ORACLE®

COMMERCE

Version 11.0

ATG-Endeca Integration Guide

Oracle ATG
One Main Street
Cambridge, MA 02142
USA

ATG-Endeca Integration Guide

Product version: 11.0
Release date: 01-10-14
Document identifier: EndecaIntegrationGuide1402071827

Table of Contents

1 Introduction

The ATG-Endeca integration enables customers of Oracle ATG Web Commerce and Oracle Endeca Commerce to index ATG product catalog data in Endeca MDEX engines, where it can then be queried and the results can be displayed on commerce sites. This document describes how to configure ATG indexing and querying components to work with Oracle Endeca Commerce.

Note that Commerce Reference Store makes extensive use of the ATG-Endeca integration to demonstrate the use of both ATG-driven and Endeca-driven content on commerce sites, and in some cases extends the capability of the integration. See the Commerce Reference Store documentation for more information.

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

Installation Requirements

The ATG-Endeca integration requires that Oracle ATG Web Commerce and Oracle Endeca Commerce software (including either Oracle Endeca Guided Search or Oracle Endeca Experience Manager), be installed in your environment. We also suggest that you initially install ATG Oracle Web Commerce Reference Store, so that you have an application and data to work with as you familiarize yourself with the integration.

For information on installing Oracle ATG Commerce software, see the *Installation and Configuration Guide*. For information on installing Commerce Reference Store, see the *Commerce Reference Store Installation and Configuration Guide*. For information on installing Oracle Endeca Commerce software, see the *Oracle Endeca* and other related Oracle Endeca installation documentation.

Creating the Endeca Applications

To create an Endeca application to integrate with ATG, use the Endeca deployment template designed to work with product catalog data. (See the Endeca *Deployment Template Module for Product Catalog Integration Usage Guide* for details.) This deployment template has a script that creates various Endeca CAS (Content Acquisition System) record stores that the ATG-Endeca integration writes to. The naming convention for these record stores is:

```
application-name_language-code_record-store-type
```

So for an application named `ATGen` that indexes ATG product catalog data in English, the record stores are:

- `ATGen_en_data`-- Holds data records representing SKUs or products.
- `ATGen_en_dimvals`-- Holds dimension value records generated from the category hierarchy and from the hierarchy of repository item types.
- `ATGen_en_schema`-- Holds records representing property and dimension definitions generated from the set of ATG properties being indexed.

The ATG-Endeca integration includes classes and components that write to these records stores. Note that there is also an `ATGen_en_prules` record store, which is used to create Endeca 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 Oracle Endeca Commerce. See the Oracle Endeca Commerce documentation for information about creating precedence rules.

Determining the Number of Endeca Applications to Create

For each ATG Server instance, you must have at least one unique Endeca application and corresponding MDEX. For example, if you are configuring a ATG Content Administration server and a production server, you will need a minimum of two Endeca applications and two MDEX engines. If your product catalog has data in multiple languages, the number of Endeca applications you have per server depends on your approach to indexing these languages, as described below.

One Language per MDEX

In this configuration, you have one MDEX for each language for each server. For example, if you have three languages—English, German, and Spanish—and you have two servers—Content Administration and Production—you must have six Endeca applications:

Content Administration/English
Content Administration/German
Content Administration/Spanish
Production/English
Production/German
Production/Spanish

Each ATG server should use a different Endeca base application name, which by default is used for all of the applications associated with that server. For example, you could set the base application name for the Content Administration server to `ATGCA`, and set the base application name for the Production server to `ATGProd`. By default, the names of the applications for the different languages on a given server are distinguished by adding

the two-letter code for the language to the base application name. So, for example, the names of the Content Administration-related Endeca applications would be `ATGCAen`, `ATGCAde`, and `ATGCAes`, and the names of the Production-related Endeca applications would be `ATGProden`, `ATGProdde`, and `ATGProdes`. (Note that you can override this naming convention if you prefer; see Configuring the ApplicationConfiguration Component (page 4).)

As you create the Endeca applications using the deployment template, be sure to specify the correct language code for each application. Also, be sure to provide unique ports for the `LiveDgraph`, `AuthoringDgraph`, and `LogServer` for each application.

All Languages in a Single MDEX

If you plan to have all languages indexed in a single MDEX, you only need to create one Endeca application for each ATG server instance. Each ATG server should use a different Endeca base application name; for example, you could set the base application name for the Content Administration server to `ATGCA`, and set the base application name for the Production server to `ATGProd`. By default, the name of the application is the base application name plus the two-letter language code for the default language for the application; for example, `ATGCAen` and `ATGProden`. (Note that you can override this naming convention; see Configuring the ApplicationConfiguration Component (page 4).) Be sure to specify the default language (in this case, `en`) in the `/atg/endeca/ApplicationConfiguration` component's `defaultLanguageForApplications` property for each ATG server instance:

```
defaultLanguageForApplications=en
```

As you create the Endeca applications using the deployment template, be sure to provide unique ports for the `LiveDgraph`, `AuthoringDgraph`, and `LogServer`.

Provisioning the Endeca Applications

For each Endeca application you create, you must provision it by running the `initialize_services.sh|bat` script found in the application's `/control` directory. Therefore, if you have six Endeca applications, you must invoke this script six times. The `initialize_services.sh` script is found in the following location: `/endeca/Endeca-application-directory/your-application/control/`.

Configuring the ATG Server Instances in CIM

You must configure your ATG server instances for an ATG-Endeca 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 ATG-Endeca integration, select ATG-Endeca Integration and Oracle ATG Web Commerce in the Product Selection menu. If your installation includes Oracle ATG Commerce Reference Store, you can select Oracle ATG Commerce Reference Store instead. Your server installations will automatically include ATG Commerce and the ATG-Endeca integration, because Commerce Reference Store requires them.

ATG Server Instance Creation

During your ATG server instance configuration, you must provide information about your Endeca environment so that the ATG server instance can communicate with Oracle Endeca Commerce. 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
Endeca base application name	ATG
Fully qualified Workbench host name, including domain	n/a
Workbench port	8006
Default MDEX host name	`localhost`
Default MDEX port number	15000

After your ATG server instances are configured in CIM, start them 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 ATG-Endeca 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.

defaultLanguageForApplications

The two-letter code for the default language for the application. This should be set only if there is a single MDEX for all languages being indexed. By default the value of this property is null.

baseApplicationName

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

keyToApplicationName

A map of application keys to application names. You can use this property to override the default application naming convention discussed in Determining the Number of Endeca Applications to Create (page 2). For example, if an environment supports English, Spanish, and German, and there is a separate Endeca application for each language, you can specify the application names like this:

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

The application keys in this case are the two-letter language codes. An array of the keys is stored in the read-only `applicationKeys` property.

If there is a single application, the key is `default`. You can specify the application name like this:

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

defaultApplicationKey

The key of the application to use if the current application cannot otherwise be determined. 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 Endeca application, you do not need to set this property; it will automatically be set to `default`.

workbenchHostName

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

workbenchPort

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

Starting the Indexing Process

The indexing process can be started in two ways: automatically as part of running a full deployment through ATG Content Administration, or manually using the Dynamo Server Admin.

Increasing the Transaction Timeout and Datasource Connection Pool Values

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

If indexing is not successful, it may be related to the transaction timeout setting in your application server. Oracle ATG recommends setting a transaction timeout of 300 seconds or greater. All supported application servers time out long-running transactions by marking the active transaction as rolled back (essentially, by calling `setRollbackOnly` on the transaction), which can result in problems when indexing. If your indexing process fails, try increasing the transaction timeout setting. For details on changing your transaction timeout, see *Setting the Transaction Timeout on* , *Setting the Transaction Timeout on* , or *Setting the Transaction Timeout on* in the *Installation and Configuration Guide*.

Increasing the Datasource Connection Pool

Oracle ATG 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.

Indexing As Part of a Deployment

You can configure your environment so that when you run a deployment in Content Administration, indexing is automatically started after the deployment is finished. To make this automatic triggering occur, add the following three components and their configuration to the `localconfig` layer for your Content Administration server.

/atg/commerce/endeca/index/CategoryToDimensionOutputConfig

Specify the following property for the `CategoryToDimensionOutputConfig` component:

```
targetName=Production
```

/atg/commerce/search/ProductCatalogOutputConfig

Specify the following property for the `ProductCatalogOutputConfig` component:

```
targetName=Production
```

/atg/search/SynchronizationInvoker

Specify the following properties for the `SynchronizationInvoker` component:

```
host=atg-production-server-host
rmi=8860
```

Manually Starting the Indexing Process

To manually start an indexing job, log in to ATG Dynamo Administration for the appropriate ATG server instance and navigate to `/atg/commerce/endeca/index/ProductCatalogSimpleIndexingAdmin` component. From here, you can click Baseline Index to start a baseline index, or Partial Index to start a partial update.

Monitoring the Indexing Process

Regardless of how an indexing process has been started, you can monitor its progress in ATG Dynamo Administration by viewing the `/atg/commerce/endeca/index/ProductCatalogSimpleIndexingAdmin`

component. Each phase of the indexing process is listed in the table under Indexing Job Status. To dynamically refresh the window, enable the Auto Refresh option below the table.

Viewing the Indexed Data

You can view the indexed data residing in your MDEX engines using Oracle Endeca'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 Endeca 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.

ATG Modules

The main ATG-Endeca integration modules are:

Module	Description
`DAF.Endeca.Index`	Includes the necessary classes for exporting data to CAS record stores and triggering indexing via the EAC, 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.

Module	Description
DCS.Endeca.Assembler	Contains Commerce-specific configuration for query-related components.

Note that when you assemble an application that includes any of the modules listed in the table above, the DAF.Search.Base and DAF.Search.Index modules are automatically included in the EAR file as well. These modules contain core repository indexing classes that are subclassed in the Endeca-specific modules. In addition, some of the Endeca-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 Overview of Indexing

To make your product catalog available for searching, the Oracle ATG Web Commerce platform must transform the data into the appropriate format, and then submit this data to Oracle Endeca Commerce for indexing.

The process of indexing ATG product catalog data in Oracle Endeca Commerce works like this:

1. ATG components transform the catalog repository data into Endeca records that represent Endeca properties, dimensions, and schema:
   - Properties of ATG products and SKUs are used to create Endeca properties and non-hierarchical dimensions.
   - The ATG category hierarchy is used to create a hierarchical category dimension in Oracle Endeca Commerce. The hierarchy of repository item types in the product catalog is used to create another hierarchical Endeca dimension.
   - An Endeca schema is created by examining the set of ATG properties to be indexed.
2. The generated records are submitted to Endeca CAS data, dimension value, and schema record stores.
3. The Endeca EAC is invoked, which creates Forge processes that process the record stores and invoke indexing.

This chapter provides an overview of the classes and components that perform these steps, and the user interface provided for managing the process. It includes these sections:

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

Indexable Classes

The ATG platform includes an interface, `atg.endeca.index.Indexable`, that is implemented by the classes responsible for creating Endeca 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 ATG-Endeca integration includes two components of this class:

- `/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 Endeca records. The configuration of these components is discussed in detail in EndecaIndexingOutputConfig Components (page 18).

ProductCatalogOutputConfig Component

The `ProductCatalogOutputConfig` component specifies how to create Endeca data records that represent items in the ATG 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 ATG 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 might 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 record. Note that the actual records submitted to the CAS data record store are in a binary object format, not XML.

```
<RECORD>
  <PROP NAME="record.spec">
    <PVAL>
  clothing-sku-xsku1013..xprod1003.masterCatalog.en__US.plist3080003__plist3080002
    </PVAL>
  </PROP>
  <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>
</RECORD>
```

CategoryToDimensionOutputConfig Component

The `CategoryToDimensionOutputConfig` component specifies how to create Endeca dimension value records that represent categories from the ATG product catalog. This category dimension makes it possible to use Oracle Endeca Commerce 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` component, as described in the CategoryTreeService Class (page 12) section.

