you specify the mapping of user properties to LDAP user attributes, CIM prompts you to link Business Control Center LDAP organizations to LDAP groups by mapping organization properties to LDAP group attributes. A Business Control Center organization is considered an LDAP organization if the organization's `isLdap` property is set to `true`. An LDAP organization links to the LDAP group whose group ID matches the name of the organization.

The value of the `isLdap` property can be set in the Organizations interface in the Access Control area of the Business Control Center:



User Authentication

As mentioned above, the LDAP directory is not writable by the Oracle Commerce Platform. Therefore, before a user can log into Commerce SSO, an account must exist for that user in the LDAP directory.

When a user who has an LDAP account but does not have an account in the internal profile repository attempts to log into Commerce SSO, an account for that user is automatically created in the `InternalGSAProfileRepository` (assuming that the LDAP authentication succeeds). Once the user is logged in, user properties that are not linked to the `LDAPRepository` can be updated.

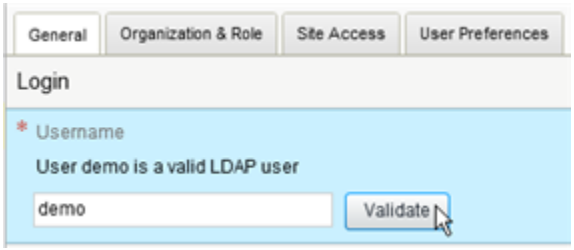
In order for Commerce SSO to automatically create an internal user in this way, the user must belong to at least one LDAP group whose group ID matches the name of an LDAP organization defined in the Business Control Center. The new user is automatically assigned to this organization (and to any other LDAP organizations that match existing LDAP groups). For each subsequent successful login, the user's organization memberships are resynchronized with the user's current LDAP group memberships.

Validating a login against the LDAP directory on the Commerce SSO server is handled through the `/atg/dynamo/security/LDAPAuthenticationService` component, which is of class `atg.security.ldap.LDAPAuthenticationService`.

Creating Users and Organizations in the Business Control Center

As discussed in the previous section, if a user who has an LDAP account but does not have an account in the internal profile repository attempts to log into Commerce SSO, an account for that user is automatically created in the `InternalGSAProfileRepository`. Profile properties that are linked to LDAP attributes are read-only and cannot be modified through the Business Control Center.

An administrator can create accounts in the internal profile repository for users who have not yet logged into Commerce SSO. Because the LDAP repository is not writable, a new user must already have an LDAP account with the same user name. The page in the Business Control Center for creating a new user has a Validate button next to the Username field that you can click to verify that the account exists in the LDAP directory:



Note that this restriction means that a user account cannot be created by duplicating an existing account and then changing the user name, since this would require writing to the LDAP directory. Therefore, the Duplicate option is disabled in the Users interface in the Access Control area.

Organizations

An administrator can create organizations in the Business Control Center that are stored in the Commerce SSO composite profile repository. There are two types of organizations supported: LDAP (which is linked to an LDAP group) and Commerce (which is stored entirely in the `InternalGSAProfileRepository` and is not linked to an LDAP group).

The name of an LDAP organization must match the group ID of the corresponding LDAP group. The page in the Business Control Center for creating a new LDAP organization has a Validate button next to the Name field that you can click to verify that the group exists in the LDAP directory.

Once you create an organization, you cannot change its name. This is true for Commerce organizations as well as for LDAP organizations.

17 Data Logging for Search Reporting

Oracle Commerce can use Oracle Business Intelligence to generate reports about how customers use Guided Search to navigate sites and access products. To make data available for use in reports, it must be collected in the form of log files, which are periodically loaded into the data warehouse.

This chapter discusses the specifics of logging search data for reporting. There are two main types of search data logging:

- Recording the content of search requests and responses, in order to generate reports about general search questions such as the most common search terms or the average number of searches per session.
- Recording when customers click on search results or dimension values to view items, and whether or not those items are subsequently purchased, in order to generate conversion reports about how customers use search and guided navigation to find and purchase items.

This chapter describes the components, processes, and configuration involved in logging the data. It includes the following sections:

The data logging and loading processes are described in the *Business Intelligence Installation and Configuration Guide*. Search reports are created from data stored in the data warehouse, which is described in the *Business Intelligence Data Warehouse Guide*.

Recording Search Requests and Responses

To record Assembler data for use in reporting, the `/atg/endeca/assembler/event/SearchRequestEventListener` component (of class `atg.endeca.assembler.event.SearchRequestEventListener`) is included in the list of Assembler event listeners specified by the `assemblerEventListeners` property of the `/atg/endeca/assembler/NucleusAssemblerFactory` component. When `SearchRequestEventListener` detects an Assembler request:

1. It constructs a JMS message of `JMSType` `atg.endeca.assembler.message.SearchMessage`, containing data from the `ContentItem` returned by the Assembler.
2. `SearchRequestEventListener` uses the `/atg/endeca/assembler/message/SearchMessageSource` component to send the JMS message.

