COMMERCE

Assembler Application Developer's Guide

Version 11.3
April 2017
Document build ID: 2017-04-20T14:55:06-04:00

Assembler Application Developer's Guide

Product version: 11.3

Release date: 4-28-17

Table of Contents

List of Tables

List of Examples

Preface

Oracle Commerce Guided Search is the most effective way for your customers to dynamically explore your storefront and find relevant and desired items quickly. An industry-leading faceted search and Guided Navigation solution, Guided Search enables businesses to influence customers in each step of their search experience. At the core of Guided Search is the MDEX Engine™, a hybrid search-analytical database specifically designed for high-performance exploration and discovery. The Oracle Commerce Content Acquisition System provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. The Oracle Commerce Assembler dynamically assembles content from any resource and seamlessly combines it into results that can be rendered for display.

Oracle Commerce Experience Manager enables non-technical users to create, manage, and deliver targeted, relevant content to customers. With Experience Manager, you can combine unlimited variations of virtual product and customer data into personalized assortments of relevant products, promotions, and other content and display it to buyers in response to any search or facet refinement. Out-of-the-box templates and experience cartridges are provided for the most common use cases; technical teams can also use a software developer's kit to create custom cartridges.

About this guide

This guide provides an overview of Assembler application development. It covers the architecture of a typical Assembler application, as well as the tasks required to enable configuration through Experience Manager in Workbench.

The Tools and Frameworks package includes a Java implementation of the Assembler, so examples in this document are Java-based.

Who should use this guide

This guide is intended for developers who are building applications using the Assembler, and are supporting business users who configure these applications using Workbench. You should familiarize yourself with the concepts in the **Oracle Commerce Guided Search Concepts Guide** before reading this guide.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Oracle Support

Oracle Support provides registered users with answers to implementation questions, product and solution help, and important news and updates about Guided Search software.

You can contact Oracle Support through the My Oracle Support site at https://support.oracle.com.

1 About the Assembler

This provides an overview of the various components of the Assembler.

Introduction to the Assembler

This section provides a conceptual overview of the Oracle Commerce Guided Search Assembler.

What is the Assembler?

The Assembler is an Oracle Commerce component that performs the following essential roles in any Oracle Commerce application:

- It acts as the runtime component of Experience Manager, a tool that enables the business user to configure the runtime appearance and behavior of the application.
- It accesses values from a variety of sources, including databases, Digital Asset Management systems, social media feeds, and the MDEX engine.
- It creates view-ready application component models known as **cartridges**. A cartridge is a series of key and value pairs known as a content item. The key and value pairs contain values accessed by the Assembler. Your application renders these values visually, in the UI controls or other components that compose the pages in your application.

  Note

  Some content items contain other content items rather than consumer information. These content items represent different types of content and together form a hierarchical tree that can be traversed by the application when rendering a page.

Assembler Libraries

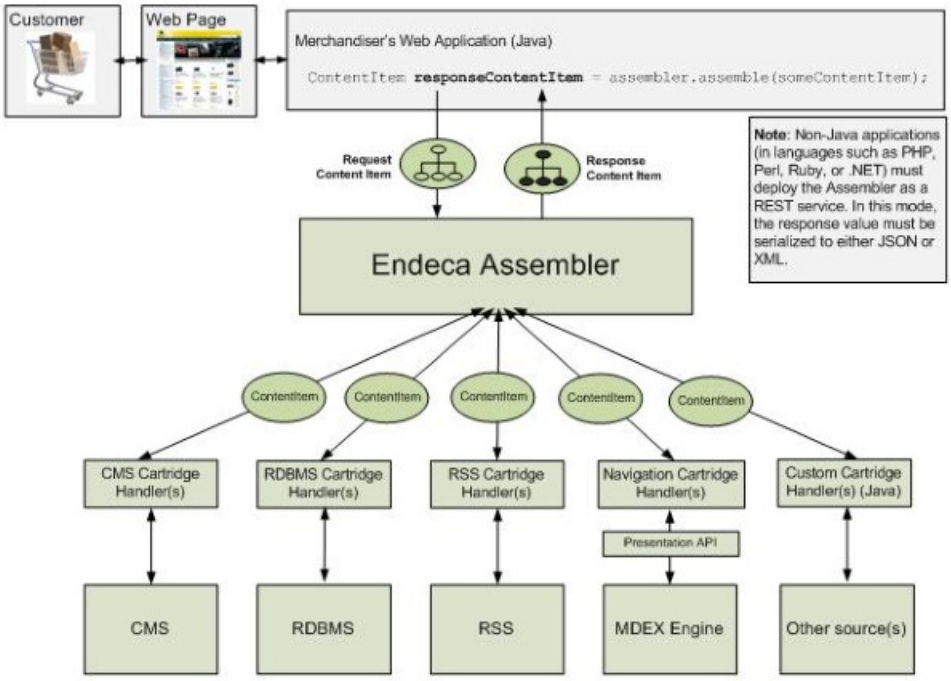
The Assembler classes are organized into two Java libraries:

- Assembler Core, packaged as `endeca_assembler_core-<version>.jar`. This library provides the core Assembler interfaces and a Spring implementation of the Assembler, along with the core facilities for building Experience Manager driven applications.
- Assembler Navigation, packaged as `endeca_assembler_navigation-<version>.jar`. This library provides the built-in cartridges and facilities for building applications with Search and Guided Navigation.

- A separate javadoc is provided for each JAR file.

The Role of the Assembler in an Oracle Commerce Application

The following diagram illustrates the role of the Assembler in an Oracle Commerce implementation:



As shown in the preceding diagram, the following things happen when customers request information through your application page:

1. Your application invokes the `assemble()` method as follows:

   ```
   ContentItem contentItem = new RedirectAwareContentInclude("/myUrl");

   ContentItem responseContentItem = assembler.assemble(contentItem)
   ```

   where `/myUrl` is the URL to a page that you are assembling in Experience Manager and `responseContentItem` is a tree of other content items.

2. The `assemble()` method sends `responseContentItem` to the Assembler.

3. The Assembler passes the individual content items in `responseContentItem` to cartridge handlers, each of which handles a different content type. Each content item specifies a request for information.

4. The cartridge handlers pass the requests on to the appropriate sources of information, such as an MDEX Engine, a relational database system, a content management system, and so on.

5. The cartridge handlers receive and process information from their respective sources. The handlers contain all the logic needed to process the information, though they may also process requests without requiring input from an external data source.

6. Each cartridge handler returns to the Assembler a content item that contains the requested information.

7. The Assembler combines the content items that it receives from all of the cartridge handlers into a `responseContentItem`, which is structured as a tree that contains all of the information required by the front end application.

8. The Assembler returns `responseContentItem` to the front end application.

9. Rendering code in the application converts the information in `responseContentItem` into a form that can be displayed in the appropriate cartridges on your application page. Typically, a cartridge renderer (a separate module of rendering code) processes and displays the information for each content item in the `responseContentItem` tree.

   Note

   The Assembler can return XML or JSON representations of objects for consumption by a variety of rendering engines, such as .NET, PHP, or Flash-based applications. It can also return model objects as POJOs (plain old Java objects) when embedded in a native Java application.

The Tools and Frameworks package includes a Java Assembler implementation that uses Spring to resolve cartridge handlers and services.

You can develop extensions to the framework to interact with your resources, centralizing runtime data retrieval and manipulation in your application. For these reasons, the Assembler can be integrated with organizations that use Service-Oriented Architecture.

Basic Assembler concepts

The Assembler stores and manipulates data as sets of `key` : `value` pairs known as content items. Content items can represent cartridges, which map to front-end features in an application.

About Content Items

Some content items are structural components such as application pages that contain additional content items. Other content items map to front-end components in an application, such as image banners.

For example, in the Discover Electronics reference implementation, the entire default "browse" page is represented by a content item that contains the page template. Each section of the page is also a content item, nested within the containing "three column page" content item. Within those sections are additional content items that represent front-end features:

- ThreeColumnPage
  - headerContent
    - Search Box
  - leftContent
    - Breadcrumbs
    - Guided Navigation
  - mainContent
    - SearchAdjustments
    - ContentSlotMain

        - ResultsList
    - rightContent
        - RecordSpotlight

Because the content items are organized as a tree, they are as a group easy to traverse for rendering.

About Cartridges and Cartridge Templates

A cartridge is a content item with a specific role in your application; for example, a cartridge can map to a GUI component in the front-end application. The Assembler includes a number of cartridges that map to typical GUI components – for example, a Breadcrumbs cartridge, a Search Box cartridge, and a Results List cartridge. You can create other cartridges that map to other GUI components expected by your business users.

Every cartridge is defined by a template. A cartridge template defines:

- The structure and initial configuration for a content item.
- A set of configurable properties and the associated editors with which the business user can configure them.

Experience Manager instantiates each content item from its cartridge template. This includes any configuration made by the business user, and results in a content item with instance configuration that is passed to the Assembler.

About Cartridge Handlers

A cartridge handler takes a content item as input, processes it, and returns a content item as output.

The input content item typically includes instance configuration, which consists of any properties specified by a business user using Experience Manager in Workbench. The content item is typically initialized by layering configuration from other sources: your application may include default values, or URL parameters that represent end user selections in the front-end application.

A cartridge handler can optionally perform further processing, such as querying a search engine for data. When processing is finished, the handler returns a completed content item to the application.

> **Note**
>
> Not all cartridges require cartridge handlers. In the case of a content item with no associated cartridge handler, the Assembler returns the unmodified content item.

For detailed information regarding the `CartridgeHandler` interface, see About the CartridgeHandler interface (page 163), or refer to the *Assembler API Reference (Javadoc)*.

Example: The Results List Cartridge

Consider the Results List cartridge included with Tools and Frameworks in the Assembler Navigation JAR file. The `ResultsList` object that backs the cartridge is a content item. The ResultsList cartridge template exposes a subset of object properties for configuration in Experience Manager. The remaining properties are configurable through the UI in the front-end application.

The Results List cartridge handler combines the default, instance, and URL configuration values to create a query to send to the MDEX Engine. The values in the query response are used to populate the `ResultsList` content item and return it to the application for rendering.

Configuring Assembler applications in Experience Manager

The Assembler interacts with the Experience Manager tool in Workbench to expose content configuration to business users.

Experience Manager instantiates each content item from its content XML. In an unconfigured cartridge, this XML is identical to the cartridge template (including any default values specified in the template). When a business user opens and modifies a cartridge in Experience Manager, their settings are saved in the content XML. In an authoring environment, this XML is stored in the Workbench. In a production environment, it is read from the path configured using configurationPath property of FileStoreFactory.

At runtime, the Assembler deserializes the content XML to create the appropriate content item object, and passes it to its corresponding cartridge handler for processing.

In addition to creating instances of front-end application components in this manner, the business user can also use structural content item templates (such as the Three-Column Navigation Page template) to create the organizational elements of a site. For example, your business user can create an "About Us" page, a "Frequently Asked Questions" page, and other static elements of a site by selecting and populating suitable cartridge templates.

Assembler Search and Guided Navigation Features

The Assembler Navigation package provides a set of Search and Guided Navigation cartridges for use with the MDEX Engine. These cartridges are included in the `endeca_assembler_navigation-<version>.jar` file.

The reference application includes templates that use these navigation cartridges to enable configuration in Experience Manager and render the resulting data in a front end application.

A navigation cartridge exposes MDEX engine features to an Assembler application. It also enables a business user to configure powerful Guided Navigation features using UI components that can be customized by an application developer to fit business needs.

The navigation cartridges include the following:

- Search Box
- Auto-Suggest Search Results
- Dimension Search Results
- Search Adjustments
- Refinement Menu
- Breadcrumbs
- Dimension Navigation
- Results List
- Record Details
- Media Banner
- Record Spotlight and Horizontal Record Spotlight

Example: The Results List cartridge

The Results List cartridge displays MDEX Engine search results for an end user query. It is backed by a `com.endeca.infront.cartridge.ResultsList` content item object, which extends the `com.endeca.infront.assembler.BasicContentItem` interface.

The input to the Assembler consists of a configuration model -- a content item with MDEX Engine query information such as the end user's search terms, selected search refinements, sorting options, and records per page. These are passed in as a `com.endeca.infront.cartridge.ResultsListConfig` object.

The `ResultsListHandler` generates a query from the properties on `ResultsListConfig`, then sends the query to the MDEX Engine. It instantiates a `ResultsList` content item using the query response, and copies over some of the properties from the configuration model (such as records per page and sorting) directly. This view-friendly `ResultsList` object is then returned to the application for rendering.

Cartridge configuration comes from the following sources:

- Default configuration — For Spring-based Assembler implementations, this is specified in the Spring context file.
- Instance configuration — Specified by the business user in the Results List cartridge in Experience Manager.
- Request-based configuration — Specified by the application end user; this includes any search terms or selected dimension refinements, among other things.

Default Cartridge Configuration

This section illustrates the default cartridge configuration of a Spring-based Assembler, using the Discover Electronics reference application as the example.

The default cartridge configuration is specified in the Spring context file, located at `ToolsAndFrameworks\<version>\reference\discover-electronics-authoring\WEB-INF\assembler-context.xml` for the authoring instance of the Discover Electronics reference application. This includes values for the following properties on the `ResultsListConfig` content item:

- `sortOption` — The sorting options available to the end user when viewing the list of query results.
- `relRankStrategy` — The Relevance Ranking strategy applied to search results. For more information about Relevance Ranking, see the *MDEX Engine Developer's Guide*.
- `recordsPerPage` — The number of records to display per page of results.

Note

The above list is a subset of configured properties and provided as an example.

Instance Configuration

This section illustrates a cartridge instance configuration for a Spring-based Assembler, using the Discover Electronics reference application as the example.

The cartridge instance configuration comes from the values in the `ToolsAndFrameworks\<version>\reference\discover-data\import\templates\ResultsList\_.json` cartridge template. The template exposes the following properties to the business user in Experience Manager:

- `boostStrata` — A list of records to elevate to the top of the Results List.

- `buryStrata` — A list of records to move to the bottom of the Results List.
- `sortOption` — The business user can override the default sorting options.
- `relRankStrategy` — The business user can override the default Relevance Ranking strategy.
- `recordsPerPage` — The business user can override the default number of records to display on each page.

Request-Based Configuration

The application end user's configuration in Discover Electronics is passed to the `ResultsListConfig` object as URL parameters, though you may choose to implement such functionality differently in your own application.

- `offset` — Controls the record offset of the displayed results in order to control record display while paging through results.
- `relRankTerms` —The end user's search terms.
- `sortOption` — The end user can override the default values and the instance configuration.
- `recordsPerPage` — The end user can override the default values and the instance configuration.

The Results List cartridge handler combines the default, instance, and request-based values to create a query to send to the MDEX Engine. The values are used to populated the `ResultsList` content item and return it to the application for rendering.

Assembler Architectural Overview

This section provides an architectural overview of the Assembler.

Related links

- The Assembler processing model (page 7)
- About serialization and de-serialization (page 11)
- The Assembler eventing framework (page 12)
- About Assembler error handling (page 13)

The Assembler processing model

The core of the Assembler is the `assemble()` method, which takes a content item representing a cartridge instance configuration and invokes cartridge handlers to process it into a response content item.

The Assembler uses the visitor pattern to traverse the input content item and any child content items, and invokes the appropriate cartridge handler (if any) for each of them.

The Assembler makes two passes over the input content item:

1. In the first pass, the Assembler calls `CartridgeHandler.initialize()` followed by `CartridgeHandler.preprocess()` on each content item in the tree. This is a pre-order traversal of the tree

(working from the top of the tree down through its children), so cartridge handlers may add or modify child content items at this stage.

2. In the second pass, the Assembler calls `CartridgeHandler.process()` on each content item, which returns the response content item for that cartridge. This is a post-order traversal of the tree (working from the bottom up), so child content items are processed before the parent. The response object for the root content item of the tree (the content item originally passed in as input to the first `assemble()` call) contains the response objects for all its child cartridges.

The default implementation of the Assembler uses Spring to map each cartridge to the appropriate handler based on its content type. This content type corresponds to the template identifier that was used to create the content item object. If no cartridge handler is defined for a particular content type, the instance configuration is passed through as the response model.

Example

For example, consider the following content item:

```
NestingDollContentItemSubclass nestingDoll
```

This content item represents a Russian Nesting Doll. It includes properties for its name, color, and its child content item:

```
nestingDoll.name = "Nesting Doll"
nestingDoll.color = red
nestingDoll.child = secondNestingDoll
```

The `secondNestingDoll` contained within is green. It contains a `thirdNestingDoll`, which is blue. Assuming there is no cartridge handler for `NestingDollContentItemSubclass`, an `assemble(nestingDoll)` call executes the following:

1. The pre-order traversal starts. There is no cartridge handler for `NestingDollContentItemSubclass`, so no `initialize()` or `preprocess()` calls are made for `nestingDoll`.

2. Similarly, no calls are made for `secondNestingDoll` or `thirdNestingDoll`. At this point, the pre-order traversal is complete.

3. The post-order traversal starts. The `thirdNestingDoll` object is returned as-is, since there is no handler to invoke a `process()` method.

4. Similarly, the `secondNestingDoll` and `nestingDoll` objects are returned, unmodified. Serialized to JSON, the response looks like the following:

```
@type": "NestingDollTemplateType",
"name": "Nesting Doll",
"color": "red",
"child": [
    {
      @type": "NestingDollTemplateType",
      "name": "Second Nesting Doll",
      "color": "green"
      "child": [
        {
          @type": "NestingDollTemplateType",
          "name": "Third Nesting Doll",
```

```
            "color": "blue"
            "child": []
          }
        ],
    },
]
```

What if you create a cartridge handler for `NestingDollContentItemSubclass` that doesn't override the `initialize()` or `preprocess()` methods, but implements logic to add a property `colorType` of value `warm` or `cool`, based on the `color` property? Steps 1-2 above don't change, but Step 3 invokes the new logic, and the property shows up in the response:

```
@type": "NestingDollTemplateType",
"name": "Nesting Doll",
"color": "red",
"colorType": "warm",
"child": [
    {
      @type": "NestingDollTemplateType",
      "name": "Second Nesting Doll",
      "color": "green",
      "colorType": "cool",
      "child": [
        {
          @type": "NestingDollTemplateType",
          "name": "Third Nesting Doll",
          "color": "blue",
          "colorType": "cool",
        }
      ],
   },
]
```

About content items

A content item is a set of `key : value` pairs where the key is a property name and the value may be any primitive type, or another content item. The `com.endeca.infront.assembler ContentItem` interface extends `java.util.Map`.

Content items in the Assembler represent either structural components of an application page, or GUI components on the page itself. A call to the `Assembler.assemble(ContentItem)` method accepts as input a `ContentItem` containing configuration, and returns a content item as output. The response content item can encompass an entire page in an application, with each sub-section of the page (such as the search box or the search results list) represented as its own nested content item.

Note

In the default implementation of the Assembler, the `ContentItem` interface is implemented by the `com.endeca.infront.assembler.BasicContentItem` class. The navigation cartridges in the package extend this implementation for their individual use cases.

About ContentInclude and ContentSlotConfig objects

The default Assembler implementation typically takes a `ContentInclude` or `ContentSlotConfig` object as input to the Assembler. The first specifies a content item by URI, while the second retrieves a content item from a specified folder according to template type and ID restrictions, trigger criteria, and content item priority.

Both methods retrieve the associated configuration for the content item in Workbench.

Defining a ContentInclude object

A `ContentInclude` object defines a single content item to pass into the Assembler (though keep in mind that a content item may contain additional content items as children). It resolves a URI to a content item within a configured content source (typically the Endeca Configuration Repository).

Defining a ContentSlotConfig object

Unlike a `ContentInclude` object, which explicitly specifies a content item to pass as input to the Assembler, the `ContentSlotConfig` object defines a set of criteria for dynamically retrieving one or more content items at runtime. In most cases the content administrator creates and populates `ContentSlotConfig` objects through editors in Experience Manager, although you can still programatically instantiate them if necessary.