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

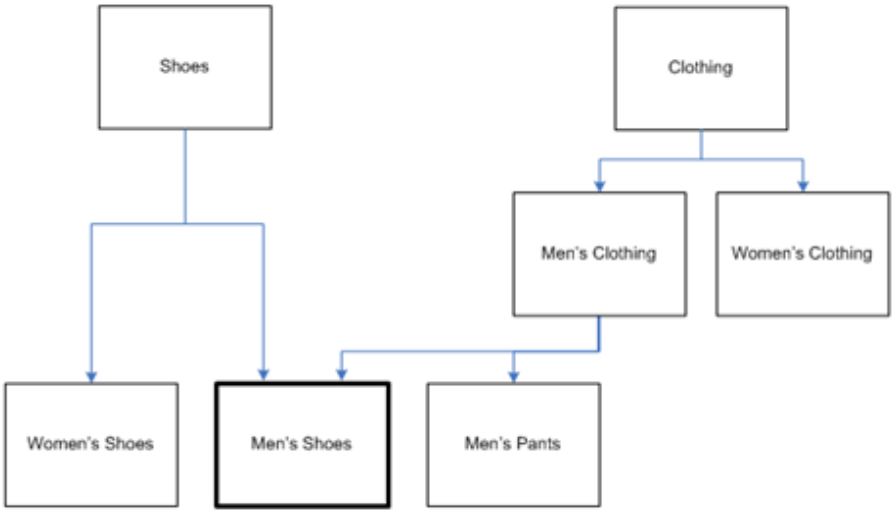
```
<RECORD>
  <PROP NAME="dimval.spec">
    <PVAL>rootCategory.cat10016.cat10014.catDeskLamps</PVAL>
  </PROP>
  <PROP NAME="dimval.qualified_spec">
    <PVAL>product.category:rootCategory.cat10016.cat10014.catDeskLamps</PVAL>
  </PROP>
  <PROP NAME="dimval.prop.category.rootCatalogId">
    <PVAL>masterCatalog</PVAL>
  </PROP>
  <PROP NAME="dimval.prop.category.ancestorCatalogIds">
    <PVAL>masterCatalog</PVAL>
  </PROP>
  <PROP NAME="dimval.dimension_spec">
    <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="dimval.prop.category.repositoryId">
    <PVAL>catDeskLamps</PVAL>
  </PROP>
  <PROP NAME="dimval.prop.category.catalogs.repositoryId">
    <PVAL>masterCatalog,homeStoreCatalog</PVAL>
  </PROP>
  <PROP NAME="dimval.display_name">
    <PVAL>Desk Lamps</PVAL>
  </PROP>
</RECORD>
```

CategoryTreeService Class

The ATG-Endeca integration uses the category hierarchy in the ATG product catalog to construct a category dimension in Oracle Endeca Commerce. In some cases, the hierarchy cannot be translated directly, because ATG's catalog hierarchy supports categories with multiple parent categories, while Endeca 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 ATG-Endeca 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 ATG-Endeca 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 Oracle Endeca Commerce 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 Endeca dimension value records from the hierarchy of repository item types in the product catalog, 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 an Endeca 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 ATG-Endeca 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.

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

```
<RECORD>
  <PROP NAME="dimval.dimension_spec">
    <PVAL>item.type</PVAL>
  </PROP>
  <PROP NAME="dimval.display_name">
    <PVAL>Product</PVAL>
  </PROP>
  <PROP NAME="dimval.qualified_spec">
    <PVAL>item.type:product</PVAL>
  </PROP>
  <PROP NAME="dimval.spec">
    <PVAL>product</PVAL>
```

```
  </PROP>
  <PROP NAME="dimval.parent_spec">
    <PVAL>item.type</PVAL>
  </PROP>
</RECORD>
```

SchemaExporter Class

The `atg.endeca.index.schema.SchemaExporter` class is responsible for generating schema records and submitting them to the Endeca schema record store. The `/atg/commerce/endeca/index/SchemaExporter` component of this class examines the `ProductCatalogOutputConfig` definition file and generates a schema record for each ATG property that is output. The schema record indicates whether the ATG property should be treated as a property or a dimension by Oracle Endeca Commerce, whether it should be searchable, and the data type of the property or dimension.

For example, the following is an XML representation of a schema record for the `product.description` property, which identifies it as a searchable Endeca property whose data type is `string`:

```
<RECORD>
  <PROP NAME="attribute.name">
    <PVAL>product.description</PVAL>
  </PROP>
  <PROP NAME="attribute.source_name">
    <PVAL>product.description</PVAL>
  </PROP>
  <PROP NAME="attribute.display_name">
    <PVAL>Product Description</PVAL>
  </PROP>
  <PROP NAME="attribute.property.data_type">
    <PVAL>string</PVAL>
  </PROP>
  <PROP NAME="attribute.type">
    <PVAL>property</PVAL>
  </PROP>
  <PROP NAME="attribute.search.searchable">
    <PVAL>true</PVAL>
  </PROP>
</RECORD>
```

Submitting the Records

Once the records have been generated, they are submitted to the appropriate CAS record stores by components of class `atg.endeca.index.RecordStoreDocumentSubmitter`. The ATG platform includes three components of this class, each of which is configured to submit to a different record store:

- `/atg/endeca/index/DataDocumentSubmitter` -- Submits records to the data record store (for example, `ATGen_en_data`).
- `/atg/endeca/index/DimensionDocumentSubmitter` -- Submits records to the dimension values record store (for example, `ATGen_en_dimvals`).

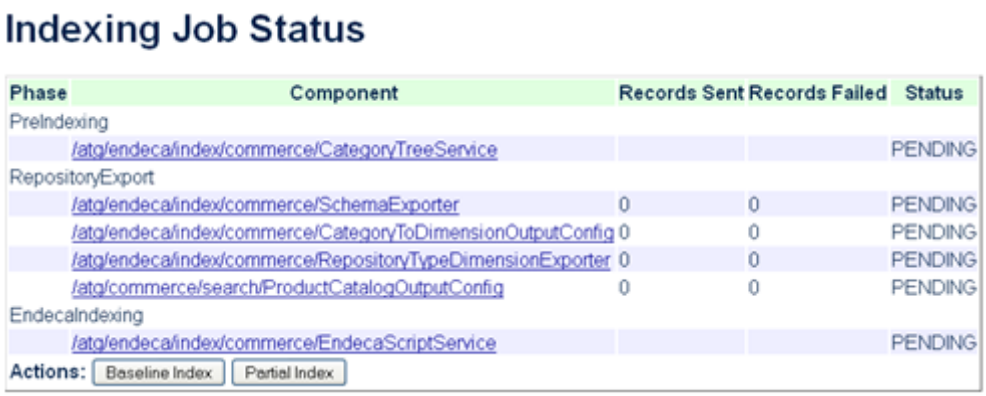
- `/atg/endeca/index/SchemaDocumentSubmitter` -- Submits records to the schema record store (for example, `ATGen_en_schema`).

The `EndecaIndexingOutputConfig`, `RepositoryTypeHierarchyExporter`, and `SchemaExporter` classes each have a `documentSubmitter` property that is used to specify a document submitter component to use to submit records to the appropriate CAS record store. 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`
`SchemaExporter`	`SchemaDocumentSubmitter`

Managing the Process

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



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

The UI provides buttons for initiating an Endeca baseline index or a partial update. Note that even if you click Partial Index, a baseline update may be invoked if the changes since the last baseline update necessitate it. See EndecaIndexingOutputConfig Components (page 18) for more information.

3 Configuring the Indexing Components

This chapter provides detailed information about the indexing-related Nucleus components in the ATG-Endeca integration, what they do, how they are configured, and how you can modify them to alter various aspects of indexing. It includes the following sections:

IndexingApplicationConfiguration Component

The `atg.endeca.index.configuration.IndexingApplicationConfiguration` class provides a central place for configuring various indexing settings. The ATG-Endeca 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 ATG environment.

CASHostName

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

```
CASHostName=localhost
```

CASPort

The port number of the machine running CAS. The default setting is:

```
CASPort=8500
```

eacHostName

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

```
eacHost=localhost
```

eacPort

The port used by the EAC server. The default setting is:

```
eacPort=8888
```

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:

definitionFile

The full Nucleus pathname of the XML indexing definition file that specifies the repository item types and properties to include in the Endeca 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 35) chapter for information about the definition file's elements and attributes that configure how ATG repository items are transformed into Endeca records.

repository

The full Nucleus pathname of the repository that the definition file applies to. For both the `ProductCatalogOutputConfig` and `CategoryToDimensionOutputConfig`, 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 *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 26) for more information.

forceToBaselineOnChange

If `true`, a baseline update is performed when a partial update is invoked, 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. For `ProductCatalogOutputConfig`, this property is set to `false` by default, because the component does not generate dimension values.

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 22) 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 22) 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 (e.g., gift wrapping) by setting the `excludedItemsAncestorIds` property of the `ProductCatalogOutputConfig` component to:

```
excludedItemsAncestorIds=\
```

```
  NonNavigableProducts,homeStoreNonNavigableProducts
```

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 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 Oracle Endeca Commerce, in case different indexed ATG 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.

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,
$repository.repositoryName=repositoryName,
$itemDescriptor.itemDescriptorName=type,
$siteId=siteId,
$url=url,
$baseUrl=baseUrl
```

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 actual generation is performed by data loader components. Depending on your ATG 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 ATG-Endeca 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 ATG-Endeca 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 ATG-Endeca 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 it is a good idea to perform a bulk load occasionally to ensure the integrity of the indexed data.

Note that Oracle Endeca Commerce always does a baseline update after ATG performs bulk loading, and typically does a partial update after incremental loading. In some cases, however, a baseline update may be invoked after incremental loading. For example, if incremental loading adds a new category dimension value, a baseline update must be performed. See EndecaIndexingOutputConfig Components (page 18) 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 IndexingOutputConfig 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 ATG platform resources to process, but reduce the number of Endeca jobs required (and hence the overhead associated with starting up and completing these jobs). Smaller generations require fewer ATG platform resources, but increase the number of Endeca 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:

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
```

excludedCategoryIds

A list of the IDs of categories that dimension values should not be created for. For example, Commerce Reference Store has root categories that are not displayed on the site and therefore should not be represented by dimension values. Commerce Reference Store excludes these categories by setting excludedCategoryIds to:

```
excludedCategoryIds=rootCategory,homeStoreRootCategory
```

excludedItemsCategoryIds

A list of the IDs of the categories whose child products and skus are excluded from indexing. As with the categories specified in the excludedCategoryIds property, no dimension values are created for the categories specified in excludedItemsCategoryIds.

By default, the excludedItemsCategoryIds property of CategoryTreeService is configured to get its value from the excludedItemsAncestorIds property of the ProductCatalogOutputConfig component:

```
excludedItemsCategoryIds^=\
/atg/commerce/search/ProductCatalogOutputConfig.excludedItemsAncestorIds
```

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. If this property is set, `CategoryTreeService` uses the catalogs associated with the specified sites for computing paths. By default, `sitesForCatalogs` is set to:

```
sitesForCatalogs^=\
  /atg/commerce/search/ProductCatalogOutputConfig.sitesToIndex
```

If `sitesForCatalogs` is null, `CategoryTreeService` 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 repository item-type hierarchy. Set by default to:

```
dimensionName=item.type
```

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 26) for more information.) Set by default to:

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

SchemaExporter

The following are 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.RecordStoreDocumentSubmitter`) to use to submit records to the CAS schema record store. (See Document Submitter Components (page 26) for more information.) Set by default to:

```
documentSubmitter=/atg/endeca/index/SchemaDocumentSubmitter
```

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.