3. The message is received by the /atg/reporting/datacollection/endeca/SearchMessageListener component, which processes the JMS message and passes the data to the components that generate the log entries for the search data.

The actual log files are written by the /atg/reporting/datacollection/endeca/SearchFileLogger component, which is of class atg.service.datacollection.RotationAwareFormattingFileLogger. This component's formatFields property specifies the JMS message properties to be logged. By default, this property is set to:

```
formatFields=timestampAsDate:MM/dd/yyyy HH:mm:ss,searchId,profileId,
  sessionId,repositoryName,languageCode,segmentList,searchType,
  searchTerm,autoCorrectTo,suggestions,suggestionSelected,spotLights,
  responseTime,numRecords,siteId,dimensionNames,dimensionValues,wordCount
```

For more information about configuring a RotationAwareFormattingFileLogger component, see the *Business Intelligence Installation and Configuration Guide*

SearchIdProvider

When the SearchRequestEventListener component construct a JMS message from the ContentItem returned by the Assembler, it assigns a unique ID to the search request.

The component that generates these search IDs is specified by the SearchRequestEventListener component's searchIdProvider property. This property must be set to a component of a class that implements the atg.endeca.assembler.event.SearchIdProvider interface. By default, the property is set to:

```
searchIdProvider=/atg/endeca/assembler/event/SearchIdProvider
```

The SearchIdProvider component is of class atg.endeca.assembler.event.SearchIdProviderImpl, which constructs the search ID using hashed values of the session ID, the search term, and the dimensions selected by the user. If you want to use different parameters for generating search IDs, you can write your own implementation of the SearchIdProvider interface, create a component of this class, and set the searchIdProvider property to your new component.

EndecaReporting Segment List

Search reporting adds the EndecaReporting segment list to the Personalization Repository (/atg/userprofiling/PersonalizationRepository). You can edit this segment list in the Business Control Center to specify the user segments that are of interest for search reporting.

The /atg/reporting/datacollection/endeca/SearchLogEntryGenerator component is configured to use this segment list when generating log entries:

```
repositoryGroupListManager=/atg/userprofiling/UserSegmentListManager
repositoryGroupListIds+=EndecaReporting
```

For more information about segment lists, see the *Personalization Programming Guide*.

Recording Search Results Selected

In order to generate reports that associate search terms with items that are viewed or purchased, your sites must record "click-through" events. These occur when a customer clicks on a product or SKU returned by a search, to view it or purchase it. The recording of these events works like this:

1. For each result returned by Guided Search, the `GetSearchClickThroughId` servlet bean generates a click-through ID, which you can append to the URL for that result using a query parameter named `searchClickId`. The servlet bean also adds the record to a cache.

2. When a customer clicks a link to view a search result, the `SearchClickThroughServlet` examines the request URL, finds the value of the `searchClickId` query parameter, and uses it to look up the record in the cache. If it finds the record, the servlet fires a JMS event containing data from the search request and response. This event is logged to be used for reporting.

This section discusses how click-through events are used to associate search results with views of specific products. The Recording Search Results Placed in Shopping Carts (page 159) section discusses how click-through events are used to associate search results with sales of specific products.

Using the GetSearchClickThroughId Servlet Bean

The `/atg/endeca/clickthrough/droplet/GetSearchClickThroughId` servlet bean is typically used in a loop that renders a list of search results. For each result, it adds the item to a cache, and generates a click-through ID to be included in the URL for viewing that item. The click-through ID consists of the search ID and the record ID, separated by a delimiter.

Input Parameter

record
The record to generate the click-through ID for.

Output Parameter

searchClickId
The click-though ID for retrieving the record from the cache.

Open Parameter

output
The open parameter for rendering the click-though ID.

Example

You can use this servlet bean in pages that render a list of search results. For example, the following JSP code creates a hyperlink to a product page and appends the `searchClickId` query parameter to the URL:

```
<c:forEach var="record" items="${contentItem.records}" >
  <dsp:droplet name="/atg/endeca/clickthrough/droplet/GetSearchClickThroughId">
   <dsp:param name="record" value="${searchResult}"/>
      <dsp:oparam name="output">
       <dsp:a href="/mystore/browse/productDetail.jsp">
         <dsp:param name="searchClickId" param="${searchClickId}"
       </dsp:a>
      </dsp:oparam>
```

```
  </dsp:droplet>
</c:foreach>
```

Configuring the Cache

The /atg/endeca/assembler/cache/SearchRequestCache component, of class atg.endeca.assembler.cache.SearchRequestCache, is a session-scoped component that manages cached search results. Each cache entry includes a search ID and the associated records for that search. Rather than storing complete records (which may use a lot of memory), the entries are objects of class atg.endeca.assembler.cache.SearchRecord, which include only a subset of the record properties.