The dynamic content slot is populated based on the following restrictions:

- **Content paths** — The path or paths to content folders in Experience Manager. Any content items within the specified folders or within sub-folders are considered valid for retrieval.
- **Template types (Optional)** — The types of content item to retrieve, based on the `type` attribute of the cartridge template used to create it. For example, a Record Spotlight slot in the Discover Electronics reference application is restricted to content items created from a template with `type="SecondaryContent"`.
- **Template IDs (Optional)** — The template IDs to match against. This is a narrower restriction than restricting by template `type`, and instead restricts based on a unique template `id`. For example, a Record Spotlight slot in the Discover Electronics reference application is restricted to content items created from a template with `id="RecordSpotlight"`.
- **Rule Limit** — The number of matching content items to retrieve. This is applied after the above restrictions, and after checking for triggered content items.

When the list of possible content items has been narrowed down, the `ContentSlotHandler` issues a content trigger request. This checks valid content items against any triggers defined in Experience Manager. Trigger criteria can include:

- The user's search terms or refinement selections, also referred to as their "navigation state."
- Characteristics of the user, such as past buying habits or geographical location. This information constitutes the user's "user segment."
- The current date and time, referred to as "schedule triggers."

The list of results is limited to triggered content items and ordered by the priority assigned to each content item in Experience Manager. The number of results is truncated to the value specified for the content slot (also specified on `ContentSlotConfig`). The Assembler then processes the content items and returns them in its response.

About nesting content items

Content items may contain other content items, which can include both `ContentInclude` references and `ContentSlotConfig` definitions

For example, in Discover Electronics the `/browse` path corresponds to a page within the sitemap. The browse page consists of a content slot that references the Web folder. Most of the pages within the Web Browse Pages folder contain a mixture of static and dynamic content items. As the Assembler processes the query for `http://`

`www.example.com/discover/browse` (assuming no search terms or refinement selections), the following steps occur:

1. The Assembler is invoked with a `ContentInclude` item with the URI `/pages/browse`.
2. The Assembler invokes the `ContentIncludeHandler` to retrieve the configuration for the browse page, which is a `ContentSlotConfig` that specifies a single content item from the Three-Column Page collection.
3. The Assembler invokes the `ContentSlotHandler` to retrieve the highest priority content item within the Three-Column Page collection. In this case, it is the Default Browse Page, which is a `ThreeColumnPage`.
4. There is no cartridge handler configured to process the `ThreeColumnPage`, but it contains child content items, so the Assembler goes on to process the child content items:
   a. It passes the configuration for the search box cartridge through to the response object.
   b. It invokes the `BreadcrumbsHandler` to process the breadcrumbs cartridge.
   c. It invokes the `ContentSlotHandler` to process the navigation slot, which in turn retrieves the Default Guided Navigation configuration from the Guided Navigation collection and invokes `DimensionNavigationHandler` to process it.
   d. It invokes the `SearchAdjustmentsHandler` to process the search adjustments cartridge.
   e. It invokes the `ContentSlotHandler` to process the results list slot, which in turn retrieves the Default Results List configuration from the Results List collection and invokes `ResultsListHandler` to process it.
   f. It invokes the `RecordSpotlightHandler` to process the spotlight records.

About serialization and de-serialization

The Assembler serializes content items, including any Workbench content, as XML in the Workbench (or on a file system in a production environment). This XML is deserialized during an `assemble()` call when retrieving a content item to pass it to its cartridge handler.

You can also use the included classes to serialize the Assembler response to a format that is more convenient for use in your front end application. For example:

```
// Invoke the Assembler on myContentItem
ContentItem responseContentItem = assembler.assemble(myContentItem);
// Serialize the Assembler response to JSON
response.setCharacterEncoding("UTF-8");
JsonSerializer serializer = new JsonSerializer(response.getWriter());
serializer.write(responseContentItem);
```

When Assembler is deployed as a service, the Assembler service web application needs to specify a serializer that will be used for the response.

Note

The Assembler includes default implementations of a `JSONResponseWriter` and an `XMLResponseWriter`. You can provide your own implementation if you need to output the Assembler response to a different format (such as a different XML representation).

For detailed information, refer to the documentation for the `com.endeca.infront.serialization` package in the Assembler Core API Reference (Javadoc).

The Assembler eventing framework

The Assembler includes an eventing framework that fires events at different points in an `assemble()` call. Creating listeners for these events enables cartridge handlers to retrieve or modify data at more granular points in the Assembly process.

Note

The logic included in an event listener is evaluated for every cartridge handler, and that event listeners do not have access to the current Assembler request or to the navigation state.

Related links

- Creating an event listener (page 167)

Assembler event framework reference

The Assembler includes an `AssemblerEventListener` interface that you can use to create and register event listeners.

The Assembler fires the following events:

Event	Condition
`assemblyStarting`	Fires when an `assemble()` call starts.
`assemblyComplete`	Fires when an `assemble()` call completes.
`assemblyError`	Fires when an `assemble()`call is aborted due to an unrecoverable error.
`cartridgeInitializeStarting`	Fires when a cartridge handler calls the `initialize()` method.
`cartridgeInitializeComplete`	Fires when a call to the `initialize()` method completes.
`cartridgePreprocessStarting`	Fires when a cartridge handler calls the `preprocess()` method.
`cartridgePreprocessComplete`	Fires when a call to the `preprocess()` method completes.
`cartridgeProcessStarting`	Fires when a cartridge handler calls the `process()` method.
`cartridgeProcessComplete`	Fires when a call to the `process()` method completes.
`cartridgeError`	Fires when a cartridge fails due to a local error. This stops execution of the cartridge handler workflow, and prevents any additional events from firing.

Event payload

Each Assembler event has an `AssemblerEvent` payload consisting of three objects:

- `Assembler` — the Assembler object responsible for servicing the request.
- `ContentItem` — the content item currently undergoing processing within the `assemble()` call.
- `CartridgeHandler` — the cartridge handler associated with the event.

About Assembler error handling

In case of an error during processing, the Assembler API defines two kinds of exceptions: `AssemblerException` and `CartridgeHandlerException`.

The exceptions are distinguished as follows:

Exception	Description
`AssemblerException`	Indicates that an exception occurred while creating or processing an Assembler request. Exceptions of this type indicate that the entire assembly process was terminated.
`CartridgeHandlerException`	Indicates that an exception occurred while invoking a single cartridge handler. Exceptions of this type do not terminate the entire assembly process.

Both types of exceptions are returned as part of the Assembler response.

Error handling in the Assembler service

The Assembler service returns an HTTP status code of 200 (OK) regardless of whether any exceptions occurred during Assembler processing. Error conditions are serialized as exceptions in the Assembler response, as with the following example:

```
{
    @error: "com.endeca.infront.assembler.CartridgeHandlerException"
    description: "Detailed cartridge handler error description"
}
```

Unchecked exceptions result in the Assembler service returning HTTP status code 500 (Internal Server Error).

About cartridges and content items

This section describes how cartridges expose content in an application.

About cartridges

The component model consists of configurable *content items*. Cartridges expose these content items in a rendered format for the front-end application.

A content item is a map of properties or key-value pairs, where the key is a string representing the property name and the value may be any primitive type (including String, Boolean, List, and Map) or another content item. This allows for content items to be nested within other content items, forming a content tree that represents the structure of a Web page and all its components.

There are generally two kinds of content items within an application:

- *Container content items* are primarily structural components. They define the logical (and sometimes physical) structure of the content to be rendered by an application. The top-level container typically represents a Web page with sections that can contain other content items (leaf content items or, occasionally, other containers). In a Web application, these sections may correspond to areas on the page with certain assumptions about layout and rendering. In other applications, they may represent logical groupings of related components.
- *Leaf content items* are typically functional components. They contain information about content to be displayed in the application, and typically encapsulate the configuration for a particular feature, such as a Guided Navigation component, spotlight, or results list. Leaf content items are also referred to as *cartridges*.

A page may contain cartridges directly (in which case the configuration for the cartridges is triggered along with the page) or the page can contain a dynamic slot, which serves as a placeholder for cartridges that can be triggered independently of the page in which they display.

Structure of cartridges

A cartridge is a functional component that a content administrator can choose to display on a page.

The basic aspects of a cartridge are the following:

- The **cartridge instance configuration**, which consists of a content item created by a business user in Experience Manager
- The **cartridge handler**, which is the Assembler component that contains the processing logic for the associated feature
- The **response model**, which is the content item returned from the Assembler to the application for rendering

The configuration model for a cartridge is defined by a *cartridge template*, which describes the properties that can be configured as well as the interface through which the content administrator can specify their values in Experience Manager. Cartridges typically have configuration options specific to the cartridge's function, such as the number of refinements to display (and the order in which to display them) for a Dimension Navigation cartridge; the records to promote for a Spotlight cartridge; or the sort options and records per page for a Results List cartridge.

1. At query time, the configured values of the cartridge properties become an input to the Assembler.

   The Discover Electronics reference application contains several sample templates or cartridges that demonstrate core functionality. You can customize them for your own application or write your own templates in order to add or remove configuration options or to pass additional information to the Assembler or the front-end application.

2. At query time, the Assembler invokes the appropriate cartridge handler to process the cartridge configuration.

   The core cartridge handlers also have access to information about the initial request context that triggered the cartridge. The cartridge handler is responsible for generating a response model based on this configuration. In most cases this involves fetching content from an ext ernal resource.

   In the case where the configuration model is the same as the response model, no cartridge handler is needed; the default behavior of the Assembler is to pass the configuration properties through to the response model.

3. The Assembler passes the response model to the corresponding renderer in the application.

As a best practice, the application should contain several modular *renderers*, each intended to handle the output model for a particular cartridge or cartridge type. The Discover Electronics application includes reference JSP pages to render each cartridge. These renderers are intended to be updated for styling or otherwise customized for your application.

2 Designing an Assembler Application

This part discusses the steps for designing your Assembler application and the steps for creating templates.

Related links

- Planning an Assembler Application (page 17)

Planning an Assembler Application

This section covers considerations for designing your Assembler application.

About planning your application sitemap

An Assembler application consists of a combination of static pages and dynamic pages that contain content related to an end user's navigation state. Your planned sitemap helps determine what pages and content folders you should create for your application.

Consider a site with the following structure:

- about
  - contact
  - faq
- promotions
  - christmas
  - mothersDay
- browse
- details

In this case, each of the pages maps directly to a set of content. To separate most of the content out from the site structure and support dynamic triggering, the organization of an Assembler application is divided into the pages within an application, and the content that populates them:

- pages

    - about
        - contact
        - faq
    - browse
    - details
- content
    - guided navigation
    - record details
    - browse pages
        - default
        - christmas
        - mothers day
    - spotlights
        - top rated
        - best sellers

In the example above, the promotional Christmas and Mother's Day pages no longer exist as explicit pages. Instead, the content associated with those promotions is stored as available "browse" page configurations that each trigger during a specified date range.

You can refer to the Discover Electronics reference application for a further example of how content and pages can be separated. When planning your own application, you should consider which locations in your site are best represented as pages, and which locations consist of triggered content on an existing page.

About page types

A typical application has several types of pages that are displayed under different circumstances or contain different content.

For example, a site may have the following three basic page types:

These pages may differ in the following ways:

- **They are intended to be displayed in different contexts.** The home page appears before the user has made any selections. The results page appears only when the user has performed some search or navigation query. The record detail page appears only when a user has selected a specific product. These conditions are configured in Experience Manager as triggering criteria.
- **They display different types of content.** A home page or category page typically displays high-level promotions and merchandising. A results page displays a list of record results as well as additional controls for the user to select additional facets or otherwise refine the search. A record detail page displays detailed

product information as well as controls for transactions (such as add to cart, wishlist, and so on). These differences in content imply differences in layout, which is configured at the template level.

- **They are accessed through different URLs.** The home page is accessed at the base URL for the site. Search results pages may be accessed at a URL that includes the path `/browse/`. Record details pages may be accessed at a URL that includes the path `/detail/`. These URL mappings are typically achieved by setting up individual services for each page type.

The Discover Electronics reference application includes servlets for results pages and record details pages.

About page structure and content types

An Experience Manager template defines the logical structure of a page and the types of content that the page can contain.

Every template defines a content item that can be processed by the Assembler. A content item describes the logic of how to promote content for display to application users. Content items have several parts: the records in a data set, the conditions that must be met for those records to display, and the templates that determine how those records are rendered in the application.

A page template defines a container content item with sections that can be populated with other content items, such as the following:

Typically, a section represents a physical area on the page, but it can also represent a functional grouping, including content that may not be visible to an end user. Each section has an associated *content type* that determines what kind of content items can be inserted in that section. An application may have multiple cartridges of each type, providing greater flexibility for the content administrator to configure the content for a specific page.

You can create templates for different page types within your application and define which content types are valid for each type of page. You can create templates for high-level page structures and different layouts for a single page type. Each of the content items that can be inserted into a template is itself defined by a template, and may be either another container content item or (more commonly) a leaf content item associated with a front end feature.

About mapping pages to services

You can map the URL paths of pages in your application to specific services.

Services can be used to set attributes on the incoming request before it is processed by the Assembler depending on the type of page being requested, which can control what content is triggered in response to the request, and the format in which the response is returned.

When a content administrator defines a new application page in the reference application, requests on that page are mapped to the `/services` servlet. Your application should include logic for mapping arbitrary pages to a controller, though you may also choose to explicitly define additional services for certain pages within your site. Additionally, your UI tier must be able to resolve whatever links you expect your content administrators to create. For more information about handling application URLs, see "Working with Application URLs."

Creating a page

The Content Tree in the left pane of Experience Manager is divided into two sections: Site Pages and Content. You create pages within the Site Pages section.

You must deploy and provision your application with the EAC in order to modify it in Workbench.

To create a page, follow these steps:

1. Log in to Workbench and navigate to Experience Manager.
2. Mouse over the Site Pages heading in the Content Tree.

   The drop-down menu arrow appears on the right.
3. Click the drop-down menu arrow and select Add Page.

   The Add Page panel appears.
4. Enter a Name/Path for the new page.

   This is the part of the URL path that uniquely identifies the page within your application.
5. Click Create.

   The new page is added to your application.

A page exists as a content item in Experience Manager. A content administrator can configure it directly by selecting a template with included editors, or they can specify a template with a dynamic slot to populate the page from one or more selected content folders.

About content folders

Before a content administrator can configure dynamic content items within an application, you must create content folders to contain those items. Content items within the same folder are evaluated against each other at runtime to determine which item (or items) should be returned to populate a defined section of the current page.

In Experience Manager, content folders define the top-level organizational structure of an application, in which the content administrator can browse for content. If a query satisfies the trigger criteria for multiple content items within a folder, items with higher priority take precedence over those with lower priority. A single application request may trigger content items from multiple folders

Content folders have the following properties:

- Template type — Specifies the type of content items that can be created in this collection, as defined by the `type` attribute of the content template.
- Template ID — Specifies the type of content items that can be created in this collection, as defined by the `ID` attribute of the content template. This is more restrictive than specifying by template type, as an ID is unique to a single template.

Oracle recommends that you create at least one content folder for pages and one for each slot on the page that can contain either shared or variable content. This provides a logical organization of content within Experience Manager. It enables content to be triggered independently of the pages that contain them and also enables content in one slot to be triggered independently of content in another slot.

For example, the Discover Electronics reference application includes the following content folders :

- **Mobile \ Mobile Browse Pages** — Top-level page configuration for pages viewed from a mobile device. Mobile pages must be more streamlined than Web pages, so they require a different page template.

- **Shared \ Auto-Suggest Panels** — Configuration for the auto-suggest panel that is displayed when a user starts to enter a search query. The Shared collections return the same response model for both the Mobile and Web versions of the application, but the renderers vary based on the client.
- **Shared \ Detail Pages** — Configuration for record details pages within the application.
- **Shared \ Guided Navigation** — Configuration for the Guided Navigation menu.
- **Shared \ Results List** — Configuration for a list of search results.
- **Web \ Spotlights** — Category-specific product spotlights that are displayed above the search results when a user navigates to those products.
- **Web \ General \ Pages** — Top-level page configuration for Web pages. These templates are structural and primarily consist of dynamic slots that pull in content items from other collections to populate the page.

Content folders example

Content folders determine which content items are evaluated and returned when populating a dynamic section of an application page.

Suppose you have a site where a typical structure for a search and navigation page looks like the following:

Based on this template, the content administrator wants to configure a page for a specific trigger (for example, Category > Cameras > Digital Cameras) using contextual, shared, and variable content as in this picture:

- The header and footer are populated as dynamic slots with default triggering criteria, in order to avoid defining them multiple times for a large number of pages.
- The Guided Navigation and Results List cartridges are configured specifically for this page and do not need to vary based on criteria other than the page triggers.
- The Banner area is configured to display a different image depending on the brand that the site visitor has selected.
- The Spotlight area displays a mix of promotions based on triggers that are independent of the triggering criteria for the page itself. For example, a "Holiday Specials" spotlight may display for the date range between November 1 and January 2.

The configuration for the page (as specified in Experience Manager) looks like this:

The configuration for Guided Navigation (including which dimensions to display and which dimension values to boost or bury within those dimensions) and for the Results List (including default sort options and record boost and bury) are specified as part of the page configuration. The other slots on the page contain only placeholders. The actual Header, Footer, Banner, and Spotlight content items that display when someone visits the site are defined in their respective content folders.

The mechanism for populating these slots is the same regardless of whether the content that should display in each slot is shared or variable content. The only difference between the two kinds of content is in the trigger criteria on the content items within those collections: variable content, such as the Spotlight, has triggers that

are more specific than the page trigger. Reusable content, such as the generic header and footer, has triggers that are more general than or orthogonal to the page trigger.

When the content administrator has created all the content needed to populate this page (and a few other pages), the application may include the following content items in the following folders:

The content folders are configured as follows:

- The Browse Page folder contains all the content items representing search and navigation pages in the site.
- The Brand Banner folder contains cartridges of type `MediumBanner` that are appropriate to display in the Banner slot. This dynamic slot on the Browse page has an evaluation limit of `1`, since the page is designed to display only one banner at a time.
- The Spotlight folder contains cartridges of type `SidebarItem` because items created in this collection are intended to display in the Spotlight slot in the right column. Because this space is intended to display several independently triggered spotlight items, the evaluation limit for the dynamic slot on the Browse page is `3`.
- The Header and Footer folders each contain cartridges of type `FullWidthContent`.

Each page or content item within these folders has an associated trigger and priority (relative to the other items in the same folder) specified by the content administrator in Experience Manager.

When a site visitor refines on Category > Cameras > Digital Cameras and Brand > Sony, the following content triggers:

- The Digital Cameras page is returned as a Page, which includes the content administrator's configuration for Guided Navigation and for Results List. Note that the Default page (with a trigger of "Applies at all locations") is also eligible to fire, but the Digital Cameras page has a higher priority, therefore it takes precedence and the Default page does not fire.
- The Banner slot is populated by the highest priority content item in the Brand Banner folder that matches the user's navigation state. In this case, it is the Sony cameras banner. Again, there is a Default banner but it does not fire because it has a lower priority.
- The Spotlight slot is populated by the highest priority content items in the Spotlight folder that match the user's navigation state. In this case, the Default spotlight does fire because there is room for three spotlights in this slot and that item has a high enough priority (among those that satisfy the user's context) to be included. These three content items display in the Spotlight area in order of priority.
- The Header and Footer folders have only one content item each, which is set to display at all locations, therefore the same content is returned for this page as for all pages.

In this example, content is returned from five content folders. Priority between items is specified within each folder. It does not make sense to prioritize the Sony cameras banner against the April spotlight cartridge, for example, because they are not competing against each other to be displayed on the page. In general, content items with more specific trigger criteria should have a higher priority than those with more general criteria, especially if they are used in a dynamic slot with an evaluation limit of `1`.

Oracle recommends that you create separate content folders for each area on the page, even if they have the same content type. For example, if you want to have two banners on the page, each populated via dynamic slots, they should reference two different folders, or else the same banner (the one with the highest priority for the current navigation state) is returned for both sections of the page.

Oracle also recommends that you do not mix reusable and variable content within the same folder. For example, if a slot (such as the Spotlight slot) can be populated with either reusable or variable content, create two different folders, Reusable Spotlights and Variable Spotlights. The content administrator can configure a particular page to populate the Spotlight slot from either folder as applicable. In order to populate the same slot with a mixture of reusable and variable content, the content administrator can insert two (or more) placeholders in the Spotlight slot, each referencing the corresponding folder for each type of content.