By default, `dimensionNameProviders` is set to:

```
dimensionNameProviders+=RepositoryTypeDimensionExporter
```

When an indexing job is run, `RepositoryTypeDimensionExporter` outputs dimension value records for the `item.type` dimension from the `product.type`, `sku.type`, and other item-type attributes. When `SchemaExporter` outputs schema records, it checks with `RepositoryTypeDimensionExporter` to determine these associations, and outputs a schema record that creates references from these attribute names to the dimension name. For example:

```
<RECORD>
   <PROP NAME="attribute.name">
     <PVAL>item.type</PVAL>
   </PROP>
   <PROP NAME="attribute.source_name">
     <PVAL>product.type</PVAL>
     <PVAL>sku.type</PVAL>
     <PVAL>product.manufacturer.type</PVAL>
     <PVAL>allAncestors.type</PVAL>
   </PROP>
   <PROP NAME="attribute.display_name">
     <PVAL>item.type</PVAL>
   </PROP>
   <PROP NAME="attribute.property.data_type">
     <PVAL>string</PVAL>
   </PROP>
   <PROP NAME="attribute.type">
     <PVAL>dimension</PVAL>
   </PROP>
 </RECORD>
```

Document Submitter Components

As described above, each component that generates records has a `documentSubmitter` property that is set by default to a component of class `atg.endeca.index.RecordStoreDocumentSubmitter`. The ATG-Endeca integration includes the following components of this class:

- `/atg/endeca/index/DataDocumentSubmitter`
- `/atg/endeca/index/DimensionDocumentSubmitter`
- `/atg/endeca/index/SchemaDocumentSubmitter`

The following are key properties of this class.

endecaDataStoreType

The type of the record store to submit to. Can be set to `data`, `dimval`, or `schema`. The following table shows the default setting for each component:

`DataDocumentSubmitter`	`data`
`DimensionDocumentSubmitter`	`dimval`
`SchemaDocumentSubmitter`	`schema`

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 document submitters import classes from the Endeca `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.

Directing Output to Files

To help optimize and debug your output, you can have the generated records sent to files rather than to the Endeca 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. Note that a separate file is created for each record generated.

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 Endeca Application Controller (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.ApplicationConfiguration` used to configure indexing settings for the integration. The default setting:

```
applicationConfiguration=/atg/endeca/index/IndexApplicationConfiguration
```

ProductCatalogSimpleIndexingAdmin

The `/atg/commerce/endeca/index/ProductCatalogSimpleIndexingAdmin` component (of class `atg.endeca.index.admin.SimpleIndexingAdmin`) manages the process of generating records, submitting

them to Oracle Endeca Commerce, and invoking indexing. The page for this component in the Component Browser of the ATG 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 Endeca 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 30) for more information.

Queueing Indexing Jobs

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

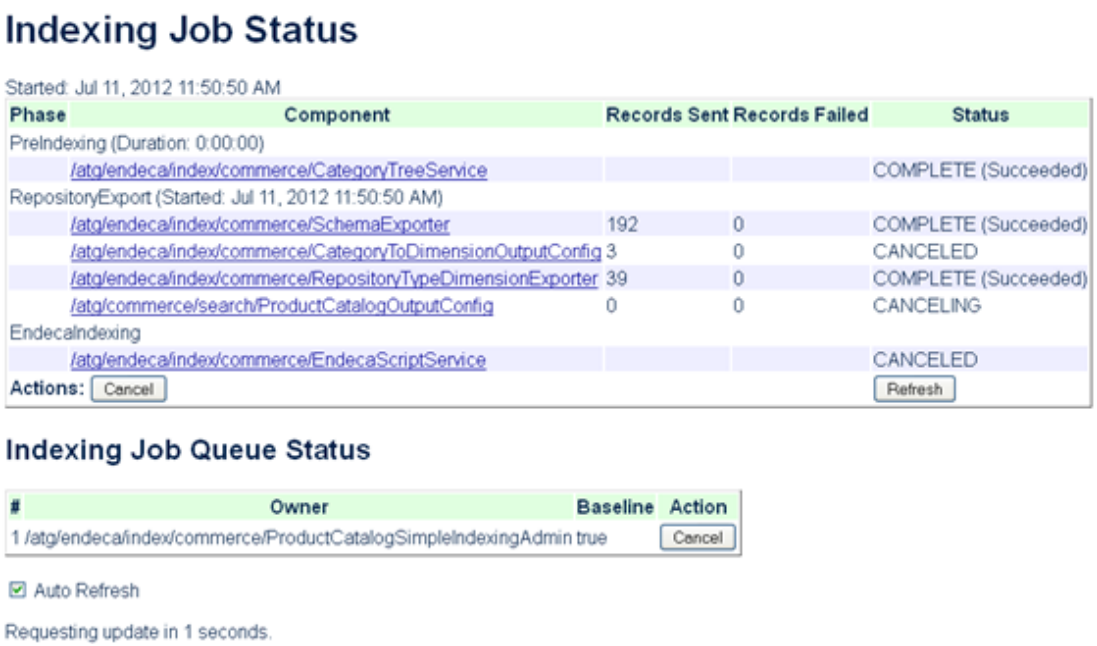
- If there is currently another indexing job running
- If an ATG Content Administration deployment is in progress

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 ATG 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:



Content Administration Components

If your ATG 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`), not the versioned repository.

The ATG-Endeca 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 EndecaIndexingOutputConfig components on the ATG Content Administration server with the name of the deployment target. You typically set the targetName property of the /atg/commerce/search/ProductCatalogOutputConfig and /atg/commerce/endeca/index/CategoryToDimensionOutputConfig components to Production:

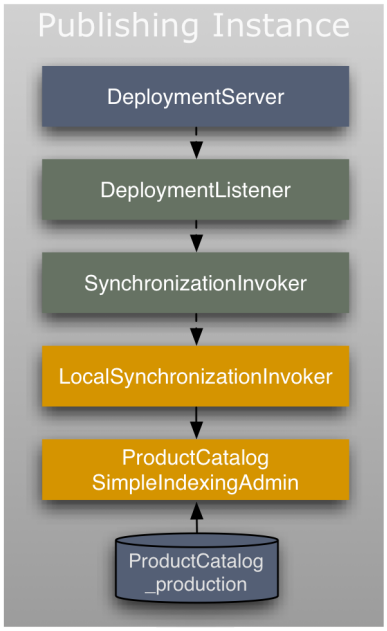
```
targetName=Production
```

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:

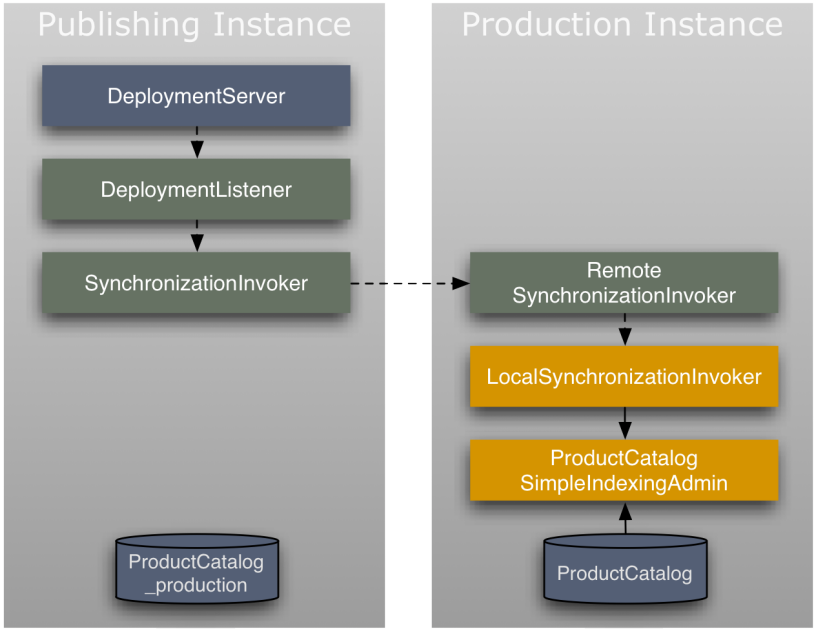


Enabling Remote Indexing

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 ATG Dynamo Server Admin to view records without submitting them to Oracle Endeca Commerce. To do this, access the page for a component that generates records and follow the instructions below.

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 ATG repository-item properties are renamed in the Endeca records, based on the values of various `EndecaIndexingOutputConfig` properties. (See the EndecaIndexingOutputConfig Components (page 18) 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.

4 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 ATG 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 an ATG 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 top-level `item` element has the `is-document` attribute 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 Oracle Endeca Commerce 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 (e.g., `brand` in the example) are repeated in each record.

When you configure the ATG-Endeca 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 ATG Modules (page 7) for information about these modules.

In addition to the properties you specify in the definition file, the output records also automatically include a few special properties. These properties provide information that identifies the repository items represented in the record: `repositoryId`, `repository.repositoryName`, and `itemDescriptor.itemDescriptorName`.

The output also includes a `url` property and a `baseUrl` property, which each contain the URL representing this repository item. The difference between these properties is that if a `VariantProducer` is used to generate multiple records from the same repository item, the `url` property for each record will include unique query parameters to distinguish the record from the others. The `baseUrl` property, which omits the query parameters, will be the same for each record.

Specifying Endeca Schema Attributes

You use various attributes of the `property` element to specify the way ATG properties should be treated in the Endeca MDEX. The `SchemaExporter` component then uses the values of these attributes in the schema records 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 an Endeca 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 an ATG property, by default Oracle Endeca Commerce 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 ATG-Endeca integration. This is done by the `CategoryToDimensionOutputConfig` (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 Endeca 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 ATG Web Commerce 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 `IndexingOutputConfig` 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 45) for more information about custom property accessors.

Suppressing Properties

The output record automatically includes certain standard JavaBean properties of the `RepositoryItem` object. These properties provide information that identifies the repository items represented in the record, and they are indicated in the definition file by a dollar-sign ($) prefix: `$repositoryId`, `$repository.repositoryName`, and `$itemDescriptor.itemDescriptorName`. (The dollar-signs are removed by default in the output records, because Endeca property names cannot include them.)

You may want to return these properties in search results, to enable accessing the indexed repository and repository items in page code. Typically you would do this for the document-level item type. For other item types, you may not need these properties. If you do not need these properties, it is a good idea to exclude them from the index, as they may significantly increase the size of the index.

To suppress one of these properties, specify the property in the indexing definition file with the `suppress` attribute. For example:

```
<item property-name="parentCategories" is-document="false">
  <properties>
    <property name="$repositoryId" suppress="true"/>
    <property name="$repository.repositoryName" suppress="true"/>
    <property name="$itemDescriptor.itemDescriptorName" suppress="true"/>
  </properties>
</item>
```

Including the siteIds Property

If you are using Oracle ATG Web Commerce multisite support, many of the item types in the catalog repository have a `siteIds` property whose value is a comma-separated list of the sites an item appears on. For example, if you have three sites, A, B, and C, and a certain product is available on sites A and C (but not B), the value of the product's `siteIds` property would be `siteA`, `siteC` (assuming those are the site IDs).

The siteIds properties in the catalog repository are defined as context membership properties. For the document-level item type, the record output includes a special siteId property representing the repository item's context membership property. (The output property is always named siteId, regardless of the actual name of the context membership property.) The records include a separate entry for each site listed in the context membership property.

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 product's siteIds property is siteA, siteC, siteD, but sitesToIndex list only sites C and D, the record will not include an entry for site A. If an item's siteIds property is null, or if it lists only sites that are not listed in the sitesToIndex property, no record is generated for the item.

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 18) 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. For example, you may want to normalize values (e.g., 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 (e.g., 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 `IndexingOutputConfig` 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 47) 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, Rose 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="Rose" output="Red"/>
  <translate locale="fr_FR" input="Rose" 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*.