The SearchRequestCache component's recordIdProperty specifies a record property to use as a unique key for storing and retrieving the SearchRecord objects in the record cache. The key can be any property that uniquely identifies the record. By default, this is set to:

```
recordIdPropertyName=product.repositoryId
```

Note that you must ensure that the record property is included in the ContentItem returned by the Assembler. To do this, the property must be listed in the fieldNames property of the /atg/endeca/assembler/cartridge/handler/config/ResultsListConfig component. By default, this property is set to:

```
fieldNames=record.id,record.type.raw,product.repositoryId
```

If you use a record property that is not in this list as your key, be sure to add that property to fieldNames.

The cache is designed as a Least Recently Used (LRU) cache, so if the cache is full and another entry is received, the oldest entry is deleted to make room. To optimize the tradeoff between reporting accuracy and resource use, you can tune the number of request objects in the cache and the number of records that should be cached per request by setting the following properties of the SearchRequestCache component:

- requestCount -- Specifies the maximum number of search request/response objects to store. Default is 10. To specify no maximum, set this value to -1.
- recordCount -- Specifies the maximum number of records to store per search request/response. Default is 1000. To specify no maximum, set this value to -1.

SearchClickThroughServlet

The /atg/endeca/clickthrough/servlet/SearchClickThroughServlet is inserted into the DAF servlet pipeline after the SiteSessionEventTrigger. When a user clicks a link to view a search result, SearchClickThroughServlet reads the click-through ID from the request URL and looks up the record in the SearchRequestCache. If it finds the record, it triggers a SearchClickThroughMessage JMS event, which includes the search ID, the record ID, and data about the search request.

To configure this servlet, set the following properties:

enabled
If true, the servlet processes the request. Default is false.

searchClickIdQueryArgs

An array of the query arguments to read to find the click-through ID for a viewed item. One of the values in this array must match the name of the output parameter set by `GetClickThroughId`. Default is `searchClickId`.

Limiting the Pages to Examine

By default, this servlet examines all URLs to look for click-through IDs. This process can be inefficient, because only product detail pages will typically have these IDs. Therefore `SearchClickThroughServlet` has a `clickThroughPages` property that you can use to limit the pages to examine. This property is an array of URLs, which can include asterisk (*) characters as wildcards. If `clickThroughPages` is not null, `SearchClickThroughServlet` examine only the URLs that match one of the `clickThroughPages` entries. For example, you could set `clickThroughPages` to:

```
clickThroughPages=\
  /*/*/productDetail.jsp,\
  /*/*/productDetailWithPicker.jsp,\
  /*/*/giftCertificateProduct.jsp
```

Recording Search Results Placed in Shopping Carts

To associate click-through events with items placed in shopping carts, the `/atg/reporting/datacollection/commerce/ItemAddedToOrderListener` message sink responds to JMS messages of type `atg.commerce.order.ItemAddedToOrder`. `ItemAddedToOrderListener` passes the message to the `/atg/reporting/datacollection/commerce/CommerceItemMarkerHandler` component, which adds a marker to the commerce item. The marker key is `atg.endeca`, and the marker value is the search ID.

When submitted orders are loaded into the data warehouse, the `OrderSubmitLoader` uses the `atg.endeca` markers to track which purchased items are associated with searches. This information is used to generate search conversion reports.

See the *Core Commerce Programming Guide* for more information about the `OrderSubmitLoader`.

Configuring CommerceItemMarkerHandler

In order to add a marker to a commerce item placed in a shopping cart, the `CommerceItemMarkerHandler` component is configured to track the type of repository item it is (typically `product`, `sku`, or a subtype of `product` or `sku`). To enable this tracking, `CommerceItemMarkerHandler` has a `recordItemDescriptorPropertyName` property that is set to the name of a record property that holds the name of the item type.

The `recordItemDescriptorPropertyName` property is set by default to `record.type.raw`. This is a special property created by the `/atg/endeca/index/accessor/ItemDescriptorNameAccessor` component. By default, the `ProductCatalogOutputConfig` component's XML definition file is configured to use this property accessor to include the `record.type.raw` property in the indexed records:

```
<property name="recordtyperaw" is-dimension="false" type="string"
 property-accessor="/atg/endeca/index/accessor/ItemDescriptorNameAccessor"
 output-name="record.type.raw" is-non-repository-property="true"
 text-searchable="false"/>
```

The value of the `record.type.raw` record property is the name of an item type for the record. Typically it has multiple values. For example:

```
<PROP NAME="record.type.raw">
  <PVAL>product</PVAL>
</PROP>
<PROP NAME="record.type.raw">
  <PVAL>sku</PVAL>
</PROP>
```

When an item is added to the cart, the record is checked to see if any of the values of `record.type.raw` match the item type. If so, a marker is added to the item. In the example above, if the item type is `product` or `sku`, a marker is added to the item.