The final result for the site visitor who is looking at Sony cameras looks something like the following:

Creating a content folder

The Content Tree in the left pane of Experience Manager is divided into two sections: Site Pages and Content. You create content folders within the Content section.

You must deploy and provision your application with the EAC in order to modify it in Workbench.

To create a content folder:

1. Log in to Workbench and navigate to Experience Manager.
2. Mouse over the Content heading in the Content Tree.

   The drop-down menu arrow appears on the right.
3. Click the drop-down menu arrow and select Add Folder.

   The Add Content Folder panel appears.
4. Enter the Name of the folder you wish to add.
5. Optionally, select a content type restriction.

   The drop-down list is populated based on the available `type` values for the set of templates uploaded to the application.

   This selection restricts the content items within the folder to the specified type.
6. Click Add.

   The new content folder is added to the Content Tree in Experience Manager.

About moving content folders

You can move and re-organize content folders in the Content Tree within Experience Manager.

If you move a content folder that includes dynamic content referenced elsewhere in the application, a warning dialog appears with a list of content items that rely on the content you are moving. You must manually update these content items if you proceed with the move.

About sites

Oracle Commerce Experience Manager enables you to build an application that can run multiple web sites using a single index. Business users can use this application to create site-specific pages that use a single index. Even if you are not building an application that supports multiple sites, your application must contain at least one site.

You can create applications to support multiple web sites that share the same code base, templates, cartridges, and search configuration. Each web site can have its own unique set of pages that display the site's unique look or branding.

Note

If you are using Oracle Commerce Guided Search, applications have one site by default, and Oracle does not support adding additional sites.

Site storage

Individual sites are stored directly under the pages node. This default site root path is configured in the `assembler.properties` file. For example, in the Discover reference application, the path looks like the one in the following example:

```
preview.enabled=true
user.state.ref=previewUserState
media.sources.ref=authoringMediaSources
repository.configuration.path=./repository/${workbench.app.name}
defaultSiteRootPath=/pages
```

In the following configuration, the Discover application contains three sites: DiscoverAll, DiscoverPrinters and DiscoverCameras.

```
/pages
        /DiscoverAll
        /DiscoverCameras
        /DiscoverPrinters
```

Sites are the top-level content in this application. Each page within a site belongs to exactly one and only one site. Each site node is a site definition and every application must have at least one site definition. The Assembler uses properties in these site definitions to facilitate site-specific behavior. Site definitions are stored in the pages node of the IFCR. Our sample configuration would have three site definitions: one each for `DiscoverAll`, `DiscoverCameras`, and `DiscoverPrinters`.

```
/pages
        /DiscoverAll
            _.json
        /DiscoverCameras
            _.json
        /DiscoverPrinters
            _.json
```

The following site definition for DiscoverAll shows a unique display name, **DiscoverAll** and description, as well as a unique URL pattern, `/DiscoverAll` for this site:

```
{
  "ecr:type":"site-home",
  "urlPattern" : "/DiscoverAll",
  "displayName" : "DiscoverAll"
```

```
  "description" : "This site shows all things that are Discover."
}
```

If only a subset of records within the index are relevant for a specific site, then you must specify site-based filters. The site-based filter is applied to all queries performed on the site. The filter that is applied is determined by site context. The following configuration has three site definitions and two sites with site-based filters. One for DiscoverCameras and one for DiscoverPrinters. The filter for DiscoverCameras would filter out all records except those that were relevant to cameras, while the DiscoverPrinters filter would filter out all records except those that were relevant to printers. DiscoverAll includes every record in the index, so that site does not have a filter.

```
ifcr/
    /pages
        /DiscoverAll
              _.json
        /DiscoverCameras
              _.json
              filterState.xml
        /DiscoverPrinters
              _.json
              filterState.xml
```

The following is an example of the `filterState.xml` file for DiscoverCameras:

```
<Item class="com.endeca.infront.navigation.model.FilterState" xmlns="http://endeca.com/
schema/xavia/2010">
          <Property name="recordFilters">
            <List>
              <String>product.category: "Cameras"</String>
            </List>
          </Property>
</Item>
```

Note that Oracle does not support site-based filters in Oracle Commerce Guided Search.

Site Awareness

In a multiple site application, the Assembler must identify the site or the **site state** for incoming requests. This identity is included in the request as part of a domain name, a URL, or a request parameter. Resolving this site state gives the Assembler the ability to retrieve the relevant site definition, the site-based filter and other site specific information.

The `ActionPathProvider` is an interface that you are free to implement as you see fit. Cartridge handlers in applications built with Spring can use the `ActionPathProvider` to determine navigation or record detail action paths. The content paths that prefix navigation and record states are configured as sets of key-value pair mappings. In a multiple site application, the `ActionPathProvider` can define different mappings for pages on different sites. The following example shows an `ActionPathProvider` in the `assembler-context.xml` file with new mappings for the DiscoverCameras site that has been added to `navigationActionUriMap` and the `recordActionUriMap`:

```
<bean id="actionPathProvider" scope="request" class="com.endeca.infront.re¬
fapp.navigation.BasicActionPathProvider">
 <constructor-arg index="0" ref="contentSource"/>
```

```
 <constructor-arg index="1" ref="httpServletRequest"/>
<!-- navigationActionUriMap -->
 <constructor-arg index="2">
 <map>
    <entry key="^/pages/DiscoverCameras/.*$" value="/pages/DiscoverCameras/
camerasbrowse" />
    <entry key="^/pages/[^/]*/mobile/detail$" value="/mobile/browse" />
    <entry key="^/pages/[^/]*/services/recorddetails/.*$" value="/services/
guidedsearch" />
    <entry key="^/pages/[^/]*/detail$" value="/browse" />
    <entry key="^/services/.*$" value="/services/guidedsearch" />
 </map>
 </constructor-arg>
<!-- recordActionUriMap -->
 <constructor-arg index="3">
 <map>
    <entry key="^/pages/DiscoverCameras/.*$" value="/pages/DiscoverCameras/
camerasdetail" />
    <entry key="^/pages/[^/]*/mobile/.*$" value="/mobile/detail" />
    <entry key="^/pages/[^/]*/services/.*$" value="/services/recorddetails" />
    <entry key="^/pages/[^/]*/.*$" value="/detail" />
    <entry key="^/services/.*$" value="/recorddetails" />
 </map>
 </constructor-arg>
 <constructor-arg index="4" ref="siteState"/>
 </bean>
```

Creating Experience Manager Templates

This section describes the process of creating templates that enable the configuration of content items in Experience Manager.

Related links

- *Designing an Assembler Application* (page 17)
- About creating templates (page 27)
- Anatomy of a template (page 27)
- Template identifiers (page 29)
- About the group of a template (page 29)
- Specifying the description and thumbnail image for a template (page 29)
- Specifying the default name for a cartridge (page 30)
- Defining the content properties and editing interface (page 31)
- Structural properties (page 33)
- About keyword redirects groups (page 35)

About creating templates

Templates define the content structure of a content item as well as the editing interface that content administrators can use to configure instances of content items in Experience Manager.

In general, you create one or more templates that define the high-level structure of the pages in your application. These templates define sections that can be populated with other content items, or cartridges. Cartridge templates specify the properties required to display the content for that component. This may include values that the client application uses directly to render the information, or inputs into the Assembler for processing (such as query parameters to the MDEX Engine).

While cartridges and template properties typically determine aspects of the visual appearance of the page, keep in mind that they can also represent page elements that are not visible in the application. For example, a property can contain meta keywords used for search engine optimization, or a cartridge can include embedded code that does not render in the page but enables functionality such as Web analytics beaconing.

The Discover Electronics reference application provides sample page templates for some standard page types, as well as templates that enable configuration of the core set of cartridges in Experience Manager. These cartridges cover basic functionality, and are provided as a starting point for your application. You can customize them to suit your needs.

Note

In some cases, the reference application includes more than one template for the same functional cartridge. This is in order to enforce the proper constraints on which cartridges are available to insert in specific template slots. The only difference between the different versions of these templates is the template type.

This section concentrates on the basic template elements that enable you to create top-level page templates appropriate to your application. Details about the template configuration for core cartridges are covered in the "Feature Configuration" section. Reference information about the full range of properties and editors that can be used in templates is provided in the appendix to this guide.

Anatomy of a template

Top-level templates, which define an entire page, and cartridge templates, which drive the content of individual components, are JSON documents that share the same structure.

Templates can be broken down into four parts:

- **General information** such as the template group, description, and thumbnail image. This information is used in Experience Manager to help the content administrator select the appropriate template for a page or section.

```
"@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:58.542+05:30",
    "@thumbnailUrl": "thumbnail.png",
```

```
    "ecr:type": "template",
```

- **Default content item values.**Specify the default name of a cartridge and the default values of properties here.

```
"defaultContentItem": {
        "@name": "Results List",
        "relRankStrategy": "",
        "recordsPerPage": "10",
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
    },
```

- **Editor panel definition.** These allow you to define the editing interface in Experience Manager for this content item. Properties are generally associated with an editor that enables content administrators to configure the content items that they create within the tool.

```
"editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "editors/NumericStepperEditor",
                "label": "${property.recordsPerPage.label}",
                "maxValue": 100,
                "minValue": 10,
                "propertyName": "recordsPerPage"
            },
            {
                "editor": "editors/BoostBuryRecordEditor",
                "buryProperty": "buryStrata",
                "label": "${property.boostBury.label}",
                "propertyName": "boostStrata"
            },
            ...CONTENT REMOVED FROM EXAMPLE...
        ]
    }
```

- **Property type information.** In this part of the template, you explicitly declare all the properties of the content item that is described by this template. Property types can include Strings, Lists, and Booleans.

```
"typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"},
        "recordsPerPage": {"@propertyType": "String"},
        "relRankStrategy": {"@propertyType": "String"},
        "sortOption": {"@propertyType": "Item"}
    }
```

By defining the properties in the template along with how they can be configured in the tool, you ensure that the content configured in Experience Manager provides the necessary properties to the corresponding cartridge handler in the Assembler.

Template identifiers

Templates are saved as JSON files named `_.json` that are then uploaded to Experience Manager. Each template is required to have a unique identifier.

The unique template identifier is the folder name where the `_.json` file resides. For example, in `ThreeColumnNavigationPage\_.json`, the folder name, `ThreeColumnNavigationPage`, is the template identifier. The identifier appears as the name of the cartridge in the cartridge selector in Experience Manager. The value should be as descriptive as possible to help the user select the appropriate template, for instance, "ThreeColumnWithLargeBanner" or "HolidaySalePromotion."

Template folder names must be unique within your application. Templates with non-unique identifiers are not available in Experience Manager. Oracle recommends that you treat templates as part of your application's configuration and store them in a version control system. It can also be useful to include a template version number in a property for debugging.

About the group of a template

Each template has a `@group` that indicates where the template fits in an application page.

The group restriction serves two purposes. For top-level container templates, such as those that define a page, a group restriction can be specified for each section of the page. This limits the cartridges that can be inserted into that section. For example, if a template that includes a "SecondaryContent" section, only cartridges of the "SecondaryContent" group are available to insert into that section in Experience Manager.

Additionally, you can specify a template group in a dynamic slot to restrict the content that appears in that slot. This restriction applies at runtime when content items are evaluated against each other and ranked by priority for display in the application; any content items that do not match the specified template group for a dynamic slot are removed from consideration.

Setting a template group

The template `@group` is specified as a required attribute in the template `_.json`. For example, **HorizontalBanner**:

```
"ecr:type": "template",
    "@group": "SecondaryContent",
    "@description": "${template.description}",
    "@thumbnailUrl": "thumbnail.png",
```

Specifying the description and thumbnail image for a template

The description and thumbnail image for a template display in the template selector and cartridge selector dialog boxes in Experience Manager. Adding a description and thumbnail image to a template is required. Without them, template importation fails.

To specify the description and thumbnail image for a template, insert the following elements within the `_.json`:

Element	Description
`"@description"`	One or two brief sentences to help the content administrator identify the template in Experience Manager. This can include information about the visual layout of the template ("Three-column layout with large top banner") or its intended purpose ("Back to school promotion").
`"@thumbnailUrl"`	The absolute URL to a thumbnail image that shows a sample page or section that is based on the template. The images must be hosted on a Web server accessible from the Experience Manager server. Any URL without a protocol or leading slash will be treated as relative to the root of the template structure. If your thumbnail is in the same folder as your `_.json` file, you can omit the path altogether. For example, `"@thumbnailUrl":"thumbnail.png"`.

For example,

```
"ecr:type": "template",
    "@group": "Page",
    "@description": "A page layout with left and right sidebars intended for general
 category pages",
    "@thumbnailUrl": "thumbnail.png",
```

About using thumbnail images in Experience Manager

Thumbnail images can help the content administrator identify the appropriate template to use for the pages they create.

The suggested size for thumbnail images is 81 x 81 pixels; smaller images are stretched to fill this size and larger images are cropped to show only the top left corner.

The images must be hosted on a Web server accessible from the Experience Manager server. If the thumbnail image for a template is either not specified or not accessible, a default image displays in the dialog box.

Specifying the default name for a cartridge

The value of `@name` within the `defaultContentItem` displays as a label for the cartridge in the Content Tree in Experience Manager.

To specify a default name for a cartridge:

1. Insert the `@name` element inside `defaultContentItem`. In the following example a default name of Results List is specified:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:58.542+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Results List",
        "relRankStrategy": "",
        "recordsPerPage": "10",
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
    },
```

`@name` is a required element. The value you specify in the template becomes the default name when a content administrator creates the page or adds a cartridge. If you insert an empty `@name` element, an empty text field displays in Experience Manager and the content administrator can supply a value.

Defining the content properties and editing interface

A template defines the properties of a content item and also the interface that enables a content administrator to configure the properties.

You define properties within the `typeInfo` element in the template. For each property, you specify a name and a property type. You can optionally specify a default value for a property within the `defaultContentItem` element..

You associate editors with properties to enable the content administrator to configure their values within Experience Manager. Properties are generally primitive types such as Strings, Booleans, or Lists. Another type of property is a section, which allows content administrators to insert and configure another content item.

You can choose not to expose a particular property in Experience Manager and simply specify a default value to pass to the Assembler and ultimately to the client application. This is useful for values that do not need to be configured by the content administrator, but are needed by the Assembler for content processing or by the client application to determine how to render the content.

Template properties

You can define the properties of a content item within the `typeInfo` element. You can define the default values for these properties in the `defaultContentItem` element.

Cartridge properties are typically used for one of the following purposes:

- The property values may be intended to be used directly by the client application. For example, the content administrator may be able to enter text to use a heading or link text, or she may supply a URL to an image. Property values can also contain information such as meta keywords that are part of the page but do not affect its display.
- The values may be intended for the relevant cartridge handler in the assembler to use for processing, for example, parameters for a query to the MDEX Engine (or another external resource) to return the actual content that the application should display.
- Occasionally, a cartridge has no properties (and therefore no configuration options in Experience Manager), but exists only as a placeholder to indicate that a certain functional component should be included on a page. The Assembler inserts the necessary information for this cartridge at query time.

Each property must have a name that is unique within the template. If the property is to be passed through directly to the renderer, this can be any name that makes sense for your application. However, some properties are part of the configuration model for the cartridge. In this case the associated cartridge handler depends on the presence of specific properties in the template.

You specify the property type by adding an element of `@propertyType` in `typeInfo`.

The following example has three property elements nested within the Results List typeInfo item. The property elements are boostStrata, buryStrata, and sortOption.

```
"typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"},
        "sortOption": {"@propertyType": "Item"}
    }
```

The following example lists the default values for the sortOption property in the Results List defaultContentItem item.

```
"defaultContentItem": {
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
    },
```

Defining the editing interface for properties

After you have defined the content properties in your template, you can define how those properties can be configured by the content administrator in Experience Manager.

You add content editors inside the `editorPanel` element in the template. The `children` editors under the `"editor":"editors/DefaultEditorPanel"` element let you specify individual property editors that display in Experience Manager and associate them with a particular property.

For example, this excerpt from a sample template defines a property named `boostStrata`:

```
{
                "editor": "editors/BoostBuryRecordEditor",
                "buryProperty": "buryStrata",
                "label": "${property.boostBury.label}",
                "propertyName": "boostStrata"
            }
```

Editors are defined in templates with the `editor` namespace. By convention, the `propertyName` is a required attribute and specifies the property that this editor is associated with. The property must be defined in the `contentItem` part of the template, and must be of the appropriate type for that editor. If you define a content editor for a property that does not exist, or that is of the wrong type, a warning displays in Experience Manager when a content administrator attempts to configure the content.

Property editors do not have to be defined in the same order as the properties in the template. The `"editor":"editors/DefaultEditorPanel"` renders the editors in a vertical layout in Experience Manager, in the order in which you define them in the template. If you do not want a property to be exposed in the Experience Manager interface, do not define an editor associated with it.

It is possible to create more than one editor associated with the same property. However, be aware that all editors that you define in the template are displayed in Experience Manager, which may be confusing to the content administrator. When the value of a property is changed, any other editors associated with that property are instantly updated with the new value.

Related links

- Experience Manager editors mapping reference (page 209)

Configuring editor default values

You can configure default values for Experience Manager editors across the entire application by modifying cartridge templates.

This configuration applies to all instances of a specific editor created based on that template. For details about configuring the core editors packaged with Oracle Commerce Tools and Frameworks, see the "Template Property and Editor Reference" Appendix.

Related links

- *Template Property and Editor Reference* (page 209)
- Defining the editing interface for properties (page 32)

Structural properties

You can define a section within a template by inserting a `contentItem` or `contentItemList` element within a property.

Adding a content item property

A content item property defines a template section by creating a placeholder for a nested content item defined by a cartridge template.

Content administrators can configure a section in Experience Manager by choosing a cartridge to insert in the section then configuring the properties of the cartridge.

To add a content item property to a template:

1. Insert a `contentItem` element inside a property element.

2. Specify the section `@group`.

   Only cartridge templates with a group that matches the section group are presented as options for the content administrator to choose from in Experience Manager. For example, when a content administrator inserts a cartridge in a `SecondaryContent` section, only templates of the `SecondaryContent` group display in the Select Cartridge dialog box. (Recall that the cartridge template is the part of a cartridge that is exposed in Experience Manager).

The following example defines sections within a template. Note that more than one section in a template can have the same type, as long as your client application expects this kind of content.