5 Customizing the Output Records

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

In addition to the classes described here, the ATG-Endeca integration includes property accessors and variant producers for accessing price data in price lists. See the *Handling Price Lists* (page 97) 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 `IndexingOutputConfig` 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 48).

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 ATG-Endeca 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:

Property	Property
`RootCatalogPropertyAccessor`	`dimval.prop.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., `rootCategory.cat10016.cat10014`).
`QualifiedSpecPropertyAccessor`	`dimval.qualified_spec` -- A qualified identifier for the dimension value consisting of the `dimval.dimension_spec` value and the `dimval.spec` value (e.g., `product.category:rootCategory.cat10016.cat10014`).
`ParentSpecPropertyAccessor`	`dimval.parent_spec` -- A reference to the category's parent category (e.g., `rootCategory.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 `IndexingOutputConfig` 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 ATG Web 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 ATG 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 ATG-Endeca 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 12) 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 ATG-Endeca 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 a separate record for each site, you can do so by using the `/atg/search/repository/UniqueSiteVariantProducer` component. This component is of class `atg.commerce.search.producer.UniqueSiteVariantProducer`, 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
```

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 `IndexingOutputConfig` 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 just 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 `IndexingOutputConfig` 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.

6 Indexing Multiple Languages

If your ATG sites include data in more than one language, there are two options for how to index this data in Oracle Endeca Commerce:

- Index each language in a separate MDEX
- Index all of the languages in a single MDEX

This chapter discusses how to configure the ATG indexing components to support each option. It includes these sections:

There are also differences in how querying works, depending on which indexing option you choose. See the *Query Integration* (page 61) chapter for information.

Specifying the Locales

To generate records in multiple languages, you specify the locales by setting the `locales` property of the `/atg/endeca/ApplicationConfiguration` component. For example:

```
locales=en_US,fr_FR
```

Using a Separate MDEX for Each Language

If you use a separate MDEX for each language, you must create a separate EAC application and a corresponding set of record stores for each MDEX. By default, the name of each application is the value of the `baseApplicationName` property of the `/atg/endeca/ApplicationConfiguration` component plus the two-letter code for the application's language.

So, for example, if the `baseApplicationName` property is set to `ATG` (the default), and catalog data is in English, German, and Spanish, the three applications would be named `ATGen`, `ATGde`, and `ATGes`.

If you do not want to use the default application naming convention, use the `keyToApplicationName` property of the `/atg/endeca/ApplicationConfiguration` component to map the application keys to the names of the applications. For example:

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

The record stores for an EAC application use the following naming convention:

`application-name_language-code_record-store-type`

So for the `MySpanishApp` application, the record stores are named `MySpanishApp_es_data`, `MySpanishApp_es_dimvals`, and `MySpanishApp_es_schema`.

For more information about application keys and application naming, see Creating the Endeca Applications (page 2).

Using a Single MDEX for all Languages

If you use the same MDEX for all languages, you must create a single EAC application and a single set of record stores. In this case, the default name of the application is the value of the `baseApplicationName` property of the `/atg/endeca/ApplicationConfiguration` component plus the language code for the default language of the application. So if your catalog data is in English, German, and Spanish, and you want to index all languages in a single MDEX with English as the default language, the default application name would be `ATGen` (assuming the `baseApplicationName` property is set to `ATG`). The names of the record stores would be `ATGen_en_data`, `ATGen_en_dimvals`, and `ATGen_en_schema`.

You specify the default language by setting the `defaultLanguageForApplications` property of the `/atg/endeca/ApplicationConfiguration` component to the two-letter code for the language. For example:

```
defaultLanguageForApplications=en
```

If you do not want to use the default application naming convention, use the `keyToApplicationName` property of the `/atg/endeca/ApplicationConfiguration` component to map the `default` application key to the name of the application. For example:

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

For more information about application keys and application naming, see Creating the Endeca Applications (page 2).

Output Records

When you index multiple languages in a single MDEX, the schema records generated are the same as the records that would be generated in the multiple-MDEX case for the first locale listed in the `/atg/endeca/ApplicationConfiguration` component's `locales` property. The data records generated include separate records for each of the listed locales, with each data record including a `product.language` property that 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.

The dimension value records consist of the same set of records that would be generated for each language in the multiple-MDEX case, but 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 `dimval.prop.displayName_`*`language-code`*, where *`language-code`* is the two-letter language code associated with one of the specified locales. For example:

```
<PROP NAME="dimval.prop.displayName_en">
  <PVAL>Product</PVAL>
</PROP>
<PROP NAME="dimval.prop.displayName_de">
  <PVAL>Produkt</PVAL>
</PROP>
<PROP NAME="dimval.prop.displayName_es">
  <PVAL>Producto</PVAL>
</PROP>
```

If the `multiLanguageSynonyms` property of the `RepositoryTypeDimensionExporter` component is set to `true`, then additional Endeca 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>
```

7 Query Integration

The Oracle ATG Platform provides two options for querying the Oracle Endeca Assembler and MDEX engine:

- Invoking the Assembler via a servlet as part of Oracle ATG'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:

ContentItem, ContentInclude, and ContentSlotConfig 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 Pages section of Experience Manager. Invoking the Assembler for a page request is also referred to as "invoking the Assembler with a `ContentInclude`." The URI for a page request must begin with a `/pages` prefix, for example, `/pages/browse`. Endeca uses the `/pages` prefix to distinguish page requests from content folder requests.

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. To return to our original example, assume a `browse` page exists in the Experience Manager Pages definitions. Passing in a `/pages/browse` path will match this `browse` page. Passing in a `/pages/browse/seo/url` path will also match this page because the deepest matching path the handler can find for `/pages/browse/seo/url` is `/pages/browse` (this example assumes that a `browse/seo/url` page does not exist in Experience Manager).

`ContentSlotConfig` is used to request a 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 collection. The handler for `ContentSlotConfig`, uses these parameters to form a content trigger request that fetches the top item (or items) from the collection by priority. The Assembler then processes the content items from the collection and returns them as part of the response for rendering.

The remainder of this chapter makes a distinction between `ContentInclude` and `ContentSlotConfig` when necessary. When the distinction is not required, the more general `ContentItem` is used.

Note: For more information on the `ContentInclude` and `ContentSlotConfig` components and their handlers, refer to the *Assembler Application Developer's Guide* in the Oracle Endeca Commerce documentation.

Invoking the Assembler in the Request Handling Pipeline

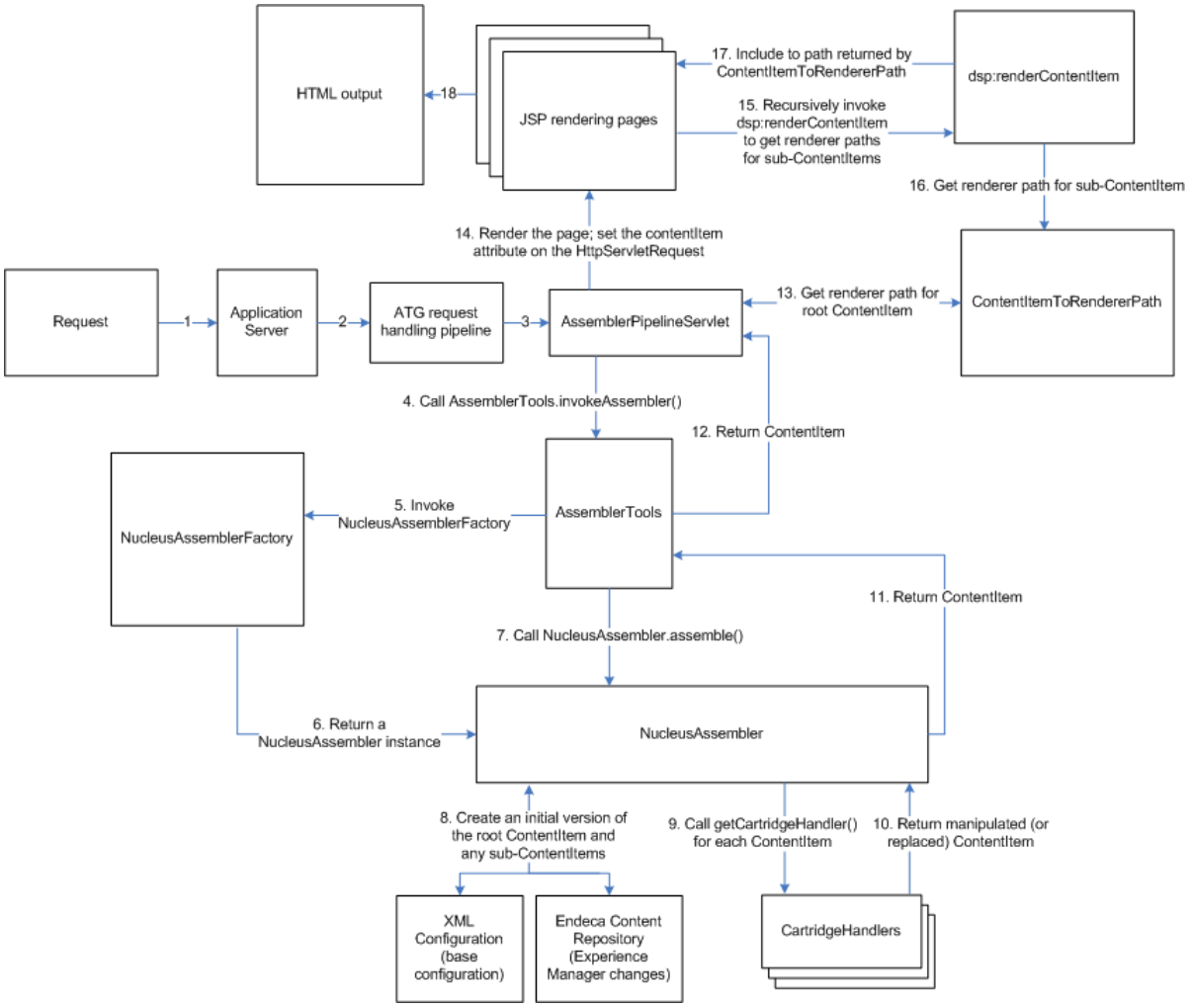
In this option, the Assembler is invoked early in the page rendering process as part of the ATG request handling pipeline. This option is appropriate when the bulk of a page's content is served by the Assembler and this guide refers to these pages as "Assembler-driven pages."

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. Endeca's Experience Manager tool, which works hand in hand 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.

Using a JSP Renderer to Render Content

The content returned to the client browser can take several forms: JSP, XML, or JSON. 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 ATG request handling pipeline.
3. The ATG request handling pipeline does some preliminary work, such as setting up the profile and determining which 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 an appropriate request `ContentItem`. Then, `AssemblerPipelineServlet` calls the

invokeAssembler() method on the /atg/endeca/assembler/AssemblerTools component and passes it the request ContentItem.

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 ContentItem.

8. The NucleusAssembler instance assembles the correct content for the request. Content, in Endeca terms, corresponds to a set of cartridges and their associated data. The NucleusAssembler instance starts with the data in the Endeca Experience Manager cartridge configuration files and then modifies that data with information stored in the Endeca Content Repository (that is, changes made and saved via the Experience Manager UI). The assembled content takes the form of a response ContentItem that consists of a root ContentItem which may have sub-ContentItem objects as attributes. This ContentItem hierarchy corresponds to the root cartridge and any sub-cartridges that were used to create the returned content.

9. The NucleusAssembler instance recursively calls the NucleusAssembler.getCartridgehandler() method, passing in the ContentItem type, to retrieve the correct cartridge handlers for the root ContentItem and any of its sub-items.

10. The cartridge handlers get resolved and executed for the root ContentItem and its sub-items. The resulting root ContentItem is passed back to the NucleusAssembler Instance.