The `CommerceItemMarkerHandler` component's `recordItemRepositoryIdProperty` property should be set to the name of the record property that holds the repository ID of the document-level item for the record. By default, `recordItemRepositoryIdProperty` is set to `product.repositoryId`.

As with the record property used for storing and retrieving items in the record cache, the record properties specified by `recordItemDescriptorPropertyName` and `recordItemRepositoryIdProperty` must be listed in the `fieldNames` property of the `ResultsListConfig` component to ensure they are included in the `ContentItem` returned by the Assembler. If you change the value of `recordItemDescriptorPropertyName` or `recordItemRepositoryIdProperty`, be sure to modify `fieldNames` accordingly.

18 Data Loading for Search Reporting

Data logging for search reporting occurs in the Core Commerce production environment. Much of the logging occurs in real time as customers use the site. Loading data into the data warehouse is performed in a separate Core Commerce environment, typically (but not necessarily) running on a different group of physical servers. These data loader instances do not handle customer requests; their primary purpose is to process data from the log files and load it into the data warehouse. Unlike logging, the loading process is not performed in real time. It is run on a regular schedule, typically once a day.

This chapter describes the data loading and processor components used for search reporting. It includes the following sections:

For detailed information about configuring data loading, see the *Business Intelligence Installation and Configuration Guide*.

Data Warehouse Tables

The search reporting modules add tables to the data warehouse for storing search data, and modify existing Core Commerce tables to add properties for associating order data with search data. The following is a list of the tables that store data used in search reporting:

- ARF_SEARCH
- ARF_SEARCH_TYPE
- ARF_SEARCH_TERM
- ARF_FACET_SEL
- ARF_FACET_SEL_GROUP
- ARF_FACET_SEL_MBRS
- ARF_MERCH_RULE
- ARF_MERCH_RULE_GROUP

- ARF_MERCH_RULE_ MBRS
- ARF_PROF_TYPE
- ARF_LINE_ITEM
- ARF_LINE_ITEM_SEARCH

Aggregated Data

The data warehouse includes aggregated data that is calculated using the data stored in the data warehouse tables. Components running on the data warehouse loader server refresh the calculated data at regular intervals so that it is available for reports. Performing intensive calculations before they are needed improves the speed of the queries that use the calculated data.

In Oracle database schemas, the aggregated data calculations are presented as virtual tables using materialized views. Other supported database products provide views or other comparable functions to make aggregated data available.

The virtual tables used for search reporting are:

- ARF_SEARCH_MV_TYPE -- Aggregates search information based on search type, language, site, and date.
- ARF_SEARCH_MV_HOUR -- Aggregates search information based on date and time, search type, site, and language.
- ARF_SEARCH_MV_VISIT -- Aggregates information about the average number of search requests per site visit.
- ARF_SEARCH_MV_SEARCH_TERM -- Aggregates search information based on search term, search type, site, and language.
- ARF_SEARCH_MV_FACET_GROUP -- Aggregates search information based on facet group, search type, language, and site.

Refresh Services

To refresh the aggregated data used for search reporting, the `ARF.DW.Endeca` module adds the `/atg/reporting/datawarehouse/refresh/search/SearchRefreshLauncher` component to the `refreshables` property of the `/atg/reporting/datawarehouse/refresh/RefreshService` component:

```
refreshables+=\
  search/SearchRefreshLauncher
```

The `SearchRefreshLauncher` component is of class `atg.reporting.datawarehouse.refresh.RefreshLauncherService`. This class has a `refreshables` property that specifies a list of components that are responsible for refreshing the calculated data. By default, this property is set to:

```
refreshables=\
  SiteSearchAggregateType,SiteSearchAggregateHour,\
  SiteSearchAggregateVisit,SiteSearchAggregateSearchTerm,\
```

```
SiteSearchAggregateFacetGroup
```

Each of these components is responsible for refreshing one of the virtual tables listed above. For example, the `SiteSearchAggregateFacetGroup` component refreshes the `ARF_SEARCH_MV_FACET_GROUP` virtual table.

For more information about data warehouse tables, aggregated data, and services for refreshing views, see the *Business Intelligence Data Warehouse Guide*.

Loader and Processor Components for Search Reports

The `ARF.DW.Endeca` module adds the `/atg/reporting/datawarehouse/loaders/SearchLoader` data loader component, of class `atg.reporting.datawarehouse.loader.Loader`. The `SearchLoader` component initiates the data loading process, but the actual processing of the data is performed by the `searchQuery` processor pipeline chain.