```
{
    "ecr:type": "template",
    "@group": "Page",
    "@description": "${template.description}",
    "@thumbnailUrl": "thumbnail.png",
    "defaultContentItem": {
        "@name": "Three-Column Page",
-- additional elements deleted from this example --
        "leftContent": [],
        "rightContent": [],

-- additional elements deleted from this example --
```

Adding a content item list property

A content item list allows content administrators to add an arbitrary number of items to a section and to reorder those items within the list using the Content Tree in Experience Manager.

Using a content item list allows the content administrator to add an arbitrary number of content items to sections in this page:

```
"typeInfo": {
        "headerContent": {
            "@propertyType": "ContentItemList",
            "@group": "HeaderContent"
        },
        "leftContent": {
            "@propertyType": "ContentItemList",
            "@group": "SecondaryContent"
        },
        "mainContent": {
            "@propertyType": "ContentItemList",
            "@group": "MainContent"
        },
        "rightContent": {
            "@propertyType": "ContentItemList",
            "@group": "SecondaryContent"
```

```
        },
    }
```

To add a content item list to a template:

1. Insert a `contentItemList` element inside a property element.
2. Specify the template `@group`.

   Only cartridge templates with a group that matches the content item list group are presented as options for the content administrator to choose from in Experience Manager. In the above example, when a content administrator inserts a cartridge in a `rightColumn` section, only templates of the `SecondaryContent` group display in the Select Cartridge dialog box.

About cartridge selectors

Unlike other types of content properties, section properties are always editable; you do not need to explicitly specify an editor in the template.

In Experience Manager, content administrators can select cartridges to insert in sections either by clicking the cartridge Add button in the content detail panel or by right-clicking the section in the Content Tree. Both options bring up the cartridge selector dialog box and are enabled automatically when you define a section in the template.

About keyword redirects groups

Business users can configure a keyword redirect in Workbench to direct end users to a specified location in an application when they enter a specified search term or terms. This lets you display a relevant promotional page or product category page instead of a typical list of search results.

A keyword redirect triggers on one or more search terms; the target of a keyword redirect is a URL to a page in your application. This target page in your application is typically a search results page.

Keyword redirects are organized into groups. One is a default group. You can enhance your redirect experience by categorizing your keyword redirects into multiple keyword redirect groups. One search result page can be a target for one redirect group, while another search result page can be a target for another redirect group. Keyword redirect groups are **not** associated with the pages that have the search box on them; the groups are associated with the target search results page that the search box uses.

Organize your results pages into redirect groups to meet the needs of your application. Redirect groups are especially useful if you are creating an application with multiple sites. In a multiple site application, you can create a keyword redirect group exclusively for pages in each site. If you implement keyword redirect groups for sites, then the search results page that the end user is sent to can be site-specific and might not be appropriate for other sites in the application.

For example, an application has two sites: site A and site B. Site A and site B might each have their own contact information pages. A user shopping at site A that enters "contacts" in the search bar is redirected to a contact page that is unique to site A rather than a global contact page for the whole application.

Implementing keyword redirects in templates

To enable multiple keyword redirect groups in an application, include a redirect group property, `redirectGroup` when you create page templates. If you do not add this property, a default redirect group is used. In the future, however, if you want to configure search pages to use a specific redirect group, you must add

this property to the template and all content that is based on that template. At a minimum, Oracle recommends adding this property to your search results page templates and their corresponding content, but you can add this to any or all page templates as you see fit. If the property is added to a template, it must be added to any content based on the template, or business users see **Updated Template** warnings in Experience Manager.

Note

In the Discover reference application, since any page template can be used for a search results page and any search results page can be configured to use a specific redirect group, all page templates and corresponding content contain the `redirectGroup` property.

The following examples show a `redirectGroup` property in the typeInfo and defaultConentItem sections of the template.

```
"typeInfo": {
     "redirectGroup": {"@propertyType": "String"}
 }
```

```
"defaultContentItem": {
     "redirectGroup": ""
 }
```

This property lets you associate redirect groups with any search results pages created from this template. You must also edit the page content JSON file to associate a redirect group with pages in the application. If you do not add any redirect groups to the content JSON file then the page is automatically associated with the default redirect group. The following section lists the steps to follow to associate a redirect group with a search results page.

Associating keyword redirect groups with pages

Follow these instructions to associate keyword redirect groups with pages in your application.

Before you begin, verify that the template from which the pages were created has a redirect group property.

1. Export the application to which you want to add a keyword group.
   a. Navigate to the <app_dir>\control\ directory on Windows (<app_dir>/control/ directory on Unix).
   b. From the command line, export the application by entering the following command: `runcommand.<bat|sh> IFCR exportApplication <destination> true`.
2. In the destination directory, navigate to `\config\import\redirects\` and copy the `Default` folder.
3. Paste the copied folder into `\config\import\redirects\`

   For example, `\config\import\redirects\Default-Copy`.
4. Rename the copied folder with the name of your new redirect group.

   For example, `\config\import\redirects\DiscoverWest`
5. In the new redirect group folder, change the name of the `_Default.json` to `_.json`.
6. Use a text editor to update the keyword redirects group JSON with unique information appropriate for your group. At a minimum, enter a `displayName` value and verify that the `enableStemming` value is appropriate

for the keyword redirect group. You do not need to enter `redirects` values since business users can use the Workbench interface for that activity. See the *Oracle Commerce Administrator's Guide* for more information on properties in the redirect-group JSON file.

Be sure that the `displayName` property that you enter is meaningful to the business users that add keyword redirects manage permissions in Workbench.

In the following example, the redirect group has been updated with a unique display name.

```
{
    "ecr:type": "redirect-group",
    "displayName": "Discover West",
    "enableStemming": true,
    "redirects": [
        { "matchmode": "MATCHEXACT",
          "url": "/browse/west/_/N-1z141ya",
          "searchTerms": "west"
        }
     ]
}
```

7. Navigate to the JSON file for the relevant search results page. For example: `\import\pages \DiscoverElectronics\browse\_.json`.

8. Use a text editor to update the JSON file with the appropriate keyword redirect group. The group name must match the folder name where the redirects JSON is stored.

   For example, for the DiscoverWest redirects group, the redirects group string has the following value:

```
"contentItem": {
        "ruleLimit": "1",
        "@name": "Desktop Browse Page",
        "templateTypes": ["Page"],
        "@type": "PageSlot",
        "contentPaths": ["/content/Web"],
        "redirectGroup": "/redirects/DiscoverWest"
     }
```

   Note that `/redirects/DiscoverWest` matches the folder name where the redirects JSON file is stored.

9. Repeat the previous two steps for every page that you plan on using as a target for search results.

   Note

   Remember, the templates that these pages are based on must have a redirects group property.

10. Import the content with the new keyword group updates.

    a. Navigate to the `<app_dir>\control\` directory on Windows (`<app_dir>/control/` directory on Unix).

    b. From the command line, import the updated content by entering the following commands:

    - `runcommand.<bat/sh> IFCR importContent pages <path to source>`
    - `runcommand.<bat/sh> IFCR importContent redirects <path to source>`

- runcommand.<*bat/sh*> IFCR importContent templates <path to source>

For example:

```
runcommand.bat IFCR importContent pages c:\myexports\Discover\config\import\pages
```

```
runcommand.bat IFCR importContent redirects c:\myexports\Discover\config\import
\redirects
```

and

```
runcommand.bat IFCR importContent templates c:\myexports\Discover\config\import
\templates
```

If you fail to update either the template or content for the search results page, then business users editing content in the Experience Manager might see the following warning:

About multiple locales

If your implementation supports multiple locales, you can localize your custom templates.

You can create resource property files for each locale for storing localized strings. Each resource property file name must follow this format: `Resources_<locale>.properties` where <locale> is the ISO language code. For example `Resources_fr.properties` indicates that French values are stored in it. Place these files in a locales folder for your custom template: `<app dir>\config\cartridge_templates\<template_identifier>\locales`. You can specify values that do not change for locale (thumbnail URLs for example) in the single `Resources.properties` file or directly in the JSON file.

In the template itself, you can use `${property.name}` notation in element content and attributes to reference a localized string in the `Resources_<locale>.properties`. Only content in the `@description`, `@thumbnailURL`, and `editorPanel` sections can reference localized strings in the resources properties files.

The following example shows a template that uses notation to reference strings in resource properties files and two resource property files containing the stings that are being referenced.

```
{
    "ecr:type": "template",
    "@group": "MainContent",
    "@description": "${my.template.description}",
    "@thumbnailUrl": ${my.template.thumbnail}",
    "defaultContentItem": {
        ...CONTENT OMITTED FOR EXAMPLE...
     },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "editors/NumericStepperEditor",
                "label": "${my.template.recordsPerPage.label}",
                "maxValue": "100",
                "minValue": "10",
```

```
                "propertyName": "recordsPerPage"
            },
            {
                "editor": "editors/BoostBuryRecordEditor",
                "buryProperty": "buryStrata",
                "label": "${my.template.boostBury.label}",
                "propertyName": "boostStrata"
            },
        ...CONTENT OMITTED FOR EXAMPLE...
        ]
    }
}
```

The English resources property file, `Resources_en.properties`, for this template contains the following:

```
# Main Content
my.template.description = Container for main content cartridges.
my.template.recordsPerPage.label = Records Per Page
my.template.boostBury.label = Boost and Bury Records
```

In the template example, the thumbnail URL is the same for all locales, so the `${my.template.thumbnailurl}` notation is only referenced in the `Resources.properties` file.

```
# Main Content
my.template.thumbnailurl = /ifcr/tools/xmgr/img/template_thumbnails/maincontent2.jpg
```

Managing Experience Manager Templates

You must upload templates to Workbench before they are available to users in Experience Manager.

Updating Experience Manager templates

All deployment template applications include a `set_templates` script in the `control` directory to update Experience Manager templates. You run the script after you locally modify JSON template files and you want the templates available in Experience Manager. The script also imports locales and thumbnail files.

This script requires that the templates you modify are stored locally in `<app dir>\config\import \templates\`.

To send updated templates to Experience Manager:

1. In a command prompt, navigate to the `control` directory of your deployed application.

   This is located under your application directory. For example: `C:\Endeca\apps\<app name>\control`.

2. Run the `set_templates` script.

   For example:

```
C:\Endeca\apps\Discover\control>set_templates.bat
```

Troubleshooting problems with uploading templates

Template errors are detailed in the `ifcr.log` file.

The `ifcr.log` file is located in:

- `%ENDECA_TOOLS_CONF%\logs` on Windows
- `$ENDECA_TOOLS_CONF/logs` on UNIX

If any templates fail validation, the upload is canceled, and the previous templates remain in Workbench.

Troubleshooting invalid templates

Some templates may be successfully uploaded to Workbench, but still contain errors that lead to unexpected behavior in Experience Manager.

The most common scenario is when a property is associated with an editor that has constraints, such as a choice editor that can only accept certain string properties. If the default value of the property does not meet the editor's constraints, the editor may discard the value and display the following messsage in the Content Details Panel when a user adds the cartridge to a page:

```
Some fields or cartridges within this cartridge may have been
updated or removed. Your content has been converted to the new cartridge.
To accept these changes click OK and Save All Changes from the List View.
To reject these changes, click Cancel. For more information, see
"Troubleshooting pages" in the Oracle Workbench Help.
```

To avoid this message, ensure that all property defaults are valid options in the associated property editor.

About modifying templates that are used by existing pages

During the development and testing phase of your application deployment, you may need to make adjustments to your templates and update them in Experience Manager.

When Experience Manager populates the Content Detail Panel for a content item, it checks the content configuration of the loaded page against the template. If the template has been changed such that it is no longer compatible with the content, Experience Manager displays a warning and attempts to upgrade existing content to fit the new template definition.

Note

Existing configurations are not upgraded to the new template until a content administrator edits and saves the affected content item in Experience Manager.

Experience Manager does the following to ensure that the content and template are in sync:

- If a property has not changed its name or type, the existing values are migrated to the new template.
- If new properties are added to a template, any corresponding property editors become available in Experience Manager when a content administrator edits a content item based on the updated template. If you specify default values for the new properties, they are applied when a content administrator edits and saves the content item using the updated template.

- If properties are removed from a template, the corresponding property editors no longer display in Experience Manager when a content administrator edits a content item based on the updated template. The properties and their values are deleted from the page configuration.
- If the type of a property has changed (for example from string to list) within a template, the corresponding property editor (if one is specified) becomes available in the Experience Manager when a content administrator edits a content item based on the updated template. The existing value for the property does not display in Experience Manager until the content administrator saves the new value, replacing the previous value.
- If a content item or content item list property has changed to specify a different content type, then any existing cartridge in that section is ejected and its configured properties deleted.
- If the default value of an existing property has changed, it is only applied to new content items that are created based on the updated template. In existing pages, the previously saved value of the property (even if it is an empty string) is preserved regardless of whether it was originally a default or user-specified value.
- Some editors may implement specific update-handling logic in cases where an existing value does not meet the editor's constraints.

Note

Changing the `name` of a property is equivalent to removing the property with the old name and adding a property with the new name. Avoid changing the names of properties that are being used by existing pages. To change the display name of a property on Experience Manager, use the `label` attribute instead.

Managing template changes

Because existing content is not automatically updated to the new templates, and default values are never updated in existing pages, any changes that you make to your rendering code to reflect changes to a template should be backward-compatible. You can trigger the content upgrade process manually by accessing all affected content, but this approach is not recommended.

For this reason, you should avoid making changes to existing templates that are being used in production. You should limit updates to templates to the early stages of application development when you have little or no legacy content to support.

Retrieving the current templates from Experience Manager

If you need to view or edit an existing template on a local machine, you can run the `get_templates` script to download templates from Experience Manager to the local `<app dir>\config\cartridge_templates` `<app dir>\config\import\templates\` directory.

To get templates from Experience Manager:

1. In a command prompt, navigate to the `control` directory of your deployed application.

   This is located under your application directory. For example: `C:\Endeca\apps\<app dir>\control`.

2. Run the `get_templates` script.

3 Developing an Assembler Application

This part provides information for developing an Assembler application.

Related links

- Deploying the Assembler (page 43)

Deploying the Assembler

The Assembler can run in process as part of a Java application, or it can be deployed as a standalone servlet. This section covers both deployment options, as well as environment requirements and Assembler dependencies.

Assembler environment requirements

Review the requirements in this section before you deploy an Assembler.

Port usage

Before you begin your deployment, you might need to request an open port. You must assign a port for the Assembler client port. If this port is set to `-1`, the system uses an ephemeral port. An ephemeral port is allocated automatically for a short time and is used only for the duration of a communication session. When the session ends, the ephemeral port is available for another request.

For a complete list of ports used by Oracle Commerce, see the *Oracle Commerce Guided Search Administrator's Guide*.

Threads

The Assembler spawns threads to monitor and query various components for updates. This affects how you manage and prioritize threads.

About authoring and production environments

When designing your application and deploying the Assembler, consider the deployment requirements that come with maintaining an authoring environment and a live environment.

You should monitor the performance of your application and make adjustments as necessary to handle the expected load in a production situation.

Note

The Assembler has no dependencies on Workbench in a live environment; rule information is published to the MDEX Engine, and content items are exported from Workbench and maintained in an external location accessible from the live server(s). All live Assembler instances for a given application access the same exported content.

For additional information, including the necessary steps for exporting conent from Workbench, see the *Oracle Commerce Administrator's Guide*.

Assembler dependencies

Assembler dependencies are packaged in the `%ENDECA_TOOLS_ROOT%\assembler\lib` directory. You must include them in any custom Assembler application that you build.

The Assembler relies on the following libraries:

- AOP Alliance 1.0
- Apache Commons Logging 1.1.1
- Endeca Navigation API 6.5.1
- Endeca Logging API 11.1.0
- Endeca Rule Engine 11.3
- Spring AOP 3.0.1
- Spring ASM 3.0.1
- Spring Beans 3.0.1
- Spring Context 3.0.1
- Spring Core 3.0.1
- Spring Expression 3.0.1
- Spring Web 3.0.1

About deploying the Assembler

The Assembler can run in process as part of a Java application that powers a Web site, or it can be deployed as a standalone servlet. Non-Java applications must use the Assembler servlet.

The Tools and Frameworks package includes an example of each deployment mode in `/reference/discover-electronics` (for the Assembler running in process) and `/reference/discover-service` (for the standalone Assembler servlet). The standalone servlet, or Assembler Service, provides a RESTful interface for Assembler queries that returns results in either JSON or XML.

Both deployment modes depend on a Spring context file for application-specific configuration. The deployment descriptor files for the reference implementations specify a context file located in `/WEB-INF/assembler-context.xml`, as follows:

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<listener>
  <listener-class>
    org.springframework.web.context.request.RequestContextListener
  </listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/assembler-context.xml</param-value>
</context-param>
```

Assembler configuration

The Assembler implementation included with Tools and Frameworks is configured through Spring. The configuration in the Spring context file applies to both the in-process Assembler and the Assembler Service.

This guide assumes an application based around the included Assembler implementations. You can provide your own implementation if you need to use an alternate means of configuring the Assembler.

In the reference implementations, application-specific Assembler configuration is specified in the Spring context file located in `WEB-INF\assembler-context.xml`.

Assembler factory

The `AssemblerFactory` is an interface for creating a new Assembler. The reference implementation uses the `SpringAssemblerFactory` implementation and defines it as follows:

```
<bean id="assemblerFactory"
 class="com.endeca.infront.assembler.spring.SpringAssemblerFactory"
    scope="singleton">
    <constructor-arg>
        <bean class="com.endeca.infront.assembler.AssemblerSettings">
            <property name="previewEnabled" value="${preview.enabled}" />
            <property name="previewModuleUrl" value="http://${workbench.host}:
${workbench.port}/ifcr" />
        </bean>
    </constructor-arg>
</bean>
```

For details about the `AssemblerFactory` interface and the `SpringAssemblerFactory` implementation, see the **Assembler API Reference (Javadoc).**

About configuring cartridge handlers

A cartridge handler is an Assembler component that takes the configuration model for a specific cartridge and interacts with an external system to produce a response model. Cartridge handler configuration is a subset of Assembler configuration.

HTTP servlet request access

The `httpServletRequest` bean provides access to the `HTTPServletRequest` object for the current request.

```
<bean id="httpServletRequest" scope="request"
          factory-bean="springUtility"
          factory-method="getHttpServletRequest" />
```

Cartridge handlers that need access to the servlet request can specify a reference to this bean as follows:

```
<property name="httpServletRequest" ref="httpServletRequest" />
```

Search and navigation request configuration

The Assembler provides several utilities for parsing incoming requests and forming MDEX Engine queries.

MDEX resource configuration

The MDEX resource provides access to the MDEX Engine and manages information about the MDEX Engine and its schema configuration. Cartridge handlers can request data from their MDEX resource during the course of processing a cartridge.

The MDEX resource has the following properties:

MDEX resource property	Description
`appName`	The name of the application that the MDEX instance is associated with. Typically there is at least one MDEX per application.
`host`	The hostname or IP address of your MDEX Engine server.
`port`	The port on which the MDEX Engine server listens.
`sslEnabled`	Enables SSL communication for the MDEX Engine.
`recordSpecName`	The name of the property that serves as the record spec in your data set.

Navigation state builder configuration

The navigation state builder is responsible for parsing the request URL into a `NavigationState` object and for generating URLs based on a `NavigationState`.

Navigation state builder property	Description
`urlFormatter`	Specifies the `UrlFormatter` object to use for parsing the request URL into a `NavigationState` object and for generating URLs based on a `NavigationState`. **Note** In the Discover Electronics application, this bean is configured in `endeca-url-config.xml`.
`mdexRequestBuilder`	The `MdexRequestBuilder` implementation to use for forming MDEX Engine requests. For more information, see "About configuring cartridge handlers that make search and navigation queries."
`contentPathProvider`	Specifies the `ContentPathProvider` implementation that provides the URL path info for a navigation query or a record query. A reference implementation, `BasicContentPathProvider`, is included as part of Discover Electronics. As configured in the example below, it returns `/browse` for navigation queries and `/detail` for record detail queries.
`defaultSearchKey`	The name of a property, dimension, or search interface against which searches (using the Search Box cartridge) are performed.
`defaultMatchMode`	The match mode to use for text searches. Valid values for this property follow the syntax of URL parameters for search mode, without the `mode+match` prefix.
`siteState`	Identifies the current site using the `siteStateBuilder` configuration.
`siteManager`	Reference to the `siteManager` component, which is used by `NavigationStateBuilder` to look up the site definition for the currently active site.

`removeAlways`
 	
`removeOnUpdateFilterState`
 	
`removeOnClearFilterState`
	These properties configure which URL parameters from the request URL are preserved when generating action strings and which ones are removed, depending on the type of transition the action URL represents.

`recordDetailsDimensionNames`	A list of dimensions whose dimension values should be applied to the navigation state for a record query based on the values that are tagged on that record. This navigation state can be used for triggering configuration for the associated record detail page or for a spotlight cartridge that has the "restrict to refinement state" option enabled.

Filter state property	Description
`rollupKey`	A rollup key (used for aggregated records) to apply to all queries made with the default filter state.
`autoPhraseEnabled`	Specifies whether to apply automatic phrasing to text search queries. By default, automatic phrasing is enabled. For more information about automatic phrasing configuration, see "About implementing automatic phrasing" in this guide.
`securityFilter`	A default record filter to apply to MDEX Engine queries. For information about the record filter syntax, refer to the *MDEX Engine Development Guide*.
`languageId`	The language ID (as a valid RFC-3066 or ISO-639 code) to specify for MDEX Engine queries. For information about working with internationalized data, refer to the *MDEX Engine Development Guide*.