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

11. The NucleusAssembler instance returns the root ContentItem to AssemblerTools.

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

13. 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 84) for more details on how the renderer path is calculated.

14. 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

15. The JSP for the root ContentItem may also have to render sub-ContentItems. In this case, the JSP must include dsp:renderContentItem tags for the sub-ContentItems.

16. dsp:renderContentItem invokes ContentItemToRendererPath to retrieve the JSP renderer for the specified ContentItem. This process happens recursively until all sub-ContentItems are rendered.

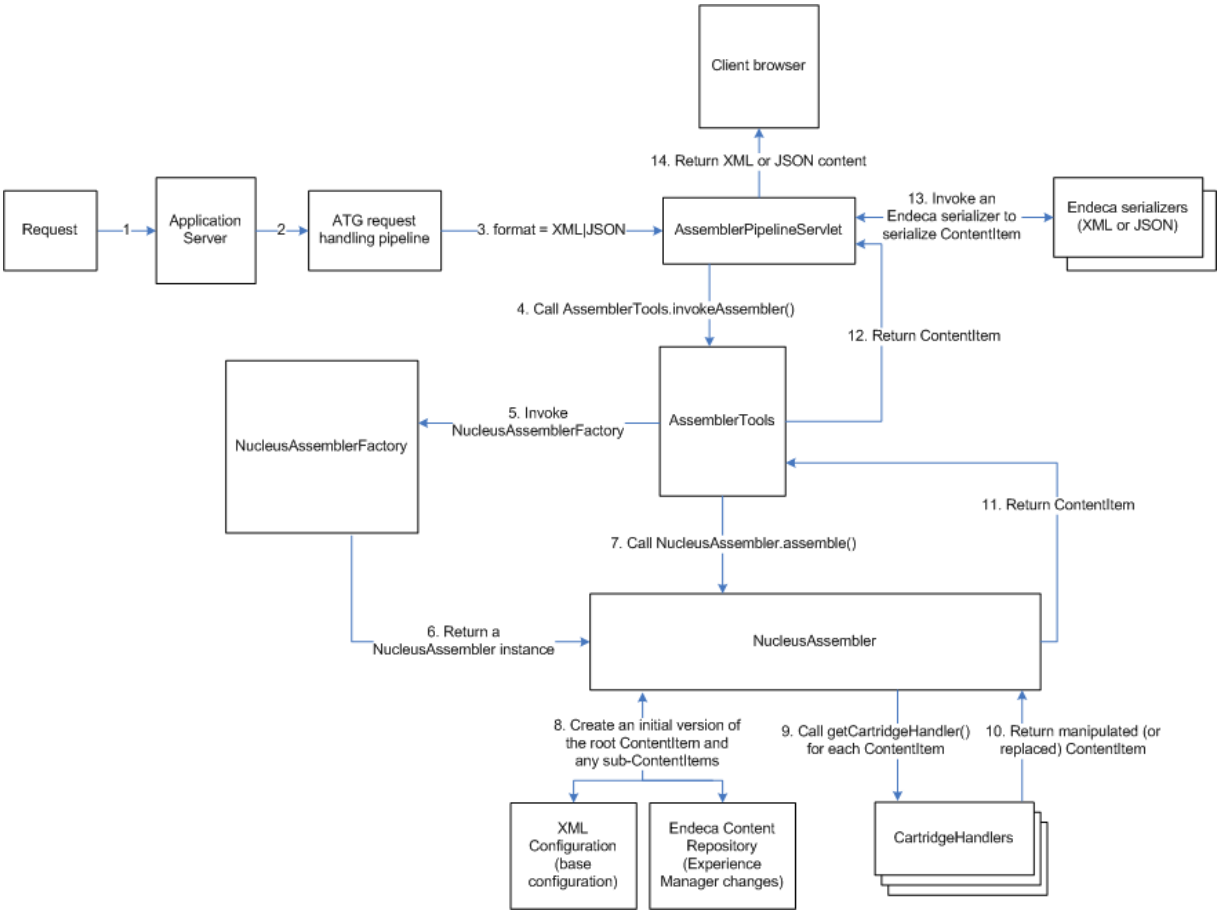
    The dsp:renderContentItem tag also sets the contentItem attribute on the HttpServletRequest, thereby making the current ContentItem available to the renderers; however, this value lasts only for the duration of the include so that after the include is done, the contentItem attribute's value returns to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

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

18.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
```

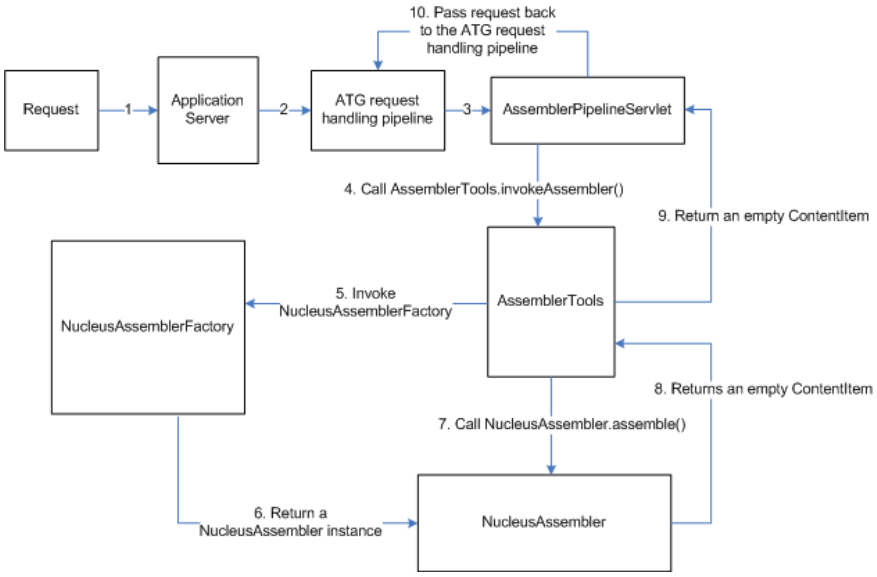
This example returns `json` for a page request:

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

If the request specifies a valid `format` parameter and 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. The `AssemblerPipelineServlet` component then returns the reformatted content to the client browser.

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 ATG request handling pipeline for further processing. This scenario is shown in the diagram below:

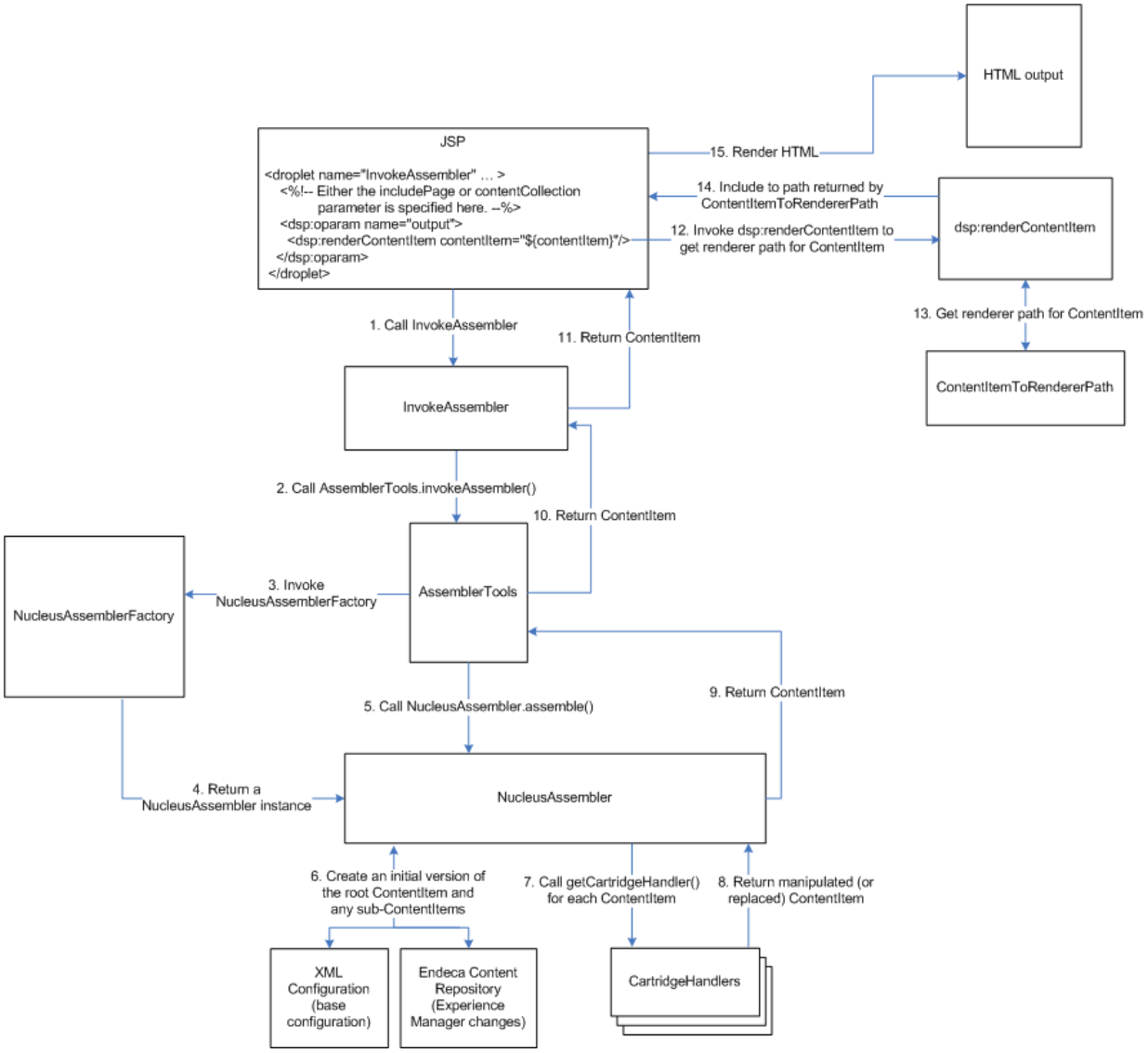


Note that you can configure an application to bypass the `AssemblerPipelineServlet` and avoid this scenario. For more information, see the AssemblerPipelineServlet (page 69) section.

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 "ATG-driven pages."

The request-handling architecture for an ATG-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 an Assembler content request, in the form of a `ContentItem`. `InvokeAssembler` then calls the `AssemblerTools.invokeAssembler()` method, passing in the `ContentItem`.

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 `ContentItem`.

6. The NucleusAssembler instance assembles the correct content for the request. Content, in Endeca terms, corresponds to a set of cartridges and their associated data. The NucleusAssembler instance starts with the data in the Endeca Experience Manager cartridge configuration files and then modifies that data with information stored in the Endeca Content Repository (that is, changes made and saved via the Experience Manager UI). The assembled content takes the form of a response ContentItem that consists of a root ContentItem which may have sub-ContentItem objects as attributes. This ContentItem hierarchy corresponds to the root cartridge and any sub-cartridges that were used to create the returned content.

7. The NucleusAssembler instance recursively calls the NucleusAssembler.getCartridgehandler() method, passing in the ContentItem type, to retrieve the correct cartridge handlers for the root ContentItem and any of its sub-items.

8. The cartridge handlers get resolved and executed for the root ContentItem and its sub-items. The resulting root ContentItem is passed back to the NucleusAssembler instance.

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

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

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

11. 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.

12. The ContentItemToRendererPath component returns the correct renderer for the ContentItem.

13. The JSP returned by ContentItemToRendererPath is included in the response.

14. The response is returned to the browser.

Choosing Between Pipeline Invocation and Servlet Bean Invocation

As you write your pages, you can choose to make a page Assembler-driven via pipeline invocation versus making it ATG-driven via servlet bean invocation. This decision is based on:

- The amount of the page's content that must be configurable by a merchandiser. Pages that must be heavily configurable by a merchandiser are good candidates for being Assembler-driven.

- The number of URLs on the resulting page that should be constructed as Endeca URLs. Pages that contain many URLs that will result in calls to the MDEX should be constructed by the Assembler, so that those URLs are properly formed. For example, the category page includes a facets rail on the left side that consists of links backed by Endeca URLs. These URLs should be constructed by the Assembler API.

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 ATG's 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 ATG 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 71) 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 must be transformed into a form that the `NucleusAssembler` class can accept. To do this, the `AssemblerPipelineServlet` component calls the

AssemblerTools.getContentPath() method to transform the page request URL into a URI and store it in a ContentInclude that can be passed to the NucleusAssembler class. The NucleusAssembler class can then match this URI to the URIs of the pages defined Experience Manager. See the AssemblerTools (page 72) section for specific details on how the URL transformation is done.

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/ component, which is part of the ATG 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 ATG 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 one of these parameters but they are mutually exclusive.

includePath
Use the includePath parameter for a page request. The path you specify must correspond to the name of a page in Experience Manager, with the addition of a /pages prefix. For example, to assemble content for a browse page, specify /pages/browse for the includePath (passing in a /browse path will not match because it is missing the /pages prefix).

InvokeAssembler parses the includePath into a ContentInclude component. This component contains a set of parameters, including the request URI, that is used to form a content request for the Assembler.

The includePath and contentCollection parameters are mutually exclusive but one of them must be passed when using the InvokeAssembler servlet bean.

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` parses the `contentCollection` into a `ContentSlotConfig` component. This component specifies a content folder and the number of content items to return from that collection (note, the number of items to return is specified using the `InvokeAssembler.ruleLimit` parameter, described next).

The `includePath` and `contentCollection` parameters are mutually exclusive but one of them must be passed when using the `InvokeAssembler` servlet bean.

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 three 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="/pages/browse"/>
  <dsp:oparam name="output">
    <dsp:getvalueof var="contentItem"
                    vartype="com.endeca.infront.assembler.ContentItem"
                    param="contentItem" />
  </dsp:oparam>
</dsp:droplet>
```