The `SearchLoader` component accesses the search log file in the `atg.reporting.endecaQuery` queue and passes it to the `/atg/reporting/datawarehouse/loaders/SearchPipelineDriver` component, of class `atg.reporting.datawarehouse.loader.FilePipelineDriver`. `SearchPipelineDriver` reads the file line by line and invokes the `searchQuery` pipeline chain. The processors in the pipeline perform such tasks as looking up data in the data warehouse, looking up data in repositories on the production site, performing calculations, and writing data to the data warehouse.

The tables below summarize the processor components in the `searchQuery` pipeline chain.

lookupQueryDay

Reads the search timestamp and extracts the date.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchDayLookupProcessor
Object	atg.reporting.datawarehouse.process.DayLookupProcessor
Transitions	Return value of 1 executes lookupQueryTime.

lookupQueryTime

Reads the search timestamp and extracts the time.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchTimeLookupProcessor
Object	atg.reporting.datawarehouse.process.TimeLookupProcessor

Transitions	Return value of 1 executes limitQueryPropertiesLength.

limitQueryPropertiesLength

Reads the length limits specified for input properties and reduces the output length of properties that exceed the specified limits.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/calculators/SearchQueryLimitLengthCalculator
Object	atg.reporting.datawarehouse.process.calculators.LimitLengthCalculator
Transitions	Return value of 1 executes lookupQueryExternalProfile.

lookupQueryExternalProfile

Looks up external user profile information associated with the query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchExternalProfileLookupProcessor
Object	atg.reporting.datawarehouse.process.SearchSwitchedLookupProcessor
Transitions	Return value of 1 or 2 executes lookupQueryInternalProfile.

lookupQueryInternalProfile

Looks up internal user profile information associated with the query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchInternalProfileLookupProcessor
Object	atg.reporting.datawarehouse.process.RepositoryItemLookupProcessor
Transitions	Return value of 1 executes lookupQuerySite.

lookupQuerySite

Looks up the site associated with the query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchSiteLookupProcessor
Object	atg.reporting.datawarehouse.process.RepositoryItemLookupProcessor
Transitions	Return value of 1 executes lookupQuerySiteVisit.

lookupQuerySiteVisit

Looks up site visit information by session ID, date, and site.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchSiteVisitLookupProcessor
Object	atg.reporting.datawarehouse.process.SiteVisitLookupProcessor
Transitions	Return value of 1 executes lookupQueryLanguage.

lookupQueryLanguage

Looks up the language code for the search query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchLanguageLookupProcessor
Object	atg.reporting.datawarehouse.process.LanguageLookupProcessor
Transitions	Return value of 1 executes lookupQuerySegmentCluster.

lookupQuerySegmentCluster

Looks up the segment cluster for the search query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchSegmentClusterLookupProcessor
Object	atg.reporting.datawarehouse.process.GroupLookupProcessor
Transitions	Return value of 1 executes lookupQueryDemographic.

lookupQueryDemographic

Looks up demographic information for the user associated with the search query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchDemographicLookupProcessor
Object	atg.reporting.datawarehouse.process.DemographicLookupProcessor
Transitions	Return value of 1 executes lookupQueryType.

lookupQueryType

Looks up the search query type.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchTypeLookupProcessor
Object	atg.reporting.datawarehouse.process.RepositoryItemLookupProcessor
Transitions	Return value of 1 executes lookupQuerySearchTerm.

lookupQuerySearchTerm

Looks up the search term of the query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchTermLookupProcessor
Object	atg.reporting.datawarehouse.process.RegularRepositoryItemLookupProcessor
Transitions	Return value of 1 executes lookupQueryFacetGroup.

lookupQueryFacetGroup

Looks up the facet group of the query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchFacetGroupLookupProcessor

Object	atg.reporting.datawarehouse.process.GroupLookupProcessor
Transitions	Return value of 1 executes lookupQuerySpotlightGroup.

lookupQuerySpotlightGroup

Looks up the spotlight group of the query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchSpotlightGroupLookupProcessor
Object	atg.reporting.datawarehouse.process.GroupLookupProcessor
Transitions	Return value of 1 executes lookupQueryProfileType.

lookupQueryProfileType

Looks up the profile type associated with the query.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchProfileTypeLookupProcessor
Object	atg.reporting.datawarehouse.process.RepositoryItemLookupProcessor
Transitions	Return value of 1 executes searchFactCalculator.

searchFactCalculator

Updates the autocorrect and suggestion properties based on values in the log file.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/calculators/SearchFactCalculator
Object	atg.reporting.datawarehouse.process.calculators.SearchFactCalculator
Transitions	Return value of 1 executes logQuery.

logQuery

Inserts the record in the ARF_SEARCH table.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchLoggerProcessor
Object	atg.reporting.datawarehouse.process.RepositoryLoggerProcessor
Transitions	Last processor in the pipeline chain.