Filtering requests

The `NavigationState` object contains two filter states:

- `getUrlFilterState` - The filter state used for generating URL actions.
- `getFilterState` - The filter state used for combining the site-based filter (`filterState.xml`) and the filter for generating URL actions. See *Combining site-based filters and URL filters*.

For more information about filtering syntax, refer to the *Assembler API Reference(Javadoc)* content for the `NavigationState` interface.

Combining site-based filters and URL filters

Using `com.endeca.infront.navigation.NavigationState.getFilterState()` combines site-based filters and URL filters.

FilterState feature	Filter results
Search (Ntt, Ntk, Ntx)	Site and URL
Security (Cannot be security filter with a URL)	Site
Nav (N)	URL
Record (Nr)	Site and URL
Range (Nf)	Site and URL
Geo (Nfg)	URL
Featured Records (Rsel)	Site and URL
EQL Filter (Nrs)	URL

FilterState feature	Filter results
Rollup Key (Cannot be specified in a URL)	Site
Language ID (Ntl)	URL
Autophrase Enabled (Ntp)	URL

Site state builder configuration

The site state builder is responsible for identifying the current site or `SiteState` object. The site state builder iterates through all `siteStateParsers` and determines the current site or site state. Site state is referenced in Assembler components that must know the current site, for example, `NavigationCartridgeHandler` and `NavigationStateBuilder`.

`SiteStateBuilder` has the following properties:

Site state builder property	Descritption
`siteManager`	Retrieves site definitions.
`siteStateParsers`	A list of site state parsers that are run in the configured order to resolve `siteState`. Oracle provides the `RequestParamParser` and `URLPatternParser`. `RequestParamParser` returns `SiteState` if site id is provided by a request parameter called `siteId`. `URLPatternParser` returns `SiteState` by matching patterns configured on each site with the incoming request.
`defaultSiteStateParser`	Returns the default site for an application. This is only used if `siteStateParsers` fails to return a `SiteState`.
`contentPathTranslator`	Retrieves the page `contentPath` from a request, for example, `/browse`.

About configuring cartridge handlers that make search and navigation queries

Cartridge handlers that need to make MDEX Engine queries can reference the navigation state, record state, site state, user state, and MDEX request builder beans configured in the cartridge support section of the Spring context file.

The navigation state and record state represent the query parameters for each type of MDEX Engine query. The MDEX request builder consolidates requests from all the cartridge handlers in a single Assembler processing

cycle into as few MDEX queries as possible. These beans are defined in terms of previously configured beans; their configuration should not need to vary between applications.

The `NavigationCartridgeHandler` references the `navigationState`, `mdexRequestBuilder` and `siteState` beans for making navigation queries. The `RecordDetailsHandler` references the `recordState` for record detail queries. Cartridge handlers (including many of the core cartridges) that need access to the navigation state, record state, site state or the MDEX request builder typically extend one of these handlers. Note that `RecordDetailsHandler` itself extends `NavigationCartridgeHandler` as shown below, thereby inheriting the references to the navigation state and MDEX request builder specified in the `NavigationCartridgeHandler` bean.

```
<bean id="NavigationCartridgeHandler" abstract="true">
    <property name="navigationState" ref="navigationState" />
    <property name="mdexRequestBuilder" ref="mdexRequestBuilder" />
    <property name="mdexRequestBuilder" ref="mdexRequestBuilder" />
    <property name="actionPathProvider" ref="actionPathProvider"/>
    <property name="siteState" ref="siteState"/>
    <property name="userState" ref="${user.state.ref}"/>
</bean>

<bean id="CartridgeHandler_RecordDetails"
  class="com.endeca.infront.cartridge.RecordDetailsHandler"
  parent="NavigationCartridgeHandler" scope="prototype" >
    <property name="recordState" ref="recordState" />
</bean>
```

About configuring cartridges to retrieve dynamic content

Cartridge handlers that retrieve dynamic content based on trigger criteria can reference the content manager bean configured in the cartridge support section of the Spring context file.

The content manager depends on the content trigger state builder and its associated content trigger state, which perform similar functions to the navigation state builder and navigation state, only for the trigger query that retrieves dynamic content configuration, rather than the main navigation query.

Application-specific configuration for these beans relates to preview and auditing functionality. For more information about configuring preview, see "Setting up the Preview Application for Workbench."

The `ContentSlotHandler` references the content manager to make dynamic content queries. Other handlers that need to retrieve content items from a folder in Experience Manager should extend from this handler.

```
<bean id="CartridgeHandler_ContentSlot"
  class="com.endeca.infront.content.ContentSlotHandler"
  scope="prototype">
    <property name="contentManager" ref="contentManager" />
</bean>
```

About configuring the Assembler servlet

The Spring Assembler servlet extends the `AbstractAssemblerServlet` class, which requires a method for retrieving an `AssemblerFactory`, and another for retrieving a `ResponseWriter` that processes Assembler output.

The Assembler servlet references the same Spring configuration as the rest of the Assembler, with an additional dependency on response writer configuration.

Response writers

The Assembler servlet uses JSON or XML response writers to serialize the results of a query. The Assembler includes default implementations of a `JSONResponseWriter` and an `XMLResponseWriter`. You can provide your own implementation if you need to output the Assembler response to a different format (such as a different XML representation).

```
<bean id="jsonResponseWriter"
  class="com.endeca.infront.assembler.servlet.JsonResponseWriter"
  scope="singleton"/>

<bean id="xmlResponseWriter"
  class="com.endeca.infront.assembler.servlet.XmlResponseWriter"
  scope="singleton"/>
```

Reference implementations

The reference content includes two Web applications that run the Spring Assembler servlet with the appropriate configuration for Discover Electronics in either an authoring or a live environment:

- The implementation for an authoring environment is located at `reference\discover-service-authoring`.
- The implementation for a live environment is located at `reference\discover-service`.

Invoking the Assembler

This section describes how to invoke the Assembler in process or as a service.

Related links

- Invoking the Assembler in Java (page 51)
- Querying the Assembler Service (page 54)
- About building an Assembler query string (page 56)
- About retrieving Assembler results using the packaged services (page 56)
- About handling the Assembler response (page 66)

Invoking the Assembler in Java

You invoke the Assembler by passing in a content item object for assembly.

If a cartridge handler exists for the input content item, the Assembler invokes that handler to process it. If not, the content item is passed through as output. Upon invoking the cartridge handler, the Assembler might in turn invoke additional cartridge handlers to process child content items. The end result of the processing cycle is an output content item representing the Assembler response.

Note

If you have purchased Oracle Guided Search, you typically query the Assembler using one of the packaged services, either with a `ContentInclude` item or via the Assembler service.

The examples in this topic are specific to a Spring implementation of the Assembler.

To invoke the Assembler in Java:

1. Create an `AssemblerFactory` object.

   Note that the example implementation below first fetches configuration via the `WebApplicationContext` in the Spring framework:

```
// Get the Spring Web Application Context
 ServletContext servletCtx = this.getServletContext();
 WebApplicationContext webappCtx =
  WebApplicationContextUtils.getRequiredWebApplicationContext(servletCtx);

 // Get an assembler factory and create an assembler
 AssemblerFactory assemblerFactory =
  (AssemblerFactory)webappCtx.getBean("assemblerFactory", AssemblerFactory.class);
```

2. Use the `AssemblerFactory` to create an `Assembler`:

```
Assembler assembler = assemblerFactory.createAssembler();
```

3. Optionally, add event listeners to the newly-created `Assembler`:

```
assembler.addAssemblerEventListener(new MyLogger());
```

4. Pass in the content item object to assemble:

```
ContentItem responseContentItem = assembler.assemble(myContentItem);
```

Note

You can instantiate any content item programmatically and pass it to the Assembler, but typically an assembly cycle begins with a `ContentInclude` or `ContentSlotConfig` item. Both of these methods retrieve content items created in Workbench, the former by URI, and the latter by triggering content from a folder populated in Experience Manager.

After invoking the Assembler, you may wish to serialize the response:

```
// Serialize the results to JSON
 response.setCharacterEncoding("UTF-8");
 JsonSerializer serializer = new JsonSerializer(response.getWriter());
 serializer.write(responseContentItem);
```

The Assembler implementation included with Tools and Frameworks comes with two classes for this purpose, `JsonSerializer` and `XmlSerializer`. See the *Assembler API Reference (Javadoc)* for details.

Related links

- About retrieving Assembler results using the packaged services (page 56)

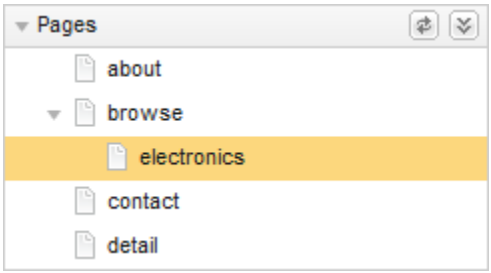
Invoking the Assembler with a ContentInclude item

A `ContentInclude` object specifies the URI from which to retrieve a content item.

In an authoring instance the content configuration is stored in the Endeca Configuration Repository. In a live instance, the Assembler retrieves content configuration from the live content source, specified in the configuration for the `ContentIncludeHandler`.

- In Oracle Experience Manager implementations, the URI begins with the path info from the request URL.
- In Oracle Guided Search implementations, the URI must begin with `/services` and specify one of the packaged Assembler services.

The `ContentIncludeHandler` retrieves the content that matches the deepest path in the URI. For example, if the request URL is `http://www.example.com/browse/electronics/Cameras`, the URI passed to the Assembler is `/browse/electronics/Cameras`. Suppose that the sitemap for this site looks like the following:



The cartridge handler first tries to retrieve the content at the exact URI. There is no content at that location, so it attempts to find the deepest matching path, which in this case is the content configuration at `/browse/electronics`. The Assembler then processes the content item at that location and returns the response for rendering.

Example 3.1. Example

The following example of a content include query retrieves page content for the Discover Electronics application with Experience Manager:

```
// Construct a content include to query the content source
// for content, given the path info of the request
ContentItem contentItem =
  new ContentInclude(request.getPathInfo());
```

Invoking the Assembler with a ContentSlotConfig item

A `ContentSlotConfig` object specifies one or more paths to a content folder in Experience Manager. The Assembler dynamically retrieves content items from the folder based on the trigger criteria and priorities set by the content administrator. It returns a number of content items equal to the evaluation limit configured for the specified content folder.

The Endeca Configuration Repository stores all Workbench content configuration for a given application within a `content` node. For example, the path to a Web → Spotlights content folder in the Discover Electronics reference application would be `content/Web/Spotlights`.

Example 3.2. Example

The following example creates a `ContentSlotConfig` object that is intended to populate the sidebar of an application page with three content items pulled from a Web → Spotlights content folder in Experience Manager:

```
ContentItem dynamicContentItem = new ContentSlotConfig();
dynamicContentItem.setContentPaths("/content/Web/Spotlights");
dynamicContentItem.setTemplateTypes("SecondaryContent");
dynamicContentItem.setRuleLimit(3);
```

It specifies a template type restriction to retrieve only "SecondaryContent" for the sidebar, but does not restrict results by template ID. This allows the query to pull in content items created from multiple cartridge templates, as long as those templates have the correct type; for example, it might return a Breadcrumbs cartridge, a Record Spotlight cartridge, and a Rich Text cartridge.

The call to the Assembler is the same as for any other content item:

```
ContentItem responseContentItem = assembler.assemble(dynamicContentItem);
```

Querying the Assembler Service

The Assembler Service provides a RESTful interface for making Assembler queries and retrieving results in either JSON or XML.

You query the Assembler Service by making a GET request to a URL that specifies the location of a content item that you wish to assemble. The URL should be of the following form:

```
http://[hostname:port]/[servlet-path]/[content-URI]?[query-params]
```

In the reference deployment of the Assembler Service, the servlet path determines the format of the Assembler response. The deployment descriptor file (`web.xml`) in the reference deployment defines two servlets:

Servlet path	Servlet description
`/json`	Returns the Assembler response as JSON.
`/xml`	Returns the Assembler response as XML.

The difference between the servlets is in the `ResponseWriter` implementation that they use. It is also possible to write an Assembler response writer that forwards the results to another servlet rather than serializing them.

The `content-URI` is the path to the content item to be assembled.

- In Experience Manager implementations, the URI begins with the path info from the request URL.
- In Oracle Guided Search-only implementations, the URI must begin with `/services` and specify one of the Assembler packaged services.

The Assembler servlet request URL `http://www.example.com/json/browse` is equivalent to passing a `ContentInclude` item to the Assembler `assemble()` method with the URI `/pages/[site-ID]/browse` and retrieving the results in JSON format.

Query parameters in an Assembler servlet request URL are processed the same way as in the embedded Java Assembler. For example, the URL `http://www.example.com/json/browse?N=101022` with the reference Assembler servlet deployment returns the same results as `http://www.example.com/discover/browse?N=101022` in the reference Java application.

Querying the Assembler Service in a multiple site deployment

If your Experience Manager implementation has multiple sites within an application, you must use a domain or URL pattern in your Assembler servlet request URL or pass a site ID parameter. For example, if your site uses a domain pattern for a cameras site, your request URL could be `http://cameras.discover.com/json/browse`. If your site uses a URL pattern for a cameras site, your request URL could be `http://localhost:8006/json/cameras/browse` where `/cameras` is the URL pattern.

To pass a site ID parameter, you can use this format `http://localhost:8006/json/browse?siteId=/DiscoverCameras` for a DiscoverCameras site.

Making dynamic content queries to the Assembler servlet

Unlike the Assembler in embedded mode, which allows assembly of any configuration content item, all queries to the Assembler servlet are treated as content include queries. To request content dynamically from a content folder based on a set of trigger criteria, you can create a content slot at a location in the sitemap that you can then specify in your Assembler request URL. In the reference implementation, the `browse` page is one example of a content item that is addressable by URI that then references content items within a specified folder path.

Related links

- Invoking the Assembler with a ContentInclude item (page 53)

The Assembler servlet response format

The Assembler provides response writer implementations that serialize the Assembler response to JSON or XML.

The Assembler response takes the form of a content item (that is, a map of properties). The properties are key-value pairs where the key is a string and the value may be one of the following types:

- String
- Boolean
- Integer
- List (of any property type)
- Item (a nested map of properties)

This structure makes it straightforward to serialize the response to JSON.

The XML output of the Assembler (using the default `XmlResponseWriter`) is not SOAP-compliant. The default XML format has the following characteristics:

- The root element of the response is `<Item>`.
- `<Item>` may have either a `type` attribute whose value is equivalent to the template `id` that defined the content item, or a `class` attribute in the case of a strongly typed reponse model for a content item.
- The child elements of `<Item>` are `<Property>` elements.
- Each `<Property>` element has a `name` attribute whose value is the property key, and contains either a `<String>`, `<Boolean>`, `<Integer>`, `<List>`, or `<Item>` element whose contents represent the property value.

For convenience, the Discover Electronics reference application provides links to the JSON and XML representations of the Assembler response, which are identical to the output of the Assembler servlet. This link can be useful for debugging purposes, but it is not recommended as a primary means of querying the Assembler for JSON or XML-formatted results.

About building an Assembler query string

Whether you invoke the Assembler programatically in Java or as a service, the content URI that you pass into the Assembler includes any MDEX Engine query parameters.

For more information about MDEX Engine query parameter syntax, refer to the *Assembler API Reference (Javadoc)* content for the `UrlNavigationStateBuilder` class.

About retrieving Assembler results using the packaged services

If you have purchased Oracle Commerce Guided Search (without Oracle Commerce Experience Manager), you must retrieve Assembler results via the packaged services.

These services are also available for Experience Manager implementations. In an Experience Manager implementation, the services must be located in the `/pages/<Default Site>/services/` directory. The packaged services include the following:

Service URI	Description
`/services/dimensionsearch`	Returns dimension search results based on a text search.
`/services/recorddetails`	Returns record detail information for a given record.
`/services/guidedsearch`	Returns search and navigation results including core features such as Guided Navigation, along with dynamic content returned from content folders.

You query the services by passing a `ContentInclude` item to the Assembler with the relevant service URI or making an Assembler servlet request specifying the service URI. The services are configured to return results for a specific cartridge or set of cartridges.

The cartridges that are returned by the services cannot be configured on a per-instance basis in Experience Manager, but application-wide default configuration for the cartridges can be specified based on your configuration framework (such as Spring). The exception is the dynamic content that can be configured in content folders and that is returned by the Guided Search Service, which can be configured in Experience Manager.

The services are populated in the Endeca Configuration Repository (for use by the authoring instance) when you run `initialize_services` after deploying an application. They are promoted to the live content source when you promote the site configuration for the live instance.

The Dimension Search Service

The Dimension Search Service returns dimension search results for a keyword search.

The service returns a single `DimensionSearchResults` object in a `dimensionSearchResults` property, representing the list of dimensions that match the search term.

The default behavior of this cartridge is configured as part of the `CartridgeHandler_DimensionSearchResults` bean in the Spring context file for the Assembler. For information about the configuration options for the Dimension Search Results cartridge, refer to the *Assembler API Reference (Javadoc)* for the `DimensionSearchResultsConfig` class.

This service exists for cases where you want to retrieve dimension search results only (such as in the case of an auto-suggest dimension search feature). Dimension search results are also returned as part of the response from the Guided Search Service.

The following is an example of the results of a Dimension Search Service query for the URI `http://localhost:8006/assembler-authoring/json/services/dimensionsearch?Ntt=fla*&Dy=1`, serialized to JSON:

```
{
    "@type": "DimensionSearchService",
    "name": "Dimension Search Service",
    "dimensionSearchResults": {
        "@type": "DimensionSearchResults",
        "totalNumResults": 13,
        "dimensionSearchGroups": [
            {
                "@class": "com.endeca.infront.cartridge.model.DimensionSearchGroup",
                "dimensionSearchValues": [ … ],
                "dimensionName": "camera.flash"
            },
            {
                "@class": "com.endeca.infront.cartridge.model.DimensionSearchGroup",
                "dimensionSearchValues": [ … ],
                "dimensionName": "product.features"
            },
            {
                "@class": "com.endeca.infront.cartridge.model.DimensionSearchGroup",
                "dimensionSearchValues": [ … ],
                "dimensionName": "product.category"
            }
        ]
    },
    "endeca:contentPath": "/services/dimensionsearch",
```

```
    "previewModuleUrl": "http://localhost:8006/preview"
}
```

The Record Details Service

The Record Details Service returns record detail information for a given record.

The service returns a single `RecordDetails` object in a `recordDetails` property, representing the details for a single record or an aggregate record.

The default behavior of this cartridge is configured as part of the `CartridgeHandler_RecordDetails` bean in the Spring context file for the Assembler. For information about the configuration options for the Record Details cartridge, refer to the *Assembler API Reference (Javadoc)* for the `RecordDetailsConfig` class.

The following is an Experience Manager example of the results of a record details service query for the URI `http://localhost:8006/assembler-authoring/json/services/recorddetails/Canon/Prima-Super-130U-Date/_/A-266556`, serialized to JSON:

```
{
    "@type": "RecordDetailsService",
    "name": "Record Details Service",
    "recordDetails": {
        "@type": "ProductDetail",
        "record": {
            "@class": "com.endeca.infront.cartridge.model.Record",
            "numRecords": 1,
            "attributes": { … },
            "records": [
                {
                    "@class": "com.endeca.infront.cartridge.model.Record",
                    "numRecords": 0,
                    "attributes": { … }
                }
            ]
        }
    },
    "endeca:siteRootPath": "/pages",
    "endeca:contentPath": "/services/recorddetails",
    "previewModuleUrl": "http://localhost:8006/preview",
    "endeca:assemblerRequestInformation": { … }
}
```

In a Guided Search implementation without Experience Manager, the site root path would be `/services`.

```
"endeca:siteRootPath": "/services",
"endeca:contentPath": "/recorddetails"
```

The Guided Search Service

The Guided Search Service returns search and navigation results including core features such as Guided Navigation, along with dynamic content returned for content slots.