AssemblerTools

The `/atg/endeca/assembler/AssemblerTools` component provides commonly used functionality to other ATG-Endeca 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 the `AssemblerPipelineServlet` component by transforming the page request URL into a request `ContentItem`.

- 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 folder process. More details on the NucleusAssemblerFactory component can be found in the Querying the Assembler (page 80) section.

Transforming a Page Request URL for the AssemblerPipelineServlet

Note: This section describes transforming the URL for a page request into a request ContentItem when using the AssemblerPipelineServlet component only. Other mechanisms exist for creating the ContentItem when requesting a content folder or when using the InvokeAssembler servlet bean. See the Content folder Request Identification and Handling (page 70) and InvokeAssembler (page 71) sections, respectively, for more information on how those mechanisms work.

For page requests, the AssemblerTools.getContentPath() method transforms the request URL into a ContentItem URI. This URI tells the Assembler the path it should use to determine what content to assemble. getContentPath() takes into account several configurable properties when it calculates the URI. For example, if a request is made to http://localhost:8080/crs/storeus/browse/, getContentPath() does the following:

1. Gets the request URI using the atg.servlet.ServletUtil class. In this case, the request URI is:

   /crs/storeus/browse/

2. If the AssemblerTools.isRemoveSiteBaseURL() property is true, getContentPath() removes the site base URL (also known as the productionURL). In this example, the site base URL is /crs/storeus, so the modified URI is:

   /browse/

3. If AssemblerTools.isRemoveContextRoot() property is true and the site base URL has not been removed, getContentPath() removes the context root. In this case, getContentPath() has already removed the site base URL, so the URL remains as is:

   /browse/

4. Finally, getContentPathPrefix() inserts the content path prefix. This prefix can be passed in on the request, using the contentPrefix parameter. When getContentPathPrefix() executes, it first checks for the existence of the contentPrefix request parameter. If this parameter exists, its value is inserted at the beginning of the URI. If contentPrefix does not exist, getContentPathPrefix() invokes the AssemblerTools.isExperienceManager() method to determine if Experience Manager is in use. If Experience Manager is in use, isExperienceManger() returns AssemblerTools.assemblerSettings.defaultExperienceManagerPrefix, which defaults to /pages. If not, isExperienceManager() returns AssemblerTools.assemblerSettings.defaultGuidedSearchPrefix, which defaults to /services.

In this example, we assume that Experience Manager is in use, so the final content path URI is:

```
/pages/browse/
```

The resulting content path URI is used to construct a content item.

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>
```

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`. Used by the `AssemblerTools` component when creating the content path prefix.
- `defaultGuidedSearchPrefix`: Defaults to `/service`. Used by the `AssemblerTools` component when creating the content path prefix.
- `experienceManager`: Defaults to `true`. Used by the `AssemblerTools.isExperienceManager()` method to determine if Experience Manager is available.

Connecting to Endeca

Some cartridges need to communicate with the Endeca Workbench while others need to communicate directly with the MDEX engines to do their work. The ATG-Endeca integration includes a number of components to facilitate both types of communication.

AssemblerApplicationConfiguration Component

The atg.endeca.assembler.configuration.AssemblerApplicationConfiguration class provides a central place for calculating and storing Workbench and MDEX host and port information. This information complements the information contained in the /atg/endeca/ApplicationConfiguration component and allows the Assembler to communicate with Workbench applications and MDEX instances. The ATG-Endeca integration includes a component of the AssemblerApplicationConfiguration class, /atg/endeca/assembler/, that other components use to retrieve Workbench and MDEX connection details. This section provides information on how the AssemblerApplicationConfiguration component identifies the correct Workbench application and MDEX to connect to. The sections after provide details on the components that reference this information.

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 a particular Endeca application's connection to 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 an Endeca requirement that a globally-scoped com.endeca.infront.content.source.WorkbenchContentSource object be instantiated for each Endeca application in your environment before any content requests are made. Environments that have multiple Endeca applications, for example, to support multiple languages, will have multiple corresponding configurations running on the Workbench server and, therefore, 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_*Endeca-application-key*

Adding the *Endeca-application-key* to the end of the WorkbenchContentSource component name uniquely identifies the correct WorkbenchContentSource to use for each Endeca 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 Endeca application name, which it gets from the AssemblerApplicationConfiguration component's currentInitializingWorkbenchContentSourceApplicationName property.

Calculating Which MDEX to Use

The AssemblerApplicationConfiguration component is responsible for determining 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, can then refer to the AssemblerApplicationConfiguration 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 calculated.

Typically, if your application uses a single MDEX, you set the defaultMdexHostName and defaultMdexPort properties of the AssemblerApplicationConfiguration component to the host and port for that MDEX, for example:

```
defaultMdexHostName=localhost
defaultMdexPort=15000
```

The AssemblerApplicationConfiguration then uses the defaultMdexHostName and defaultMdexPort settings to set the currentMdexHostName and currentMdexPort properties.

If your application uses multiple MDEX engines, for example, one MDEX for each language, you must configure the applicationKeyToMdexHostAndPort property. This property contains a map where the keys identify each Endeca application and the values specify the host names and port numbers for the MDEX engines associated with each application. For example, if your environment has two Endeca applications to support two languages, English and German, the applicationKeyToMdexHostAndPort would be set as follows:

```
applicationKeyToMdexHostAndPort=\
en=host-for-English-MDEX:port-for-English-MDEX\
de=host-for-German-MDEX:port-for-German-MDEX
```

To calculate which MDEX host and port to use, the AssemblerApplicationConfiguration component retrieves the Endeca application key for the current request from the ApplicationConfiguration component, and then uses that key to retrieve the correct host and port values from the applicationKeyToMdexHostAndPort map. To enable the call to the ApplicationConfiguration component, the AssemblerApplicationConfiguration component includes the following required property:

```
applicationConfiguration=../ApplicationConfiguration
```

Note: For more details, on the ApplicationConfiguration component, see the Configuring the ApplicationConfiguration Component (page 4) section.

On a final note, the /atg/endeca/assembler/AssemblerApplicationConfiguration .defaultMdexHostName and /atg/endeca/assembler/AssemblerApplicationConfiguration .defaultMdexPort properties are commented out in the DAF.Endeca.Assembler module out of the box. If you use CIM to configure your application, it does not modify these settings, so you will need to configure them manually in your localconfig environment, using the guidance provided in this section.

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 Endeca Workbench

Oracle ATG Web Commerce has several components for creating a connection to the Workbench server. The Workbench connection components can vary depending on whether your environment has a single Endeca application or multiple applications (for example, to support multiple languages). Here is a brief overview of the process:

- On startup, the `/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource` component is instantiated. This component contains default Endeca application, host name, and port information for the Workbench server. The `DefaultWorkbenchContentSource` component gets its host and port values from the `AssemblerApplicationConfiguration.workbenchHostName` and `AssemblerApplicationConfiguration.workbenchPort` properties, respectively. It gets the Endeca application name from the `ApplicationConfiguration.defaultApplicationName` property.
- If the environment has more than one Endeca application (for example, because it supports multiple languages with one language per Endeca application), the `AssemblerApplicationConfiguration` component creates globally-scoped, `WorkbenchContentSource_`*`Endeca-application-key`* components for each Endeca application. Each component has a suffix that identifies which Endeca application the component is for, for example, `WorkbenchContentSource_en` and `WorkbenchContentSource_de`.
- 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` for the current request.
- If the `DefaultWorkbenchContentSource` is resolved, the host, port, and Endeca application defined by this component are used when connecting to the Workbench server.
- If the `PerApplicationWorkbenchContentSourceResolver` is resolved, the component relies on the `AssemblerApplicationConfiguration` to determine what the current Endeca application is and then it references the correct Endeca application-specific `WorkbenchContentSource` that the `AssemblerApplicationConfiguration` component has already created.

The remaining sections provide more details on the individual Workbench-related components.

WorkbenchContentSource

The `/atg/endeca/assembler/cartridge/manager/WorkbenchContentSource` component represents the current Endeca application's connection to the Workbench server. The `NucleusAssembler` class uses this component to connect to the correct Endeca application configuration running on the Workbench server.

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 Endeca application-specific `WorkbenchContentSource` to use, based on the current request. This default configuration is primarily intended to support environments that have multiple Endeca 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 always have a single Endeca 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-application environment because it avoids having to resolve the WorkbenchContentSource for every request. If you have a single-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, of class atg.endeca.assembler.content.ExtendedWorkbenchContentSource, is a globally-scoped component that includes default Endeca application, host, and port properties for connecting to the Workbench server. Out of the box, this property 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
```

In a single Endeca application environment, the DefaultWorkbenchContentSource component provides connection details to the single Endeca application running on the Workbench server that should be used for all requests. In a multi-application environment, this component provides connection details to a default Endeca application when the PerApplicationWorkbenchContentSourceResolver cannot resolve an Endeca application-specific WorkbenchContentSource.