Processor Components for Search Conversion Reports

To enable creation of search conversion reports, the search data loader modules modify the Core Commerce `lineItem` pipeline to add several processors. These processors are used to link purchased items with the search terms, facet groups, and spotlight groups that customers use to navigate to the items. They create these links by processing the markers that are added to the order items by the `CommerceItemMarkerHandler` component, as described in the *Data Logging for Search Reporting* (page 155) chapter.

The tables below summarize the processor components that search reporting adds to the Core Commerce `lineItem` pipeline.

getSearchId

Fetches the search ID using the marker key.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchIdProcessor
Object	atg.reporting.datawarehouse.process.CommerceItemMarkerProcessor
Transitions	Return value of 1 executes lookupSearch.

lookupSearch

Looks up the search record in the fact table using the search ID.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/SearchFactLookupProcessor
Object	atg.reporting.datawarehouse.process.RepositoryItemLookupProcessor
Transitions	Return value of 1 executes lookupSearchTerm.

lookupSearchTerm

Associates the search term with the line item.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/LineItemSearchTermProcessor
Object	atg.reporting.datawarehouse.process.PropertyAssignmentProcessor
Transitions	Return value of 1 executes lineItemSearchFacetGroup.

lineItemSearchFacetGroup

Associates the search facet group with the line item.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/LineItemSearchFacetGroupProcessor
Object	atg.reporting.datawarehouse.process.PropertyAssignmentProcessor
Transitions	Return value of 1 executes lineItemSpotlightGroup.

lineItemSpotlightGroup

Associates the search spotlight group with the line item.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/LineItemSearchSpotlightGroupProcessor
Object	atg.reporting.datawarehouse.process.PropertyAssignmentProcessor
Transitions	Return value of 1 executes logLineItem.

logLineItemSearch

Updates the search-related information for the line item.

Transactional Mode	TX_MANDATORY
Component	/atg/reporting/datawarehouse/process/LineItemSearchLoggerProcessor

Object	atg.reporting.datawarehouse.process.LineItemSearchRepositoryLoggerProcessor
Transitions	Last processor in the pipeline chain.

Maintenance Services

If search data is loaded into the data warehouse after order data has been loaded, the order data in the data warehouse must be updated to associate purchased items with the search data. The order line items are updated to link them to the search terms, facet groups, and spotlight groups that customers used to navigate to those items.

To ensure that the search data and order data are linked properly, the search data loader modules include the `/atg/reporting/datawarehouse/process/jobs/SearchMaintenanceService` component, of class `atg.reporting.datawarehouse.process.job.MaintenanceService`. This service is responsible for executing jobs specified in its `maintenanceJobs` property. This property is an array of components that implement the `atg.reporting.datawarehouse.process.job.MaintenanceJob` interface. By default, it is set to:

```
maintenanceJobs=\
  /atg/reporting/datawarehouse/process/jobs/SearchLineItemMaintenanceJob
```

The `SearchMaintenanceService` component is registered with the `SearchLoader` component as a listener. `SearchMaintenanceService` detects when search data is loaded and invokes the `SearchLineItemMaintenanceJob` component to update the associated order data in the data warehouse.

19 Search Reporting Dashboards

Search reporting adds the Search Performance Dashboard to the Oracle Business Intelligence user interface. This dashboard displays a variety of analyses of Oracle Commerce Guided Search data. In addition, search reporting adds search-related analyses to the ATG Web Commerce Performance Dashboard.

This chapter summarizes the search analyses available in these dashboards. It includes the following sections:

Search Performance Dashboard (page 171)

ATG Web Commerce Performance Dashboard (page 172)

For more information about accessing Oracle Commerce reports and analyses in Oracle Business Intelligence, see the *Reports Guide*.

Search Performance Dashboard

The Search Reporting Dashboard consists of four pages of analyses:

- Search Activity -- statistical breakdowns of the way Guided Search has been used, including searches per visit and per site, search types, and response times for search queries
- Keyword Analysis -- information about search terms used, such as top search terms, terms that returned no results, and terms that were autocorrected or returned alternate suggestions
- Search Merchandising -- information about the relationships between search queries and items sold
- Dimension Analysis -- information about how often specific search dimensions or groups of dimensions were selected

Search Activity

The Search Activity page includes the following reports:

- Key Search Indicators
- Response Time Analysis
- Searches Per Visit by Site
- Search Type Breakdown
- Request and Site Visit Summary

Keyword Analysis

The Keyword Analysis page includes the following reports:

- Top Search Terms
- Search Terms with No Results
- Spell Correction and Alternate Suggestions Summary
- Top Autocorrected Search Terms
- Top Search Terms without Spotlight

Search Merchandising

The Search Merchandising page includes the following reports:

- Searches Resulting in Most Purchases, which also contains links to:
  - Top Products Associated with Search
  - Top Dimensions Associated with Search
- Search Conversion Funnel
- Search Conversion Analysis
- Conversion Rate for Spotlights
- Top Spotlights Triggered