The properties returned as part of the response model, as well as their associated configuration, are listed below:

Property name	**Response model,** **Configuration Bean,** **Configuration model**
navigation	GuidedNavigation CartridgeHandler_GuidedNavigation GuidedNavigationConfig
breadcrumbs	Breadcrumbs CartridgeHandler_Breadcrumbs BreadcrumbsConfig
resultsList	ResultsList ResultsListConfig CartridgeHandler_ResultsList
searchAdjustments	SearchAdjustments CartridgeHandler_SearchAdjustments SearchAdjustmentsConfig
dimensionSearchResults	DimensionSearchResults CartridgeHandler_DimensionSearchResults DimensionSearchResultsConfig
zones	Depends on contents of referenced content folders. CartridgeHandler_ContentSlotList ContentSlotConfig

The following is an example of the results of a guided search service query for the URI `http://localhost:8006/assembler-authoring/json/services/guidedsearch?Ntt=pink+camera`, serialized to JSON:

```
{
    "@type": "GuidedSearchService",
    "name": "Guided Search Service",
    "navigation": {
        "@type": "GuidedNavigation"
    },
    "breadcrumbs": {
        "@type": "Breadcrumbs",
        "removeAllAction": "/services/guidedsearch",
        "refinementCrumbs": [ ],
        "searchCrumbs": [ … ],
        "rangeFilterCrumbs": [ ]
    },
    "resultsList": {
        "@type": "ResultsList",
        "totalNumRecs": 213,
        "sortOptions": [ … ],
        "firstRecNum": 1,
        "lastRecNum": 10,
        "pagingActionTemplate": "/services/guidedsearch?No=%7Boffset%7D&Nrpp=
%7BrecordsPerPage%7D&Ntt=pink+camera",
        "recsPerPage": 10,
        "records": [ … ]
    },
    "searchAdjustments": {
        "@type": "SearchAdjustments",
        "originalTerms": [
            "pink camera"
        ]
    },
    "zones": {
        "@type": "ContentSlotList"
    },
    "endeca:contentPath": "/services/guidedsearch",
    "previewModuleUrl": "http://localhost:8006/preview"
}
```

Note

For details about the contents of the `zones` property, see "About dynamic content and the Guided Search Service."

Configuring dynamic content for the Guided Search Service

For each dynamic slot that you wish to populate as part of the response from the Guided Search Service, you must configure a `ContentSlotConfig` object. Each of these objects is set as a property of the default input content item for the `ContentSlotHandler`.

Specify the following properties for each instance of `ContentSlotConfig`:

Property name	Value
`contentPaths`	A List of String typed paths to the content folders from which you want to return results.
`templateTypes`	(Optional) A `List` of `String` typed template type restrictions for the dynamic slot.
`templateIds`	(Optional) A `List` of `String` typed template ID restrictions for the dynamic slot.
`ruleLimit`	The maximum number of content items to return from this collection. The Assembler returns up to this number of items that match the trigger criteria, based on priority.

Note

The content within a folder depends on the template type or ID restrictions configured for that folder in Experience Manager. While it is possible to configure your default `ContentSlotConfig` objects with any restrictions you wish, you should ensure that the type and ID restrictions you enter match those in Experience Manager. For example, it is possible to create a `ContentSlotConfig` object that is restricted by template type "MainContent," while the `contentPaths` property points to folders in Experience Manager that are restricted to "SecondaryContent" (and thus will never contain any "MainContent" content items).

Example 3.3. Example

In the example below, the input content item to the `ContentSlotHandler` is a `ContentSlotListConfig` object. It is instantiated as "contentSlotList," and contains a `ContentSlotConfig` object for each dynamic slot in the application. The `contentSlotList` is passed in to the `ConfigInitializer` that instantiates it as the input content item for the cartridge handler.

The `contentSlotList` for the Discover Electronics reference application is configured in the `CartridgeHandler_ContentSlotList` bean in the Spring context file, `assembler-context.xml`. For each content folder that is enabled for the Guided Search Service, a `ContentSlotConfig` bean appears in the `contentSlotList` as in the example below:

```
<bean id="CartridgeHandler_ContentSlotList"
 class="com.endeca.infront.content.ContentSlotListHandler"
 scope="prototype">
    <property name="contentItemInitializer">
        <bean class="com.endeca.infront.cartridge.ConfigInitializer" scope="request">
            <property name="defaults">
                <bean class="com.endeca.infront.content.ContentSlotListConfig"
 scope="singleton">
                    <property name="contentSlotList">
                        <list>
                        <bean class="com.endeca.infront.content.ContentSlotConfig"
 scope="singleton">
                                <property name="contentPaths">
                                  <list>
                                    <value>/content/Right Column Spotlights</value>
                                  </list>
                                </property>
                                <property name="templateTypes">
                                  <list>
                                    <value>SecondaryContent</value>
                                  </list>
                                </property>
                                <property name="templateIds">
                                  <list>
                                    <value>RecordSpotlight</value>
                                    <value>RichTextSecondary</value>
                                  </list>
                                </property>
                                <property name="ruleLimit" value="3"/>
                            </bean>
                        </list>
                    </property>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

For detailed information about the `ContentSlotConfig` configuration model and its included properties, see the *Assembler API Reference (Javadoc)*.

Handling the Guided Search Service response

The Assembler returns the matching content items for each configured `ContentSlotConfig`, so the response consists of a list of lists of content items:

- `ContentSlotList` response content item
  - 1st content item, returned from a `ContentSlotConfig` with a `ruleLimit` of 3
    - Highest priority matching content item
    - Second highest priority matching content item
    - Third highest priority matching content item
  - 2nd content item, returned from a `ContentSlotConfig` with a `ruleLimit` of 2
    - Highest priority matching content item
    - Second highest priority matching content item

Note that the Guided Search Service response is not view-friendly. You must parse it in your application logic to determine if any of the content items returned in the tree correspond to page sections you wish to populate for the end user's current location in the application.

Below is a sample JSON response from the Guided Search Service in the Discover Electronics reference application when the user selects the "Cameras" category:

```
"zones": {
    "@type": "ContentSlotList",
    "contentSlotList": [
        {
            "@type": "ContentSlot",
            "templateTypes": [
                "RecordSpotlight"
            ],
            "contents": [
                {
                    "@type": "RecordSpotlight",
                    "title": "Most Popular Cameras",
                    "name": "Spotlight Records",
                    "records": [
                        { … },
                        { … },
                        { … },
                        { … }
                    ]
                },
                {
                    "@type": "RecordSpotlight",
                    "title": "Top Rated Products",
                    "name": "Spotlight Records",
                    "records": [
                        { … },
                        { … },
                        { … }
                    ]
                }
            ],
            "contentPaths": [
                "/content/Right Column Spotlights"
            ],
            "ruleLimit": 3,
            "templateIds": [ ]
```

```
        }
    ]
},
```

It populates two sidebar Record Spotlight cartridges, the first with four records, and the second with three.

About retrieving content item properties from packaged services

This topic outlines the logic required for retrieving properties from the Assembler response model for the included Guided Search Service.

The example below includes processing logic within a renderer JSP file. Oracle recommends including most of your logic for handling Assembler responses in your cartridge handlers, as this minimizes the amount of duplicate code required across multiple renderers.

> **Note**
>
> API documentation for the Assembler packages is available in the `assembler\apidoc\assembler` directory of your Tools and Frameworks installation.

Retrieving information from the Assembler response

Recall the serialized Assembler response for the URI `http://localhost:8006/assembler-authoring/json/services/guidedsearch?Ntt=pink+camera`:

```
{
    "@type": "GuidedSearchService",
    "name": "Guided Search Service",
    "navigation": {
        "@type": "GuidedNavigation"
    },
    "breadcrumbs": {
        "@type": "Breadcrumbs",
        "removeAllAction": "/services/guidedsearch",
        "refinementCrumbs": [ ],
        "searchCrumbs": [ … ],
        "rangeFilterCrumbs": [ ]
    },
    "resultsList": {
        "@type": "ResultsList",
        "totalNumRecs": 213,
        "sortOptions": [ … ],
        "firstRecNum": 1,
        "lastRecNum": 10,
        "pagingActionTemplate": "/services/guidedsearch?No=%7Boffset%7D&Nrpp=
%7BrecordsPerPage%7D&Ntt=pink+camera",
        "recsPerPage": 10,
        "records": [ … ]
    },
    "searchAdjustments": {
        "@type": "SearchAdjustments",
        "originalTerms": [
            "pink camera"
        ]
    },
    "zones": {
        "@type": "ContentSlotList"
```

```
    },
    "endeca:contentPath": "/services/guidedsearch",
    "previewModuleUrl": "http://localhost:8006/preview"
}
```

To create a sample JSP file that invokes the Assembler:

1. Import the required packages and create the necessary objects for supporting the Assembler:

```
<%@page language="java" contentType="text/html; charset=UTF-8" %>
<%@page import="com.endeca.infront.assembler.Assembler"%>
<%@page import="com.endeca.infront.assembler.AssemblerFactory"%>
<%-- additional imports removed from this example --%>
<%@page import="org.springframework.web.context.WebApplicationContext"%>
<%@taglib prefix="discover" tagdir="/WEB-INF/tags/discover" %>
<%
    // Create the Web Application Context object
    WebApplicationContext webappCtx =
 WebApplicationContextUtils.getRequiredWebApplicationContext(application);

    // Get the AssemblerFactory from the Spring context file
    AssemblerFactory assemblerFactory =
 (AssemblerFactory)webappCtx.getBean("assemblerFactory");
```

2. Recall that the packaged services invoke the Assembler with a `ContentInclude` item. Create the `assembler` object and the `ContentInclude` item, and pass it into the Assembler as the configuration model:

```
    // Create an Assembler object
    Assembler assembler = assemblerFactory.createAssembler();

    // Construct a content include item for the Guided Search service
    ContentItem contentItem = new ContentInclude("/services/guidedsearch");

    // Assemble the content
    ContentItem responseContentItem = assembler.assemble(contentItem);
```

   The Assembler returns a `com.endeca.infront.assembler.ContentItem` interface as the response model; this extends the Java `Map` interface. Assign this response to a `responseContentItem` object.

3. `get` the `resultsList` object from the `responseContentItem`:

```
    ContentItem resultsListItem = (ContentItem)
 responseContentItem.get("resultsList");
```

   This retrieves the top-level `resultsList` object, which is itself an extension of `BasicContentItem`, from the Assembler response.

4. You can now retrieve and use the individual values stored on the `resultsList` object, for example, the total number of records:

```
    String totalNumRecs = resultsListItem.get("totalNumRecs");
```

This assigns a value of "213" to the totalNumRecs variable (based on the sample response presented at the start of this topic). Similarly, you could retrieve any other value from the key/value pairs that comprise resultsList, including other objects that extend the Map interface and are themselves made up of key/value pairs.

Refer to the Assembler API documentation for additional information about available Assembler interfaces, implementations, and methods. Following the pattern described in Steps 3-4, you can continue to retrieve values from the Assembler response by calling the get method on the response model object to traverse the nested values.

About handling the Assembler response

As a best practice, your application should have modular renderers to handle the response model for each content item.

A typical page consists of a content item that contains several child content items representing the individual feature cartridges. The Discover Electronics application maps each response model to the proper renderer by convention, based on the @type. The model @type corresponds to the template identifier (the directory name) of the template that was used to configure it. (Recall that the template type determines where a cartridge can be placed in another content item, while the template ID uniquely identifies the cartridge and its associated content definition.) For each cartridge, the associated renderer is located in WEB-INF/views/<channel>/<TemplateID>/<TemplateID>.jsp. For example, the renderer for the Breadcrumbs cartridge is located in WEB-INF/views/desktop/Breadcrumbs/Breadcrumbs.jsp.

In the Discover Electronics application, this logic is implemented in include.tag. Your application should implement a similar mapping of response models to their corresponding rendering code.

Source code for the renderers in the Discover Electronics application is provided as an example of how to work with the model objects returned by the Assembler in Java. The sample rendering code is intentionally lightweight, enabling it to be more easily customized for your own site. For information about the response models for the core cartridges, refer to the *Assembler API Reference (Javadoc)*.

Some features in the Discover Electronics application are designed with certain assumptions about the data set, such as property and dimension names. Mirroring the Discover Electronics data schema for your own data can facilitate reuse of the reference cartridges, reducing the need to update rendering logic and Assembler configuration for your data set.

About rendering the Assembler response

As soon as you have retrieved the necessary information for your page, Oracle recommends subdividing your view logic to correspond to the hierarchy of content items returned by the Assembler.

The renderer for the Three Column Navigation Page content item in Discover Electronics provides an example of the page rendering process as implemented in the reference application. It is located in your Tools and Frameworks installation directory under reference\discover-electronics-authoring\WEB-INF\views\desktop\ThreeColumnPage\ThreeColumnPage.jsp. You can use this JSP file as a point of reference for developing your own application pages. While the details are specific to the Discover Electronics implementation of the Assembler API, your general approach should be similar.

Recall that each of the elements that make up the page uses a custom <discover:include> tag, defined in WEB-INF\tags\discover\include.jsp, to include the rendering code for the associated page component:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
DTD/xhtml1-strict.dtd">
```

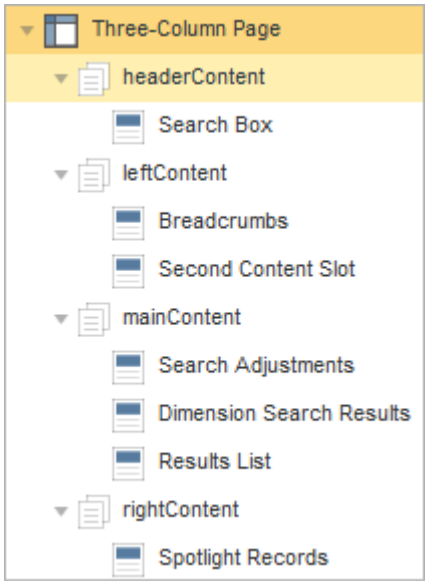
```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <!-- Additional elements removed from this sample -->
</head>
<body>
    <endeca:pageBody rootContentItem="${rootComponent}">
        <div class="PageContent">
            <%--include user panel --%>
            <%@ include file="/WEB-INF/views/userPanel.jsp" %>
            <%--include user page logo --%>
            <%@ include file="/WEB-INF/views/pageLogo.jsp" %>
            <div class="PageHeader">
                <c:forEach var="element" items="${component.headerContent}">
                    <discover:include component="${element}"/>
                </c:forEach>
            </div>
            <div class="PageLeftColumn">
                <c:forEach var="element" items="${component.leftContent}">
                    <discover:include component="${element}"/>
                </c:forEach>
            </div>
            <div class="PageCenterColumn">
                <c:forEach var="element" items="${component.mainContent}">
                    <discover:include component="${element}"/>
                </c:forEach>
            </div>
            <div class="PageRightColumn">
                <c:forEach var="element" items="${component.rightContent}">
                    <discover:include component="${element}"/>
                </c:forEach>
            </div>
            <div class="PageFooter">
                <%--include copyright --%>
                <%@include file="/WEB-INF/views/copyright.jsp" %>
            </div>
        </div>
    </endeca:pageBody>
</body>
</html>
```

For the example above, the JSP is composed as follows:

1. The static `<div class="UserPanel">` and `<div class="PageLogo">` elements are included from the specified JSP files.

2. The `<div class="PageHeader">` element retrieves the list of `headerContent` content items from the component.

    - In an Oracle Experience Manager installation, this is the list of content items defined by the content administrator in Experience Manager:



- In an Oracle Guided Search installation, this is the list of content items specified application-wide under `WEB-INF\services\browse.jsp`:

```
<div class="PageContent">
    <%--include user panel --%>
    <%@ include file="/WEB-INF/views/userPanel.jsp" %>
    <%@ include file="/WEB-INF/views/pageLogo.jsp" %>

    <div class="PageHeader">
        <discover:include component="${searchBox}"/>
    </div>
    <div class="PageLeftColumn">
        <discover:include component="${component.breadcrumbs}"/>
        <discover:include component="${component.navigation}"/>
    </div>
    ...
```

3. For each of the included content items, the JSP includes the output of the associated renderer.

4. The `<div class="PageLeftColumn">`, `<div class="PageCenterColumn">`, and `<div class="PageRightColumn">` elements are included in the same fashion.

5. The static `<div class="Copyright">` element is included from the specified JSP file.

Implementing Multichannel Applications

This section covers how to design and implement multichannel applications built on the Assembler and managed using Workbench with Experience Manager.

Related links

- Overview of multichannel applications with the Assembler (page 69)
- About creating templates for mobile channels (page 69)

Overview of multichannel applications with the Assembler

The Assembler provides an API for delivering content across an entire site, allowing content configuration to be shared between channels when appropriate, and also enabling a more targeted channel-specific experience where desired.

Enabling the full flexibility of the cross-channel experience involves the following:

- **Creating channel-specific templates.** Content administrators may wish to configure different features or cartridges for different channels. For example, pages designed for mobile devices typically have a simpler structure and present fewer options than pages designed for desktop Web.
- **Writing channel-specific rendering code.** Due to the limitations of mobile browsers and device bandwidth, renderers for mobile Web applications are typically more lightweight than those for desktop Web, while native applications for mobile devices require platform-specific client code.
- **Enabling device detection.** The UI tier of your Assembler application should include logic for handling device detection. Typically, this also includes redirecting a client to the appropriate service for their user agent.

About creating templates for mobile channels

Templates for mobile-specific content in a multichannel application can give content administrators the flexibility to manage channel-specific content in Experience Manager. When planning the set of templates for your application, however, use more general templates whenever possible in order to share configuration across multiple channels.

The following general practices help enable this combination of flexibility and consistency:

- Create different top-level page templates for channels that have a different high-level structure. For example, the same range of cartridges may be appropriate for pages designed to be displayed on desktop computers but not for pages designed to be displayed on mobile devices. Native applications for mobile devices may display content in simplified "pages" that differ from those intended for Web browsers.
- Use dynamic slots for configuration that should be shared across channels, because they enable reuse of content between pages. For example, if the same refinement configuration (such as overall dimension order, refinement ordering, and boost and bury options) should apply at a specific navigation state regardless of channel, it may make sense to configure it within a separate content folder and reference it from the appropriate pages for each channel.

To enable the greatest flexibility in Experience Manager while ensuring that content administrators create configurations that are appropriate to each channel, you can restrict the cartridges that can be placed on a page or in a content folder by content type. These content types may vary depending on the intended purpose of a page or dynamic slot. For example, you may have the following in your application:

- Page templates for desktop Web, which may define a section of type `SecondaryContent`. This section may be populated with Guided Navigation cartridges, Spotlight cartridges, or dynamic slots serving as a placeholder for either type.

- A content folder designed for Guided Navigation cartridges. This is similar to the Navigation section of the mobile page, but it should not allow a content administrator to create a dynamic slot within a dynamic slot, so it should have a third content type (such as `Navigation`) to enforce this restriction.

In most cases, the set of Dimension Navigation cartridges in an application should be identical. Variance between different output channels tends to manifest at the page design level, rather than at the level of the individual components of a page.

Tuning an Assembler application

The Assembler and the MDEX Engine both include logging functionality that you can use to debug and fine tune your application. In addition, Workbench includes Preview functionality that your Content Administrator can use to evaluate the results of their changes.

Related links

- Enabling the preview application for Workbench (page 70)
- Configuring logging for an Assembler Application (page 87)
- Configuring cartridge performance logging (page 95)
- Debugging MDEX Engine query results (page 95)

Enabling the preview application for Workbench

If you are using Experience Manager, you can use a preview application to simulate sets of trigger conditions, such as time-based triggers, in order to determine which content items are displayed when specific conditions are met. This section describes how to set up a custom application to function as the preview application in Workbench.

About the preview application

The preview application enables content administrators to determine whether each content item is or is not displayed by particular combinations of navigation queries and triggers. This chapter describes how to set up your own custom application as the Workbench preview application.

You can start the preview application for a specific page or for an individual cartridge. A selected cartridge is displayed in the context of a page that includes it.

The preview application does not need to be an exact representation of your final front-end application if it uses the correct data. The business logic that is built into Workbench is not tied to the physical representation of the front-end application. It is good practice, however, to make sure that your preview application represents your final application closely enough to enable business users to verify that their changes are correct.

By default, Workbench is configured to use the Discover Electronics reference application as the preview application. This application is located under `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring` (`$ENDECA_TOOLS_ROOT/reference/discover-electronics-authoring` on UNIX).