The DefaultWorkbenchContentSource component has a set of properties that represent the constructor arguments that are used to create the WorkbenchContentSource. The DefaultWorkbenchContentSource component gets the values for these properties from the ApplicationConfiguration and AssemblerApplicationConfiguration components. It is the responsibility of these other two components to calculate the correct Endeca application and Workbench connection to use. The DefaultWorkbenchContentSource properties include:

- # Arg1 - Workbench app name: This property provides the first constructor argument for WorkbenchContentSource and it points to the Endeca application. The default property setting is:

  ```
  $constructor.param[1].value^=/atg/endeca/
  ApplicationConfiguration.defaultApplicationName
  ```

- # Arg3 - Workbench host: This property provides the third constructor argument for WorkbenchContentSource and it points to the host that the Workbench is installed on. The default property setting is:

  ```
  $constructor.param[3].value=../../
  AssemblerApplicationConfiguration.workbenchHostName
  ```

- # Arg 4 - Workbench port: This property provides the fourth constructor argument for WorkbenchContentSource and it points to the port that the Workbench is using. The default property setting is:

  ```
  $constructor.param[4].value^=../../AssemblerApplicationConfiguration.workbenchPort
  ```

PerApplicationWorkbenchContentSourceResolver

In an environment that has multiple Endeca applications, it is the /atg/endeca/assembler/cartridge/manager/PerApplicationWorkbenchContentSourceResolver component's responsibility to determine

the correct globally-scoped, Endeca application-specific WorkbenchContentSource component to use for the current request. This component also defines a default WorkbenchContentSource component to use if an Endeca 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 Endeca 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 Endeca application-specific version cannot be resolved. Defaults to /atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource.
- componentBasePath: The base path for the Endeca application-specific WorkbenchContentSource components. PerApplicationWorkbenchContentSourceResolver adds the Endeca 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 . . .
- useDefaultIfSingleApplication: Indicates that the PerApplicationWorkbenchContentSourceResolver should use the DefaultWorkbenchContentSource if there is only one Endeca application and avoid resolving an Endeca application-specific WorkbenchContentSource.

Manually Adding Application-specific WorkbenchContentSource Components

It is an Endeca requirement that the WorkbenchContentSource component used to communicate with any given Workbench 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 Endeca application on start up.

If the automatically-created WorkbenchContentSource components are not sufficient for your needs, you can manually create .properties files for other Endeca application-specific WorkbenchContentSource components, for example:

```
$basedOn=DefaultWorkbenchContentSource

# Arg1 - Workbench app name
$constructor.param[1].value=Endeca-application-name

# Arg3 - Workbench host
$constructor.param[3].value=Workbench-host-name

# Arg 4 - Workbench port
$constructor.param[4].value=Workbench-host-port
```

After creating the Endeca 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_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.

Note that, out of the box, the handlerMapping property provides override mappings to handlers for the default set of Endeca cartridges:

```
# Explicit cartridge handler mappings
handlerMapping=\
  DimensionSearchAutoSuggestItem=/atg/endeca/assembler/cartridge/handler/\
                                 DimensionSearchResults,\
  HorizontalRecordSpotlight=/atg/endeca/assembler/cartridge/handler/\
                           RecordSpotlight,\
  ContentSlotHeader=/atg/endeca/assembler/cartridge/handler/ContentSlot,\
  ContentSlotSecondary=/atg/endeca/assembler/cartridge/handler/ContentSlot,\
  ContentSlotMain=/atg/endeca/assembler/cartridge/handler/ContentSlot,\
  PageSlot=/atg/endeca/assembler/cartridge/handler/ContentSlot
```

Cartridge Handlers and Their Supporting Components

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 URLs

If a cartridge provides links to another Endeca navigation or record state, the URL path for each link is provided as an action string in the response `ContentItem`. Two components, `BasicUrlFormatter` and `DefaultActionPathProvider`, assist the cartridges in forming action strings. This section provides some details on both.

BasicUrlFormatter

The `/atg/endeca/url/basic/BasicUrlFormatter` component is of class `com.endeca.soleng.urlformatter.basic.BasicUrlFormatter`. This class is responsible for serializing action strings from a navigation state, for example, `?N`. It includes properties such as `defaultEncoding` and `prependQuestionMarks` that control how the strings 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 *Assembler Application Developer's Guide* in the Oracle Endeca Commerce documentation.

DefaultActionPathProvider

The `/atg/endeca/assembler/cartridge/manager/DefaultActionPathProvider` component, of class `atg.endeca.assembler.navigation.DefaultActionPathProvider`, creates the *action path* portion of the action strings that are stored in `ContentItem` objects. The action path is defined as the combination of the site root path and the content path. For example, in the link below:

```
/pages/browse?N=4294967263
```

The site root path is `/pages` and the content path is `/browse` (the remainder of the URL represents the query parameters that define the request). The `DefaultActionPathProvider` class generates both the site root and the content path values to be used in the action string. To do so, the `DefaultActionPathProvider` class implements the `com.endeca.infront.navigation.url.ActionPathProvider` interface and its four methods:

- `getDefaultNavigationActionContentPath()`: Returns the content path for a navigation request.
- `getDefaultNavigationActionSiteRootPath()`: Returns the site root path for a navigation request.
- `getDefaultRecordActionContentPath()`: Returns the content path for a record request.
- `getDefaultRecordActionSiteRootPath()`: Returns the site root path for a record request.

The `DefaultActionPathProvider` component also has the following properties that support action 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/brand action path can be replaced with a /pages/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 like 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 default site root paths are defined by this component's properties. 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 application. Defaults to /atg/endeca/assembler/cartridge/manager/WorkbenchContentSource. See Connecting to Endeca (page 74) for details on this component.

Calculating the Content Path

To calculate the content path for a navigation request, the getDefaultnavigationActionContentPath() method is invoked. This method, in turn, 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 the defaultGuidedSearchNavigationActionPath property, which defaults to /guidedsearch.

To calculate the content path to use for navigation requests when Experience Manager is in use, the DefaultActionPathProvider component checks the current request to determine if it contains an Endeca-specific URL. If it does, the DefaultActionPathProvider component extracts the entire action path from the URL 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 portion from the current request's action path. If it cannot resolve a content path from either the current request 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 requests when Experience Manager is in use is very similar to that for navigation requests. The getDefaultRecordActionContentPath() method is invoked and it performs the same URL checking, extraction, and comparison process, however, it uses the recordActionUriMap property for the lookup instead. Also, if a content path cannot be resolved from either the current request or the navigationActionUriMap, this method returns the value specified in the DefaultActionPathProvider.defaultExperienceManagerNavigationActionPath property, which defaults to /product.

Calculating the Site Root Path

To calculate the site root path for a navigation request, the getDefaultNavigationActionSiteRootPath() method is invoked. First, this method checks the current request to determine if it contains an Endeca-specific URL. If it does, the DefaultActionPathProvider component extracts the entire action path from the URL and looks for a match in the keys of its navigationActionUriMap property. If a match is found, the DefaultActionPathProvider component returns the site root portion of the matching entry's value. If no match is found, the DefaultActionPathProvider component returns the site root portion from the current request's action path.

If it cannot resolve a content path from either the current request or the navigationActionUriMap, 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 requests is very similar to that for navigation requests. The getDefaultRecordActionSiteRootPath() method is invoked. This method performs the same URL checking, extraction and comparison process, 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 either the current request or the recordActionUriMap is the same; a call is made to the AssemblerTools.assemblerSettings() method to retrieve the default site root prefix.

Also, if the DefaultActionPathProvider component cannot resolve a content path from either the current request or the navigationActionUriMap, it returns the value specified in its defaultExperienceManagerNavigationActionPath property, which defaults to /product.

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 action path 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 action 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="/pages/mobile/browse"/>
    <dsp:oparam name="output">

      <dsp:getvalueof var="contentItem"
                      vartype="com.endeca.infront.assembler.ContentItem"
                      param="contentItem" />

    </dsp:oparam>

  </dsp:droplet>
```

Retrieving Renderers

The ATG 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 ATG 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 ATG'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 Endeca Workbench to work in an integrated ATG-Endeca environment, you may have to do some additional configuration on the ATG side. Specifically, keyword redirects that point to servers other than the one where the ATG 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.

8 Content Promotion

Oracle Endeca Commerce provides two methods for promoting content from the Endeca Workbench to the production environment: a direct method, in which data is transferred via remote calls, and a file-based method, in which data is published to the file system and retrieved from there.

By default, the ATG-Endeca integration uses the direct method of content promotion. This chapter discusses how to modify the configuration to support file-based content promotion. The default Endeca deployment template assumes the use of file-based content promotion, so if you are using that deployment template, you should configure file-based content promotion as described in this chapter. Note, however, that the deployment template used for Commerce Reference Store assumes the use of the direct method.

For more information about the content promotion methods and how to determine which method to use, see the *Oracle Endeca Commerce Administrator*.

Configuring Content Promotion

The Endeca `WorkbenchContentSource` uses a *store factory* to retrieve promoted content. There are two store factory types: `com.endeca.infront.content.source.EcrStoreFactory`, which supports the direct method, and `com.endeca.infront.content.source.FileStoreFactory`, which supports the file-based method. If no store factory is specified, the `WorkbenchContentSource` creates an instance of `EcrStoreFactory`.

This section describes how to configure your ATG server instance to use the file-based promote content method. The configuration required differs depending on whether your Endeca environment uses a single MDEX engine or multiple MDEX engines.

Single-MDEX Environment

The ATG-Endeca integration includes the `/atg/endeca/assembler/cartridge/manager/DefaultFileStoreFactory` component, which is of class `atg.endeca.assembler.content.ExtendedFileStoreFactory`. (This class extends the Endeca `FileStoreFactory` class.) `ExtendedFileStoreFactory` has a `configurationPath` property for specifying the directory to retrieve the promoted content from.

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), you can enable file-based content promotion by performing the following steps:

1. Set the `storeFactory` property of the `/atg/endeca/assembler/cartridge/manager/DefaultWorkbenchContentSource` component to the `DefaultFileStoreFactory` component:

```
storeFactory=\
/atg/endeca/assembler/cartridge/manager/DefaultFileStoreFactory
```

2. Set the `storeFactory` property of the `/atg/endeca/assembler/admin/EndecaAdministrationService` component to the `DefaultFileStoreFactory` component:

   ```
   storeFactory=\
   /atg/endeca/assembler/cartridge/manager/DefaultFileStoreFactory
   ```

3. Set the `configurationPath` property of the `/atg/endeca/assembler/cartridge/manager/DefaultFileStoreFactory` to the file-system pathname of the directory to retrieve promoted content from. For example:

   ```
   configurationPath=\
   ToolsAndFrameworks/11.0.0/server/workspace/state/repository/ATG
   ```

Multiple-MDEX Environment

If your environment includes multiple MDEX engines (for example, multiple languages with a separate MDEX per language), configuring file-based content promotion requires a few more steps than in a single-MDEX environment.

As discussed in the *Query Integration* (page 61) chapter, in a multiple-MDEX environment, the ATG-Endeca integration uses a separate `WorkbenchContentSource` for each EAC application. In order to use file-based content promotion, each `WorkbenchContentSource` requires a separate instance of `FileStoreFactory` to retrieve the appropriate Experience Manager 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 file-based content promotion 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 performed the following steps to set up a multiple-MDEX environment:

1. Create one EAC application per language (for example, `ATGen`, `ATGes`, and `ATGde`).
2. Modify the `/atg/endeca/ApplicationConfiguration` component in the local server configuration. Set the `defaultLanguageForApplications` to null, and set the `keyToApplicationName` property to create a mapping of application keys to application names. For example:

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

```
en=ATGen,\
de=ATGde,\
es=ATGes
```

3. Modify the /atg/endeca/assembler/AssemblerApplicationConfiguration component in the local server configuration to set the applicationKeyToMdexHostAndPort property to create a mapping of application keys to hostname/port combinations. For example:

```
applicationKeyToMdexHostAndPort=\
en=localhost:15000,\
de=localhost:16000,\
es=localhost:17000
```

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 is configured to point to the /atg/endeca/assembler/cartridge/manager/PrototypeFileStoreFactory component. 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/11.0.0/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/11.0.0/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.

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
   ```

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/11.0.0/server/workspace/state/repository/ATGde
```

2. 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
   ```