Dimension Analysis

The Dimension Analysis page includes the following reports:

- Top Single Dimensions Selected
- Top Single Dimension Values Selected
- Top Dimension Pairs Selected
- Top Two Dimension Values Selected
- Top Dimension Triple Selected
- Top Three Dimension Values Selected
- Top Dimension Values without Spotlight

ATG Web Commerce Performance Dashboard

Search reporting adds a Key Search Indicator report to the Traffic page on the ATG Web Commerce Performance Dashboard. It also adds a Search page to the dashboard that includes the following reports:

- Key Search Indicators
- Response Time Analysis
- Search Conversion Analysis
- Top Search Terms
- Search Terms with No Results
- Searches Resulting in Most Purchases

20 Appendix A: Support for Older Deployment Templates

This manual assumes you are using a CAS-based deployment template for your EAC applications, as described in the *Oracle Commerce Guided Search Administrator's Guide*. If you are creating new EAC applications, it is highly recommended that you use this type of deployment template.

If you have EAC applications created in an earlier release, they may be using an older Forge-based deployment template such as the one described in the *Oracle Endeca Commerce Deployment Template Module for Product Catalog Integration Usage Guide* in the Oracle Endeca Commerce Tools and Frameworks 11.0 documentation. You can either recreate your EAC applications with a CAS-based deployment template or continue to use your existing applications based on the older-style template. If you choose the latter option, you will need to reconfigure several Oracle Commerce Platform components, because the default configuration of these components now assumes the use of a CAS-based template.

This appendix describes the configuration changes needed for Oracle Commerce Platform components to work with EAC applications that use an older Forge-based Oracle Commerce Guided Search deployment template. It discusses the following topics:

Record Store Naming

For a CAS-based deployment template, the record store names have the format *`applicationName-recordStoreType`*. For example, for an EAC application named `ATGen`, the dimension values record store is named `ATGen-dimvals`.

For a Forge-based deployment template, the format of the record store names is *`applicationName_languageCode_recordStoreType`*. So, for an EAC application named `ATGen` that indexes records in English, the dimension values record store is named `ATGen_en_dimvals`.

The format of the record store names is configured by the `recordStoreNameFormatString` property of the `/atg/endeca/index/IndexingApplicationConfiguration` component. The value of this property is a format string in which 0 is the EAC application name, 1 is the two-character language code, and 2 is the type of record store. By default, this is set to:

```
recordStoreNameFormatString={0}-{2}
```

Note that the 1 does not appear, because record store names for CAS-based applications do not include the language code.

For EAC applications that use Forge-based deployment templates, set the value of this property to:

```
recordStoreNameFormatString={0}_{1}_{2}
```

Schema Export

For a CAS-based deployment template, an application's schema definition is created as Configuration Import API objects, which are submitted to the Endeca Configuration Repository. These objects are created and submitted by the `/atg/endeca/index/ConfigImportDocumentSubmitter` component. The `documentSubmitter` property of components of class `atg.endeca.index.schema.SchemaExporter` (including the `SchemaExporter`, `ArticleSchemaExporter`, and `MediaContentSchemaExport` components) is set by default to:

```
documentSubmitter=/atg/endeca/index/ConfigImportDocumentSubmitter
```

For a Forge-based deployment template, an application's schema definition is created as schema records, which are written to the schema record store. To configure this behavior, change the value of each `SchemaExporter` component's `documentSubmitter` property to:

```
documentSubmitter=/atg/endeca/index/SchemaDocumentSubmitter
```

Hierarchical Dimension Export

There are a number of differences in how the names and the values of hierarchical dimension value properties are represented in applications that use a CAS-based deployment template versus applications that use a Forge-based template. These differences are discussed below.

Root Node Naming and Export

In a CAS-style application, the name of the root node in a dimension hierarchy is the forward slash (`/`). So, for example, the forward slash is the name of the root node of the category dimension.

The following example is part of a record created by the `CategoryToDimensionOutputConfig` component that represents a top-level category. The value of this category's `dimval.parent_spec` property is the root node:

```
<PROP NAME="dimval.parent_spec">
  <PVAL>/</PVAL>
</PROP>
```

In a Forge-based application, the name of the root node of a hierarchical dimension is the name of the dimension. For example, the root node of the category dimension hierarchy is `product.category`. So a record representing a top-level category includes this instead:

```
<PROP NAME="dimval.parent_spec">
  <PVAL>product.category</PVAL>
</PROP>
```

The value for the name of the root node is configured separately for the category hierarchy and the repository item type hierarchy. For the repository item type hierarchy, the `rootParentSpecifier` property of components of class `atg.endeca.index.dimension.RepositoryTypeHierarchyExporter` (including the `RepositoryTypeDimensionExporter`, `ArticleDimensionExporter`, and `MediaContentDimensionExporter` components) is set by default to:

```
rootParentSpecifier=/
```