Workbench communicates with the preview application via settings you specify in the Preview Settings tool. The Preview URL field lets you specify the preview application URL.

Note

The preview application must not use frames, because they are likely to collide with the frames of the Workbench preview toolbar.

Enabling your Java application for preview

In order to enable auditing and editing in your custom preview application, your JSP file rendering code must include logic for adding preview frames and buttons for auditing and editing content items.

Your custom preview application should include tags that specify paths to the required JavaScript and CSS resources, as well as tags for enabling audit and edit functionality. These are provided in the tag library.

These requirements assume an application that uses JSP files for cartridge renderers (as in the case of the Discover Electronics reference application). If you are using a different technology stack to implement your Assembler application, you must write your own auditing functionality. See Enabling non-Java applications for preview (page 72).

Adding Preview resources

All JSP files must include the tag library, as shown below:

```
<%@ taglib prefix="endeca" uri="/endeca-infront-assembler/utilityTags"%>
```

Each <head> tag must contain a reference to the pageHead tag. This includes paths to the Preview JavaScript and CCS files:

```
<head>
    <endeca:pageHead rootContentItem="${rootComponent}"/>
    <title><c:out value="${component.title}"/></title>
    <meta name="keywords" content="${component.metaKeywords}" />
    <meta name="description" content="${component.metaDescription}" />
</head>
```

Enabling auditing and editing

All slots and content items must include a PreviewAnchor tag that wraps them in a div class that contains preview information. This tag requires the current content item element and enables audit and edit functionality. Oracle recommends that instead of including this in every renderer, have a centralized place where content items are dispatched. In the Discover reference application, this is done in the include.tag:

```
<%-- save the parent's component currently in request scope into page scope --%>
<c:set var="parentComponent" scope="page" value="${requestScope['component']}"/>
<%-- set the content item the child will use as this one (the one passed in the tag) --
%>
<c:set var="component" scope="request" value="${component}"/>
<c:catch var="importException">
    <endeca:previewAnchor contentItem="${component}">
        <c:import url="${resourcePath}" charEncoding="UTF-8"/>
    </endeca:previewAnchor>
</c:catch>
```

Each <body> tag must contain a reference to the pageBody tag. This tag requires the root and current content item elements and enables audit and edit functionality:

```
<script type="text/javascript" src="<c:url value='/js/global.js'/>"></script>
</head>
<body>
    <endeca:pageBody rootContentItem="${rootComponent}" contentItem="${component}">
        <div class="PageContent">
            <%--include user panel --%>
            <%@ include file="/WEB-INF/views/userPanel.jsp" %>
```

Device-specific auditing and editing

In order to handle preview for different devices, you must implement conditional rendering logic for different user agent strings. The rendering code should include the tags described in the previous section.

You can retrieve the user agent String by getting a reference to the `UserState` object and calling `getUserAgent()` on it. The `UserState` class is documented in the Javadoc for the `com.endeca.infront.content` package.

For example, the Discover Electronics reference application includes the following logic in the `WEB-INF\services\assemble.jsp` page:

```
                    UserState userState =
 webappCtx.getBean(properties.getProperty("user.state.ref"), UserState.class);

String userAgent =  userState.getUserAgent();

//If the userAgent is null, then no user-agent was specified and we need to get the
 user agent from the request header.
if(userAgent == null){
    userAgent = request.getHeader("user-agent");
}
```

Decorating the page

Preview requires a request attribute to decorate the page. For example, the Discover Electronics reference application includes the following logic in the `WEB-INF\services\assemble.jsp` page. The last line retrieves the value of the `preview.enabled` property from the `assembler.properties` file. The value of the constant `PreviewAnchor.ENDECA_PREVIEW_ENABLED` is *`endeca:previewAnchorsEnabled`*.

```
//If the userState has no specified userAgent, use the one from the request header.
    if(userAgent == null){
        userAgent = request.getHeader("user-agent");
    }

    request.setAttribute(PreviewAnchor.ENDECA_PREVIEW_ENABLED,
 Boolean.valueOf(properties.getProperty("preview.enabled")));
############################################################
```

Enabling non-Java applications for preview

This section describes how to enable hotspots in any Assembler-based Web application that lets a user audit and edit cartridges and slots in Experience Manager.

The preview JavaScript framework should be added to the Assembler application's Web pages to enable this behavior. The framework supports single page applications (SPAs) and the popular module loader *RequireJS*. Note that the framework depends on the open source *JQuery* library. If your application already uses a JQuery version higher than 1.9, and if its available on the window object, you do not have to include the JQuery version provided by Oracle Commerce.

Preview CSS

Include the following style sheet in the head of your page:

```
<link rel="stylesheet" href="http://@@WORKBENCH_HOST@@:@@WORKBENCH_PORT/ifcr/tools/
xmgr/app/preview/css/audit-site.css" />
```

Standard Web page

If you do not use an AMD module loader, include the following scripts in the head tag of the page or at the bottom of the body tag - depending on your script loading strategy.

```
<!-- You can skip inclusion of the following two scripts if your web app already uses
 jquery version higher than 1.9. If not, make sure jquery is loaded before loading the
 preview script -->
<script type="text/javascript" src="http://@@WORKBENCH_HOST@@:@@WORKBENCH_PORT/ifcr/
static/oraclejet/js/libs/jquery/jquery-2.1.1.min.js"></script>
<script type="text/javascript">
 ocJQuery = jQuery.noConflict(true);
</script>

<!-- Oracle commerce preview framework that enables hotspots in a web page -->
<script type="text/javascript" src="http://@@WORKBENCH_HOST@@:@@WORKBENCH_PORT/ifcr/
tools/xmgr/app/preview/js/preview.js"></script>
```

RequireJS (AMD-based) Web page

```
<script type="text/javascript">
 requirejs.config({
   paths: {
     jquery: "/ifcr/static/oraclejet/js/libs/jquery/jquery-2.1.1.min",
     xmgrPreview: "/ifcr/tools/xmgr/app/preview/js/preview"
   }
 });
</script>
```

When you develop your application, you must pass the assembled content item (tree) returned by the Assembler to the preview framework to support editing and auditing a Web page from within Experience Manager. The content-item is used by Experience Manager to construct the manifest panel. Optionally, preview framework can add hotspots to slots and cartridges in the page by looking at the **data-oc-content-item-id** attribute on all DOM elements in the page.

You initialize the preview framework by calling the `initialize()` method and passing it the assembled content item returned by the Assembler. You can instead pass the content-uri that was used to invoke the

Assembler service. In this case, the framework makes a `jsonp` call to retrieve the content item. Be sure that the Assembler Service URL is set in Preview Settings in Workbench.

The framework provides an *eventing* mechanism to let application developers hook into its life cycle. The available events are `hotspotsOn`, `hotspotsOff`, `addContentItem`, and `removeContentItem`. For example, in order to be notified when the framework decorates the content items with hotspots, use:

```
framework.on("hotspotsOn", function() {
      // your code
    }, this);
```

The following preview framework API lists all the public methods and events. The API enables Assembler application pages to be decorated with hotspots when a business user previews pages in Experience Manager. The API must be used to enable Experience Manager to construct the manifest panel. The manifest panel assists the user in understanding how the page was assembled and allows the user to audit or edit slots and cartridges.

addContentItemId(pElement, pContentItem)

Adds **data-oc-content-item-id** attribute to the DOM element. All elements that should be decorated with hot spots must call this method.

Table 3.1. addContentItemId Parameters

Name	Type	Description
pElement	Object	Required. The DOM element that is injected with **data-oc-content-item-id** attribute. The element maps to the content item: slot or cartridge.

Name	Type	Description
pContentItem	Object	Required. The content item that represents a slot or cartridge. Example:


```
{
@type: "SearchBox",
endeca:auditInfo: {
"ecr:resourcePath": "/content/Web/Categories/
Pages/Category - Bags - Cases",
"ecr:assertedFacts": {
             "endeca:ancestorDimValId:100972",
             "endeca:ancestorDimValId:101022",
             "endeca:ancestorDimValId:101024",
             "101927",
             "endeca:ancestorDimValId:20001",
             "4294967266",
             "4294967247"
       ],
       "endeca:navAndSearchCount":  [
          "3"
       ],
       "endeca:time":  [
          "1441745067199"
       ],
       "endeca:workspaceId":  [
          "d826ca36-80a2-4278-a09f-2933dab92d1f"
       ],
       "endeca:path":  [
          "/content/Web"
       ],
       "endeca:templateType":  [
          "Page"
       ],
       "endeca:templateId":  [
          ""
       ],
       "endeca:workflowState":  [
          "1"
       ]
},
ecr:innerPath: "headerContent[0]"
},
name: "Search Box",
contentPaths: ["/content/Shared/Auto-Suggest
 Panels"],
templateTypes: ["AutoSuggestPanel"],
templateIds: [ ],
minAutoSuggestInputLength: "3",
ruleLimit: "1"
}
```

initialize(pContentItem, pCallback)

Initializes the preview framework, processes the content item tree and opens up a communication channel with Experience Manager. If Experience Manager responds with a load hot spots message, the framework traverses the application DOM and decorates the page with hot spots. This method should be invoked at the beginning of the page construction. Single page applications should use the **addContentItemId()** method to add **data-oc-content-item-id** attribute to the DOM elements that should be decorated with hotspots and then call **addHotspots()** to decorate the content items with hot spots.

Table 3.2. initialize Parameters

```
        ],
        "endeca:workflowState":  [
           "1"
        ]
},
},
name: "About Page",
headerContent: [
{
@type: "SearchBox",
endeca:auditInfo: {
"ecr:resourcePath": "/pages/DiscoverElectronics/
about us",
"ecr:assertedFacts": {
             "endeca:ancestorDimValId:100972",
             "endeca:ancestorDimValId:101022",
             "endeca:ancestorDimValId:101024",
             "101927",
             "endeca:ancestorDimValId:20001",
             "4294967266",
             "4294967247"
        ],
        "endeca:navAndSearchCount":  [
           "3"
        ],
        "endeca:time":  [
           "1441745067199"
        ],
        "endeca:workspaceId":  [
           "d826ca36-80a2-4278-a09f-2933dab92d1f"
        ],
        "endeca:path":  [
           "/content/Web"
        ],
        "endeca:templateType":  [
           "Page"
        ],
        "endeca:templateId":  [
           ""
        ],
        "endeca:workflowState":  [
           "1"
        ]
}
ecr:innerPath: "headerContent[0]"
},
name: "Search Box",
contentPaths: [
"/content/Shared/Auto-Suggest Panels"
],
templateTypes: [
"AutoSuggestPanel"
],
templateIds: [ ]
}
],
endeca:siteState: {
@class:
 "com.endeca.infront.site.model.SiteState",
contentPath: "/about-us",
siteId: "/DiscoverElectronics",
siteDisplayName: "Discover Electronics"
}

}
```

Name	Type	Description
pCallback	Function	Optional. The callback that is invoked after the framework is initialized. Always use the callback if the content uri is passed as the first argument. This ensures that the framework has been initialized.

addContentItem(pContentItem, pElement, pParentElement)

Add the given content item tree to the Manifest. Optionally turns on hot spots for the dynamically generated DOM element and its descendants - elements with the **data-oc-content-item-id** attribute which visually represent the content item tree. Information about the parent content item is needed in order to add it to the correct location in the page components area. Page components are uniquely identified by their **resourcePath** and **innerPath** information. This information can be found in the parent DOM element's **data-oc-content-item-id** attribute. If there is no parent DOM element, the content item will be added to the end of the page components tree in the Manifest.

An example of this method can be found in the `SearchBox.jsp` page in the Discover Electronics application. The parent search box DOM element is passed so that the auto suggest results cartridge is added to the correct location in the page components tree.

This method should be invoked to add dynamic content item trees, such as auto suggest, that are returned by the Assembler after a page has been constructed.

Table 3.3. addContentItem Parameters

Name	Type	Description
pContentItem	Object	Required. The new content item tree returned by the Assembler's JSON serializer.
pElement	Object	Optional. The new DOM element with data-oc-content-item-id attribute.
pParentElement	Object	Optional. The parent DOM element with **data-oc-content-item-id** attribute. Adds the content tree to the correct location in the page components area of the Manifest.

removeContentItem(pElement)

Removes hot spots from the element and its descendants. The associated content item tree is removed from the page components area of the Manifest.

Table 3.4. removeContentItem Parameters

Name	Type	Description
pElement	Object	Required. The DOM element with the {@link CONTENT_ITEM_SELECTOR} attribute whose hot spots and associated content item tree are to be removed.

addHotspots(pElement, pTraverseDom)

Decorates the element and optionally its descendants with hot spots. An error occurs if the preview framework has not been prepared yet. You must call the framework's **prepare()** method before using this method.

Table 3.5. Parameters

Name	Type	Description
pElement	Object	Required. The DOM element that is to be decorated with hot spots.
pTraverseDom	Boolean	Required. Flag indicating whether or not the descendants of the element have to be decorated.

removeHotspots(pElement, pTraverseDom)

Removes hot spots from the element and optionally removes its descendants. An error occurs if the preview framework has not been prepared yet.

Table 3.6. removeHotspots Parameters

Name	Type	Descripton
pElement	Object	Required. The DOM element from which hot spots are to be removed.
pTraverseDom	Boolean	Required. Flag indicating whether or not hot spots have to be removed from the descendants.

on(pEvent, pListener, pScope)

Registers a listener to one of the framework life cycle events. Supported events are:

- hotspotsOn
- hotspotsOff
- addContentItem
- removeContentItem

Table 3.7. on Parameters

Name	Type	Description
pEvent	String	Required. The name of the event.
pListener	Function	Required. The listener function that is invoked when the event fires.
pScope	Object	The scope that invokes the listener.

off(pEvent, pListener)

Removes a listener to one of the framework life cycle events.

Table 3.8. off Parameters

Name	Type	Description
pEvent	String	Required. The name of the event.
pListener	Function	Required. The listener function to be removed.

Enabling your preview application

After you have finished instrumenting your preview application, you can enable it for use in Workbench.

Ensure that your application has been correctly instrumented before enabling it for preview in Workbench. See Enabling your Java application for preview (page 71).

All examples shown below are taken from the configuration files for the Discover Electronics authoring application, located in `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring` (on Windows) or `$ENDECA_TOOLS_ROOT/reference/discover-electronics-authoring` (on UNIX). The exact mechanisms used for configuring your Assembler and content sources will depend on your implementation details.

For a full description of the properties described below, see the Assembler API Javadoc for the `AssemblerFactory` and `ContentSource` interfaces and their corresponding implementations.

To enable your custom preview application:

1. In the constructor arguments for your `AssemblerSettings`, set the following:

Property	Value
`previewEnabled`	`true`
`previewModuleUrl`	`http://localhost:8006/ifcr`

In the Discover Electronics reference implementation, these are configured as properties in `WEB-INF\assembler.properties`:

```
workbench.host=localhost
workbench.port=8006

# ... Additional settings removed from this example ...

preview.enabled=true
```

These properties are then included in the Assembler context file, `WEB-INF\assembler-context.xml`:

```
<!--
    ###########################################################################
    # ASSEMBLER FACTORY
    #
```

```
    # Required.
    #
-->
<bean id="assemblerFactory"
class="com.endeca.infront.assembler.spring.SpringAssemblerFactory"
scope="singleton">
    <constructor-arg>
        <bean class="com.endeca.infront.assembler.AssemblerSettings">
            <property name="previewEnabled" value="${preview.enabled}" />
            <property name="previewModuleUrl" value="http://${workbench.host}:
${workbench.port}/ifcr" />
        </bean>
    </constructor-arg>
    <constructor-arg>
        <list>
            <bean class="com.endeca.infront.logger.SLF4JAssemblerEventLogger" />
        </list>
    </constructor-arg>
</bean>
```

2. In the constructor arguments for your `StoreFactory` bean, set `isAuthoring` to `true`.

```
<bean id="StoreFactory" class="com.endeca.infront.content.source.FileStoreFactory"
    init-method="init" destroy-method="destroy">
    <property name="configurationPath" value="${repository.configuration.path}"/>
    <property name="isAuthoring" value="${preview.enabled}"/>
    <property name="appName" value="${workbench.app.name}" />
    <property name="host" value="${workbench.host}" />
    <property name="clientPort" value="${workbench.publishing.clientPort}"/>
    <property name="serverPort" value="${workbench.publishing.serverPort}"/>
    <property name="messageTimeout" value="10000" />
</bean>
```

Property name	Value
configurationPath	Required. The path from which the content and configuration zip files are retrieved
appName	Required. The name of the EAC application
isAuthoring	Optional. Default is false. If this is an authoring server, the value must be set to true.
host	Required if the `isAuthoring` property is set to true. The name of the server the workbench is running on. Default is localhost..
serverPort	Optional. Used only when the `isAuthoring` property is set to true. The port the Workbench is listening on. Default is 8007.

Property name	Value
clientPort	Optional. Used only when the `isAuthoring` property is set to true. The port used to initiate contact to the workbench from this server. If client port is set to -1, the system will assign an ephemeral port automatically.
messageTimeout	Optional. Used only when the `isAuthoring` property is set to true. The amount of time in milliseconds to wait on communications with the workbench. Default is 10000ms.

3. Configure a link service for your application that returns a preview link as a JSONP response.

   This service must construct a link to the page selected for preview; for example, if a content administrator previews the Brand - Canon Web Browse page in the reference application, the service returns `"/browse/_/N-25y6"`. Additionally, the response from the service is used to construct the links in the Audit Panel.

   In Discover Electronics, the link service is configured as a link servlet that uses the `com.endeca.infront.web.spring.PreviewLinkServlet` class. The servlet is defined in `WEB-INF\web.xml`:

```
<servlet>
    <servlet-name>link</servlet-name>
    <servlet-class>
        com.endeca.infront.assembler.servlet.spring.SpringPreviewLinkServlet
    </servlet-class>
    <init-param>
        <description>
            The ID of the NavigationStateBuilder in the spring
            contextConfig file
        </description>
        <param-name>navigationStateBuilderBeanId</param-name>
        <param-value>navigationStateBuilder</param-value>
    </init-param>
    <init-param>
        <description>
            The ID of the MdexResource in the spring
            contextConfig file
        </description>
        <param-name>mdexResourceBeanId</param-name>
        <param-value>mdexResource</param-value>
    </init-param>
</servlet>
```

Changing the preview link service

If you have implemented your own link service for use with preview, you can specify the path to the service.

After you have created your own preview link service, you can specify it for use with preview instead of the default link service included with the Discover Electronics reference application.

Note

For information about the required inputs and outputs for a link service, see the Javadoc for the `AbstractPreviewLinkServlet` class in the `com.endeca.infront.assembler.servlet` package.

To change the preview link service:

1. Stop the Tools Service.
2. Open your application's deployment descriptor file, `web.xml`.

   For the Discover Electronics reference application, this file is located at `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring\WEB-INF\web.xml`.
3. Define the link servlet.

   The servlet definition for the Discover Electronics reference application is shown below:

```
<servlet>
    <servlet-name>link</servlet-name>
    <servlet-class>
        com.endeca.infront.assembler.servlet.spring.SpringPreviewLinkServlet
    </servlet-class>
    <init-param>
        <description>
            The ID of the NavigationStateBuilder in the spring
            contextConfig file
        </description>
        <param-name>navigationStateBuilderBeanId</param-name>
        <param-value>navigationStateBuilder</param-value>
    </init-param>
    <init-param>
        <description>
            The ID of the ContentSource in the spring
            contextConfig file
        </description>
        <param-name>contentSourceBeanId</param-name>
        <param-value>contentSource</param-value>
    </init-param>
</servlet>
```

4. Define the link servlet mapping.

   For example:

```
<servlet-mapping>
    <servlet-name>link</servlet-name>
    <url-pattern>/servlet/link.json/*</url-pattern>
</servlet-mapping>
```

5. Save and close the file.
6. Start the Tools Service.

Managing the preview application in Workbench

You can manage the preview URL configuration and the preview devices for an application in Workbench by using the Preview Settings tool.

After you have instrumented your application and it for preview, it becomes the default preview application for your initial site or any sites that you add to your application. You can manage preview devices for displaying content and change the default application preview URLs or site-specific preview URLs.