3. 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
   ```

4. 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.

9 Record Filtering

Endeca provides a mechanism for filtering the records returned by a query, based on the values of record properties. For example, for a multisite application, you can use record filters to control which sites a query returns results for. To return results for only a single site, you use a filter to exclude all records except the ones that include a `product.siteId` property whose value is the ID of the desired site.

This chapter discusses ATG classes you can use to build and apply Endeca record filters. It includes these sections:

RecordFilterBuilder Interface and Implementing Classes

The ATG-Endeca integration includes the `atg.endeca.assembler.navigation.filter.RecordFilterBuilder` interface. Classes that build Endeca record filters implement this interface. The `RecordFilterBuilder` interface includes a `buildRecordFilter()` method that is responsible for building the actual record filter.

The ATG-Endeca integration includes several classes that implement the `RecordFilterBuilder` interface:

- `SiteFilterBuilder`
- `LanguageFilterBuilder`
- `CatalogFilterBuilder`
- `PriceListPairFilterBuilder`

The first three of these classes are described below. See the *Handling Price Lists* (page 97) chapter for information about the `PriceListPairFilterBuilder` class.

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.) SiteFilterBuilder has a number of properties that it uses to determine which sites to include when it constructs the filter:

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 Endeca 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.

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 Endeca 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 46) 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` to use 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 Endeca records to use for filtering. This is typically set to:

```
catalogIdPropertyName=product.catalogId
```

Enabling Record Filter Builder Components

The ATG-Endeca integration includes several record filter builder components:

```
/atg/endeca/assembler/cartridge/manager/filter/SiteFilterBuilder
/atg/endeca/assembler/cartridge/manager/filter/LanguageFilterBuilder
/atg/endeca/assembler/cartridge/manager/filter/CatalogFilterBuilder
/atg/endeca/assembler/cartridge/manager/filter/PriceListPairFilterBuilder
```

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
```

10 Handling Price Lists

If your application stores prices in product or SKU properties, indexing price data and accessing it on site pages is handled much like other properties, such as color or brand. If prices are stored in price lists, however, additional mechanisms are required to index the price data and access it on sites.

This chapter describes how the ATG-Endeca integration handles price data in price lists. It includes these sections:

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 ATG-Endeca 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.

When the ATG-Endeca 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 ATG-Endeca 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 ATG-Endeca 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, `PriceListVariantProducer` 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 ATG Commerce 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
```

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 ATG-Endeca integration includes a component of this class, `/atg/endeca/assembler/cartridge/manager/filter/PriceListPairFilterBuilder`.

The name of the price list pair property in Endeca records to use for filtering is specified through the `priceListPairPropertyName` property. This is typically set to:

```
priceListPairPropertyName=product.priceListPair
```

See the *Record Filtering* (page 93) chapter for more information about configuring and using record filters.

11 Dimension Value Caching

This chapter discusses dimension value caching, which the ATG-Endeca integration uses to map ATG repository items to the Endeca 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 ATG-Endeca integration involves treating product categories both as `category` items in the ATG product catalog repository and as Endeca 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 ATG-Endeca integration maintains a cache that maps each ATG category ID to the equivalent Endeca `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 ATG-driven page to an Endeca-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 ATG 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 Endeca 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 class `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 ATG 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 ATG repository ID for the category; e.g., `cat50087`
- `url` -- the Endeca 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`.

navigationStateThe component representing the Endeca `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 61) 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 Endeca-driven category page. For example:

```
<dsp:a page="${categoryCacheEntry.url}">
  <dsp:valueof value="${categoryDisplayName}">
    <fmt:message key="common.categoryNameDefault" />
  </dsp:valueof>
</dsp:a>
```

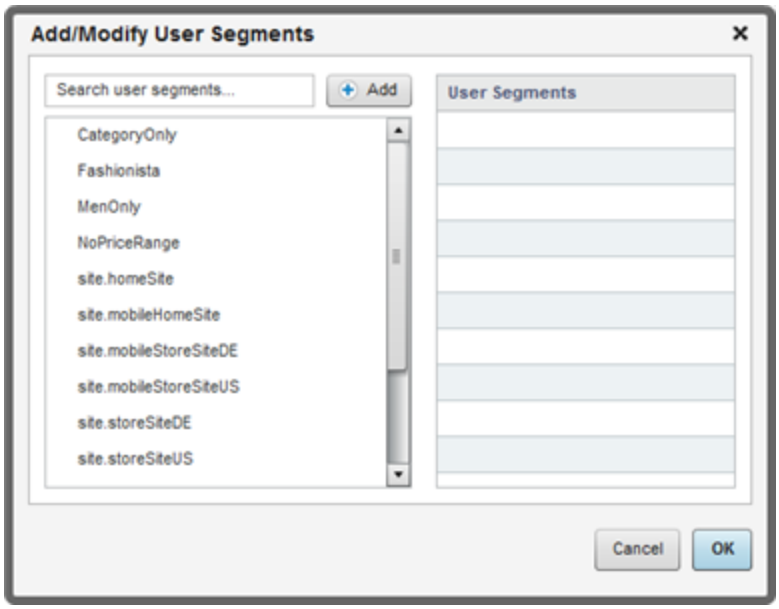
12 User Segment Sharing

This chapter discusses the user segment sharing feature included with the ATG-Endeca integration. This feature allows a content administrator to choose an ATG 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 included with the ATG-Endeca integration automatically populates the Add/Modify User Segments dialog box in Experience Manager with a list of ATG user segments. 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 on both the ATG side, in the Business Control Center, and on the Endeca side, 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. (There are some exceptions to this rule and Commerce Reference Store uses them. See *Using Sites and Site Groups as Triggers in Experience Manager* in the *Commerce Reference Store Overview* for more information.)

User segment sharing is a one-way relationship. When configured to do so, ATG user segments created in the Business Control Center are shared with Experience Manager. However, Endeca user segments created in Workbench are not shared with the Business Control Center.

Configuring User Segment Sharing

This section provides details on the configuration required for user segment sharing.

ATG Configuration

When configuring the user segment sharing feature, you must specify an ATG server to act as the user segment server. This server responds to requests for the list of ATG user segments. You should use the Content Administration server as the user segment server. Doing so ensures that the list of user segments that is passed to Experience Manager only contains segments that have been checked into Content Administration and deployed to the production server. It also ensures that user segment requests and responses occur in a secure environment. By default, a Content Administration server, as configured via CIM, is able to function as the user segment server.

Endeca Configuration

On the Endeca side, you must specify the hostname and port used to connect to the REST Service (the remainder of the URL after the port is well known and cannot be changed.) The REST Service allows Experience Manager to query the user segment server to retrieve the ATG user segments. You can configure the hostname and port in either of the following ways:

- By directly modifying the `atgServices.json` file for the Endeca application that needs access to ATG user segment data. With this approach, you are making a one-time change that will have to be repeated if you re-create the Endeca application in the future.
- By modifying the deployment template used to create your Endeca application to add the necessary hostname and port prompts and then store the responses in the application's configuration. With this approach, every Endeca application that is created using the modified deployment template will include the proper user segment sharing configuration.

Modifying the Endeca Application Directly

To specify the REST Service hostname and port directly in the Endeca application:

1. Navigate to the `config/editors_config` directory of your deployed Endeca application on disk, for example, `/usr/local/endeca/Apps/CRS/config/editors_config`.
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://<ATG host>:<ATG HTTP port>
   /rest/model/atg/userprofiling/ProfileGroupsActor/
   getAllProfileGroups?atg-rest-output=json&atg-preview=false
   }
   ```

   Note: Oracle recommends specifying the host and HTTP port for the Content Administration server.

4. Save and close the file.

5. Navigate to the `control` directory of your deployed Endeca 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 Endeca 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>Please enter the hostname of the ATG server to
   retrieve the user segments. This usually is the CA/Merch server.
   [Default:localhost]</prompt-question>
   <install-config-option>userSegmentsHost</install-config-option>
   <default-value>localhost</default-value>
   </token>
   <token name="USER_SEGMENTS_PORT">
   <prompt-question>Please enter the port number of the ATG server to
   retrieve the user segments. This usually is the CA/Merch server.
   Typical values are 7003 for WebLogic, 8080 for JBoss and Tomcat, and
   9080 for WebSphere.</prompt-question>
   <install-config-option>userSegmentsPort</install-config-option>
   </token>
   </custom-tokens>
   ```

3. In the same directory as the `deploy.xml` file, add an `/editors_config/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 ENDECA APP menu:

```
Enter the hostname of the ATG server to retrieve the user segments.  This usually
is the CA/Merch server. [[localhost]] >

Enter the port of the ATG server to retrieve the user segments.  This usually is
the CA/Merch server.  Typical values are 7003 for WebLogic,8080 for JBoss and
Tomcat, and 9080 for WebSphere. [[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 an ATG 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 an ATG 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.

13 Commerce Single Sign-On

The ATG Business Control Center and the Oracle Endeca Commerce Workbench are both used for managing content and its presentation on sites built with the integration of Oracle ATG Web Commerce and Oracle Endeca Commerce. To simplify working in both environments, ATG-Endeca 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 ATG Commerce SSO server instance that is responsible for managing the SSO sessions shared by the Business Control Center and the Workbench.
- An ATG plug-in that is responsible for communication between the Business Control Center and the SSO server.
- An Endeca plug-in that is responsible for communication between the Workbench and the SSO server.

The Endeca plug-in is discussed in the *Oracle Endeca Commerce Administrator*. This chapter discusses the SSO server and the ATG plug-in, and includes the following sections:

Commerce Single Sign-On Server

Commerce SSO is managed by a dedicated ATG server instance. When you set up your ATG environment in CIM, it gives you the option of setting up this server. The server includes the `SSO` and `DPS.InternalUsers` modules, and uses the same datasources as the Content Administration server, so it can access the ATG 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 the ATG Internal Profile Repository, and if it succeeds, the requested application is displayed.

To access the Business Control Center via Commerce SSO, the user must have an account in the Internal Profile Repository. To access the Workbench, the user must also have an account with the same user name in Endeca. If a user attempts to log into the Business Control Center with an ATG account that does not have a

corresponding Endeca account, the login to the Business Control Center succeeds, but the user is not able to access the Workbench. If a user attempts to log into the Workbench with an Endeca account that does not have a corresponding ATG account, the login fails.

The SSO module includes a web application that manages the single-sign on process. The application, whose context root is `sso`, provides five main functions that can be accessed via plug-ins by client applications: login, validate, keep alive, 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 ATG request-handling pipeline on the SSO server. This servlet dispatches requests to other servlets that provide the five 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 in to 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.
- `/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 Endeca 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.

ATG Plug-In

The ATG 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 fifth function, control, is accessed only by the Endeca plug-in.) These extensions include the `/atg/dynamo/servlet/dafpipeline/CommerceSSOServlet` component (of class `atg.userprofiling.commercesso.CommerceSSOServlet`), which is inserted in the ATG request-handling pipeline on the Content Administration server. This component manages much of the communication between the 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 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.

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 Content Administration server retrieves the corresponding user profile from the Internal Profile Repository and associates the current session with the profile. If authentication fails, the user is redirected back to 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.

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 `CommerceSSOServlet` component handles keep-alive calls to the SSO server. After a user's login is authenticated, `CommerceSSOServlet` 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 ATG session is terminated. If the SSO server cannot be reached, additional attempts are made to contact the server before ending the ATG session.

The polling behavior of the `CommerceSSOServlet` 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 ATG 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 ATG logout process is invoked, and the current ATG 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 ATG 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 ATG session after the SSO session has been terminated by the user logging out of the Workbench. The `CommerceSSOServlet` detects such logout requests and invokes the `logoutUser()` method of the `/atg/userprofiling/ProfileServices` component to handle logging out the user.

Maintaining User Accounts

To enable Commerce SSO to work for a specific user, the user must have accounts for both the Business Control Center and the Workbench. Both versions of the account must have the same user name.

A system administrator is responsible for creating the accounts and ensuring they are in sync. The system does not enforce this requirement itself.

Index

L

M

N

P

Q

R

S

T

U

V

W