For a Forge-based application, set the `rootParentSpecifier` property of each `RepositoryTypeHierarchyExporter` component to null so it uses its default value, which is the dimension name:

```
rootParentSpecifier^=/Constants.null
```

For the category hierarchy, the name of the root node is configured through the `defaultValue` property of the `/atg/commerce/endeca/index/accessor/ParentSpecPropertyAccessor` component. (See Category Dimension Value Accessors (page 63) for information about this component.) This property is set by default to:

```
defaultValue=/
```

In a Forge-based application, change the value of this property to the name of the dimension:

```
defaultValue=product.category
```

Root Node Export

For a CAS-based application, the Oracle Commerce Platform does not create a record representing the root node of a dimension hierarchy. Instead, Oracle Commerce Guided Search automatically creates the root node and names it "/". For a Forge-based application, however, the Oracle Commerce Platform does create a record representing the root node, and gives the node the name of the dimension.

This behavior is configured separately for the category hierarchy and the repository item type hierarchy. For the repository item type hierarchy, the `createRootNode` property of `RepositoryTypeHierarchyExporter` components (including `RepositoryTypeDimensionExporter`, `ArticleDimensionExporter`, and `MediaContentDimensionExporter`) is set by default to:

```
createRootNode=false
```

In a Forge-based application, set the `createRootNode` property of all `RepositoryTypeHierarchyExporter` components to `true`.

For the category hierarchy, creation of a record for the root node is controlled through the `indexingSynchronizations` property of the `CategoryToDimensionOutputConfig` component. This property accepts an array of components of classes that implement the `atg.repository.search.indexing.IndexingSynchronization` interface. The default configuration of this property includes the `/atg/commerce/endeca/index/CategoryRootNodeSynchronization` component, which is responsible for creating the root node for Forge-based applications.

Configuring for Preview

To support previewing your sites in Experience Manager preview and Business Control Center preview, the `CategoryRootNodeSynchronization` component includes the following properties:

- `rootNodeParentSpec` -- specifies the value of the `dimval.parent_spec` property of the root node of the category hierarchy
- `dummyNodeParentSpec` -- specifies the value of the `dimval.parent_spec` property of the dummy node used for previewing unindexed categories

These properties are configured by default to support a CAS-based application:

```
dummyNodeParentSpec=/
rootNodeParentSpec=/
```

For a Forge-based application, set these properties to the name of the dimension:

```
dummyNodeParentSpec=product.category
rootNodeParentSpec=product.category
```

Dimension Value Property Names

There are a number of differences in the names of dimension value properties in Forge-based and CAS-based applications:

- In a Forge-based application, the ID property of a dimension is named `dimval.qualified_spec`. In a CAS-based application, the property is named `Endeca.Id`.
- In a Forge-based application, the property specifying the name of a dimension must be named `dimval.dimension_spec`. In a CAS-based application, the property is named `dimval.dimension_name`.
- In a Forge-based application, custom properties on dimension nodes must have names that begin with `dimval.prop`. In a CAS-based application, no prefix is required.

By default, the components that produce dimension value records (including the `CategoryToDimensionOutputConfig`, `RepositoryTypeDimensionExporter`, `ArticleDimensionExporter`, and `MediaContentDimensionExporter` components) output records whose property names reflect the older Forge-based deployment template. To support the naming conventions

used with CAS-based deployment templates, the `propertyNameReplacementMap` property of the `DimensionDocumentSubmitter` component is used to map the older-style names to the new ones. By default, this property is set as follows:

```
propertyNameReplacementMap=\
  dimval.qualified_spec=Endeca.Id,\
  dimval.dimension_spec=dimval.dimension_name,\
  dimval.prop.category.ancestorCatalogIds=category.ancestorCatalogIds,\
  dimval.prop.category.rootCatalogId=category.rootCatalogId,\
  dimval.prop.displayName_es=displayName_es,\
  dimval.prop.displayName_en=displayName_en,\
  dimval.prop.displayName_de=displayName_de,\
  dimval.prop.category.repositoryId=category.repositoryId,\
  dimval.prop.category.catalogs.repositoryId=category.catalogs.repositoryId,\
  dimval.prop.category.siteId=category.siteId
```

So, for example, `dimval.qualified_spec` is renamed to `Endeca.Id` in the output records, `dimval.dimension_spec` is renamed to `dimval.dimension_name`, and `dimval.prop.category.repositoryId` is renamed to `category.repositoryId`.

For a Forge-based application, set the `propertyNameReplacementMap` property to null to restore the older-style names:

```
propertyNameReplacementMap^=/Constants.null
```

And set the `idPropertyName` property of the `DimensionDocumentSubmitter` component to `dimval.qualified_spec`:

```
idPropertyName=dimval.qualified_spec
```

Index

S

T

U

V

W