1. Navigate to the Application Settings → Preview Settings tool.

2. Navigate to the Preview URLs section.

   The default preview URL that you set up in Enabling your preview application (page 80) appears in the Default Preview URL field. The link service URL that you set up in Changing the preview link service (page 82) appears in the Default Link Service URL field. The URL that exposes the Assembler Services REST API for non-Java applications appears in the Default Assembler Service URL field. These URLs also appear by default for the sites listed in the section below the default URLs section.

3. To change the default preview application follow these steps:

   a. Enter the fully qualified preview URL of the default preview application in the Default Preview URL field.

   b. Enter the URL of the service within the application that constructs links for preview in the Default Link Service URL.

   c. Enter the URL that exposes the Assembler Services REST API for preview in the Default Assembler Service URL. This field is only required for non-Java applications. It is not required for Java applications. See Formatting the Assembler Service URL (page 85) for more information.

4. To update the preview URLs and link service URLs for a site follow these steps:

   a. In the Preview URL field of the site, enter the fully qualified preview URL of the site that you want to preview .

   b. In the Link Service URL of the site, enter the URL of the service within the site that constructs links for preview.

      If your organization has integrated Oracle Core Commerce Platform with Guided Search, you must include the `pushSite` parameter in the Link Service URL for each site: `pushSite=<siteID>`. For example:

      ```
      http://<PREVIEW_HOST>:<PREVIEW_PORT>/crs/link.json?pushSite=mobileHomeSite
      ```

   c. In the Assembler Service URL of the site, URL that exposes the Assembler Services REST API for preview. This field is only required for non-Java applications. It is not required for Java applications.

5. Click Save.

6. In the Manage Preview Devices section, enter values for the attributes of each preview device.

   All devices can be rotated, so enter the height and width of the page orientation that is previewed more frequently.

Option	Description
Name	The name of the device
User Agent	Agent types supported as a string
Height	The view port height of the device, in pixels

Option	Description
Width	The view port width of the device, in pixels
Zoom	The Zoom factor can simulate displays on devices other than the current monitor. For example, the display on a retina display monitor can be simulated by setting the Zoom factor to 30 -- that is, 30%.

The following table shows example preview settings by device.

Option	First Preview Device	Second Preview Device
Name	Handheld	Tablet
User Agent	Mozilla/5.0 (iPhone; U; CPU like Mac OS X; en) AppleWebKit/420+ (KHTML, like Gecko) Version/3.0 Mobile/1A537a Safari/419.3	Mozilla/5.0 (iPad; U; CPU OS 3_2 like Mac OS X; en-us) AppleWebKit/531.21.10 (KHTML, like Gecko) Version/4.0.4 Mobile/7B334b Safari/531.21.10
Height	680	1224
Width	400	848
Zoom	30	60

7. Click Save.

Formatting the Assembler Service URL

At runtime, the Assembler Service URL is used to build the various URLs for individual content items in your site.

For example, if your site uses the following Assembler Service URL:

```
http://localhost:8006/assembler-authoring/json
```

The JSON for the `/browse` page can be retrieved by concatenating the path at the end of the URL:

```
http://localhost:8006/assembler-authoring/json/browse
```

If the path is not appended at the end of the URL, you can also use a placeholder `%s` to designate where the path should be inserted at runtime. For example, to retrieve the following `/browse` page:

```
http://localhost:8006/discover-authoring/browse?format=json
```

You can format the Assembler Service URL with a placeholder:

```
http://localhost:8006/discover-authoring%s?format=json
```

Note that if you do not use a placeholder, the path is always appended to the end of the Assembler Service URL.

Experience Manager makes a JSONP call to the deployed Assembler Service by using the *jsonp* query parameter. For example, to construct the Manifest panel for the browse page, Experience Manager makes a call to the Assembler Service at the following:

```
http://localhost:8006/assembler-authoring/json/browse?jsonp=<dynamic_method_name>
```

Testing your preview application

After instrumenting and enabling your preview application, you can test the preview and audit functionality in Workbench.

Your custom preview application must be fully instrumented and enabled in Workbench in order for the preview option to be displayed.

To test your custom preview application:

1. In Workbench, navigate to the Experience Manager tool.
2. Navigate to a content item of your choosing.
3. Hove the mouse over the content item.

   The Action menu button appears.
4. Select Preview from the Action dropdown.
5. Optionally, specify the preview time instead of using the default indicated by the system clock for the MDEX Engine:

   a. Click the arrow beside the selected device in the Preview Toolbar:

      The Preview Toolbar expands to show configuration options.

   b. Select a device from the Device list and click Apply.

   Specifying a preview device lets you see how the application renders on that device.
6. To test audit functionality:

   a. Hove the mouse over the cartridge you wish to audit.

   b. Click the gear button and select Audit

      The Audit Panel appears with a list of all content items considered for the specified content slot.

   c. Click any of the listed Locations to navigate to that location in the preview application.

   d. Click the name of any of the listed content items and confirm that you return to that item in Experience Manager.

Disabling preview

You can disable the ability to preview your application in Experience Manager.

To disable preview, you export the preview configuration of your application, replace the preview URL and link service URL with null values and then import your updated preview configuration, as follows:

1. Export the preview configuration.

   The following command exports the Discover preview configuration to the `/import` folder in unzipped files:

```
runcommand.sh IFCR exportContent configuration/tools/preview /localdisk/apps/
Discover/config/import/configuration/tools/preview true
```

2. Replace the preview URL and link services URL information with null values in the configuration-preview JSON file.

```
{
 "ecr:type":"configuration-preview",
 "linkServiceUrl":"",
 "previewUrl":"",
 "assemblerServiceURL":"http://localhost:8006//assembler-authoring/json",
 "devices": [
   {
    "userAgent":"Mozilla/5.0 (iPhone; U; CPU like Mac OS X; en) AppleWebKit/420+
    (KHTML, like Gecko)
      Version/3.0 Mobile/1A537a Safari/419.3",
    "name":"Handheld (Portrait)",
    "height":1280,
    "width":960,
    "zoom":50
    }
 ]
}
```

3. Import the updated preview configuration.

   The following command imports the Discover preview configuration to the `configuration/tools/preview` folder:

```
runcommand.sh IFCR importContent configuration/tools/preview /localdisk/apps/
Discover/config/import/configuration/tools/preview
```

Configuring logging for an Assembler Application

The Assembler logs information to the Platform Services Log Server component.

In order to implement this logging feature in an application, you must instantiate a `LogServerAdapter` and pass it in to the `AssemblerFactory`, along with any other Event Listeners. To log front-end information, you must also register a `ContentItemAugmentAdapter`. The `LogServerAdapter` and `ContentItemAugmentAdapter` require a `RequestEventInitializer` and a `MdexQueryInfoInitializer` to log request event information to the Dgraph request log, so these should also be configured. Registering a `MdexQueryInfoInitializer` lets you establish the relationship between the Assembler request and its corresponding MDEX Engine queries in the Dgraph request logs. This helps you to identify and troubleshoot problems.

Configuring the RequestEventInitializer and the MdexQueryInfoInitializer

The `RequestEventIntiailizer` is used to initialize the RequestEvent thread local variable. Similarly, `MdexQueryInfoInitializer` is used to initialize the MdexQueryInfo thread local variable in the assembly process. If the `MdexQueryInfoInitializer` is not included in the assembler configuration, then the additional request information will not be included in the Dgraph request log.

In the following example of an `assembler-context.xml` file in a Spring implementation, note that the `RequestEventIntiailizer` is configured before the `LogServerAdapter` and `ContentItemAugmentAdapter` so that the request event is logged. The `MdexQueryInfoInitializer` is also configured so that query information such as the request id, and the session id is added to the Dgraph request log.

```
                <bean
 class="com.endeca.infront.assembler.event.request.RequestEventInitializer">
    <property name="sessionIdProvider" ref="springUtility"/>
    <property name="requestIdProvider" ref="springUtility"/>
</bean>
<bean class="com.endeca.infront.navigation.event.MdexQueryInfoInitializer">
</bean>
<bean class="com.endeca.infront.assembler.event.request.ContentItemAugmentAdapter">
</bean>
<!-- Remove the following lines to disable logging to an Oracle Endeca Log Server -->
<bean class="com.endeca.infront.navigation.event.LogServerAdapter">
    <property name="logServerHost" value="${logserver.host}"/>
    <property name="logServerPort" value="${logserver.port}"/>
    <property name="isSslEnabled" value="${logserver.sslEnabled}"/>
</bean>
```

The `RequestEventInitializer` specifies a `SessionIdProvider` and a `RequestIdProvider` so you can associate Assembler requests with MDEX query entries in the Dgraph request log. You specify the following properties:

- An instance of an object that implements the `com.endeca.infront.assembler.event.request.SessionIdProvider` interface , which requires a `String getSessionId()` method that returns a user's session ID.
- An instance of an object that implements the `com.endeca.infront.assembler.event.request.RequestIdProvider` interface , which requires a `String getRequestId()` method that returns a request ID.

The referenced bean is configured as follows:

```
<bean id="springUtility" class="com.endeca.infront.web.spring.SpringUtility"
 scope="singleton"/>
```

Instantiating a ContentItemAugmentAdapter

The `ContentItemAugmentAdapter` augments content items with front-end application information, such as a user's search and navigation state. As a result, the response content item returned from an `assemble()` call includes cartridge logging information.

```
<bean class="com.endeca.infront.assembler.event.request.ContentItemAugmentAdapter">
```

```
</bean>
```

Instantiating a LogServerAdapter

The `LogServerAdapter` logs server-side information. You specify the following properties:

- Log server host
- Log server port
- SSL Enabled (optional)

In the Spring implementation, it is configured in the `assembler-context.xml` file as follows:

```
<bean class="com.endeca.infront.navigation.event.LogServerAdapter">
    <property name="logServerHost" value="${logserver.host}"/>
    <property name="logServerPort" value="${logserver.port}"/>
    <property name="isSslEnabled" value="${logserver.sslEnabled}"/>
</bean>
```

Dgraph request logs

The following Dgraph request log shows the request component (reqcom), session ID (sid) , and request ID (rid) information:

```
1402522721112 127.0.0.1 - 3 8357 3.13 2.82 200 5684 0 1 /graph?
node=0&offset=0&nbins=0&log=reqcom%3dcontentRequestBroker%26sid
%3dDD2CAF659BE6CFDFFDED091F69037113%26rid%3d1402522720981640 91&irversion=640 - Accept
%3A+%2A%2F%2A%0D%0APragma%3A+no%2Dcache%0D%0ACache%2DControl%3A+no%2Dcache%0D%0AUser
%2DAgent%3A+Java%2F1%2E7%2E0%5F25%0D%0AHost%3A+localhost%3A15000%0D%0AConnection%3A
+keep%2Dalive%0D%0A
```

The request component is the class that made the MDEX Engine query.

Customizing logging information

All key-value pairs in `com.endeca.infront.assembler.event.request.MdexQueryInfo` are logged to the Dgraph response log. You can extend `MdexQueryInfoInitializer` and add the desired key-value pairs. For example:

```
public class CustomMdexQueryInfoInitializer extends MdexQueryInfoInitializer {
    public CustomMdexQueryInfoInitializer() {
        super();
    }
    public void assemblyStarting(AssemblerEvent event) {
        super.assemblyStarting(event);
        String customKey = "CUSTOM_KEY";
        String customValue = "CUSTOM_VALUE";
        //Get the MdexQueryInfo associated with the current assembler request
        MdexQueryInfo info = MdexQueryInfoFactory.getMdexQueryInfo();
        //Store the custom information
        info.put(customKey ,customValue);
    }
```

```
}
```

Customizing session ID information

Depending on the information you wish to include in a session ID object, you can create a custom implementation of the `SessionIdProvider` interface. For additional information, refer to the *Assembler API Reference (Javadoc)*.

Customizing request ID information

Depending on the information you wish to include in a request ID object, you can create a custom implementation of the `RequestIdProvider` interface. For additional information, refer to the *Assembler API Reference (Javadoc)*.

Configuring the Log4J logger

The logging implementation in the Discover Electronics reference application uses the Log4J logger. Log level settings are configurable through the properties file located at `ToolsAndFrameworks\<version>\reference\discover-electronics-[authoring|live]\WEB_INF\classes\log4j.properties`. If you choose to use this implementation in your own application, you can configure the log level by opening the corresponding file.

Locate and uncomment the following line:

```
# Uncomment to see Oracle Commerce Assembler debug info.
# log4j.logger.com.endeca.infront.logger=DEBUG
```

At the `DEBUG` level, Assembler and cartridge handler entrances and exits are logged, although the details of the navigation context passed in to the cartridge handler do not appear.

Configuring logging for custom events

You can create custom cartridge handlers to collect and act on any information that is important to your application.

About request events

Each invocation of the Assembler creates an associated `RequestEvent` object that tracks request information.

Information on a `RequestEvent` is stored as key/value pairs. You can include arbitrary information about an Assembler request by extending the `RequestEvent` object in a cartridge handler's `process` method. For example:

```
/**
  * Cartridge Handler process method
  */
public void process(ConfigType pContentType) {

  // Create a new RequestEvent from the global RequestEvent object
  RequestEvent event = RequestEventFactory.getEvent();

  // Store arbitrary information
```

```
  event.put("myKey","my arbitrary value");

  ...
}
```

The `NavigationEventWrapper` class

The `NavigationEventWrapper` class provides convenience methods for getting and setting common search and navigation information about a request event. It modifies the `RequestEvent` object specified in the constructor, as in the example below:

```
/**
  * Cartridge Handler process method
  */
public void process(ConfigType pContentType) {

  // Create a new NavigationEventWrapper around the global RequestEvent object
  NavigationEventWrapper navigationEvent = new
 NavigationEventWrapper(RequestEventFactory.getEvent());

  // Store navigation event information
  navigationEvent.setAutocorrectTo("autocorrected term");

  ...
}
```

For additional information about the `RequestEvent` and `NavigationEventWrapper` classes, including a full list of the convenience methods available for the `NavigationEventWrapper`, see the *Assembler API Reference (Javadoc)*.

About request event adapters

Request event adapter classes perform some action based on information included with a request event.

A request event adapter class implements the `handleAssemblerRequest()` method in the abstract `RequestEventListener` class. This method is invoked at the end of the Assembler's `assemble()` method.

The following is an example of a simple request event adapter:

```
/**
 * Add log information to root content item
 */
public class SampleRequestEventAdapter extends RequestEventListener {

    /**
     * Constructor
     * @param sessionIdProvider provides an ID for the current user session
     */
    public SampleRequestEventAdapter() {
        super();
    }

    /**
     * Prints the request event's session id, request id, and search term (if present)
 to the console
```

```
     * @param assemblerRequestEvent the event containing all of the
     * information about the Assembler request
     * @param rootContentItem the Assembler output
     */
    public void handleAssemblerRequestEvent(RequestEvent event, ContentItem
 rootContentItem) {
        NavigationEventWrapper navigationEvent = new
 NavigationEventWrapper(assemblerRequestEvent);
        // Print Session ID - Note that the session Id has already been determined and
 set in the event object
        System.out.println("The current session is: "+event.getSessionId());
        // Print Request ID - Note that the request Id has already been determined and
 set in the event object
        System.out.println("The request ID is: "+event.getRequestId());
        // Print Search Term
        if (navigationEvent.getSearchTerms() != null && !
navigationEvent.getSearchTerms().trim().isEmpty()) {
            System.out.println("The current search terms are:
 "+navigationEvent.getSearchTerms());
        } else {
            System.out.println("There were no search terms in the current request");
        }
    }
}
```

The SessionIdProvider interface

The example request event adapter registers an implementation of `SessionIdProvider` in the constructor. This enables it to retrieve the server session ID.

The Tools and Frameworks installation implements this interface in the included `SpringUtility` class. You can create your own `SessionIdProvider` class by extending the `SessionIdProvider` interface and implementing the `getSessionID()` method.

The RequestIdProvider interface

The example request event adapter registers an implementation of `RequestIdProvider` in the constructor. This enables it to retrieve the request ID.

The Tools and Frameworks installation implements this interface in the included `SpringUtility` class. You can create your own `RequestIdProvider` class by extending the `RquestIdProvider` interface and implementing the `getRequestID()` method.

Request event adapters in the reference application

The Discover Electronics reference application includes the following implementations of the `AssemblerEventListener` interface:

- AssemblerEventAdapter
- ContentItemAugmentAdapter
- LogServerAdapter
- RequestEventListener

For additional information about these classes, see the *Assembler API Reference (Javadoc)*.

About registering a request event adapter

To use a request event adapter, you must register it with your `AssemblerFactory`.

You can disable request event adapters by removing them from the `AssemblerFactory` configuration.

Request event adapter configuration in the reference application

In the reference application, the `AssemblerFactory` interface is implemented as `SpringAssemblerFactory`, and the `AssemblerEventListener` objects are specified as constructor arguments in the Assembler context file:

```
<!--
  ################################################################################
  # ASSEMBLER FACTORY
  #
  # Required.
  #
-->
<bean id="assemblerFactory"
 class="com.endeca.infront.assembler.spring.SpringAssemblerFactory"
      scope="singleton">
    <constructor-arg>
        <bean class="com.endeca.infront.assembler.AssemblerSettings">
            <property name="previewEnabled" value="${preview.enabled}" />
            <property name="previewModuleUrl" value="http://${workbench.host}:
${workbench.port}/ifcr" />
        </bean>
    </constructor-arg>
    <constructor-arg>
        <list>
            <bean class="com.endeca.infront.logger.SLF4JAssemblerEventLogger" />
            <bean
 class="com.endeca.infront.assembler.event.request.RequestEventInitializer">
                    <property name="sessionIdProvider" ref="springUtility"/>
                    <property name="requestIdProvider" ref="springUtility"/>
             </bean>
             <bean
 class="com.endeca.infront.navigation.event.MdexQueryInfoInitializer">
             </bean>
             <bean
 class="com.endeca.infront.assembler.event.request.ContentItemAugmentAdapter">
             </bean>
             <!-- Remove the following lines to disable logging to an Oracle Endeca Log
 Server -->
             <bean class="com.endeca.infront.navigation.event.LogServerAdapter">
                 <property name="logServerHost" value="${logserver.host}"/>
                 <property name="logServerPort" value="${logserver.port}"/>
                 <property name="isSslEnabled" value="${logserver.sslEnabled}"/>
             </bean>
        </list>
    </constructor-arg>
</bean>
```

Request event adapters in the reference application

The Discover Electronics reference application includes two request event adapters, `ContentItemAugmentAdapter` and `LogServerAdapater`.

Adapter	Description
`com.endeca.infront.assembler.request.ContentItemRequestEventAdapter`	Appends request event information to the Content Item returned by the `assemble()` method. Information is included as a nested Content Item of type `AssemblerRequestEvent`, with the key `endeca:assemblerRequestInformation`. For a list of attributes that are available out-of-the-box, see Request Event Attributes (page 327).
`com.endeca.infront.navigation.event.LogServerAdapter`	Formats data from the request event and sends it to the Log Server, which enables Workbench users to generate reports using the Report Generator. The adapter must be configured with the host and port of the log server. In the reference application, these values are configured in the `WEB-INF\assembler.properties` file.

Client side click events

The Oracle log server tracks the following click events from the client side of an Assembler application:

Attribute Key	Type	Description
IN_DIM_SEARCH	`Boolean`	Did the user select a dimension search result.
IN_DYM	`Boolean`	Did the user select the "did-you-mean" value.
IN_MERCH	`Boolean`	Did the user select a merch rule (spotlight).
CONVERTED	`Boolean`	Did an action cause a conversion.

You can include the information collected from these events in your application reports. For more information about the Log Server and Report Generator components, refer to the *Platform Services Log Server and Report Generator Guide*.

Configuring cartridge performance logging

The Assembler tracks performance statistics for registered events; this information is available from the administrative servlet at `http://<workbench host>:<workbench port>/<application>/admin` using the `/admin?op=stats` operation.

For example, you can view the performance statistics for the default Discover Electronics application by navigating to `http://localhost:8006/discover-authoring/admin?op=stats`. For more information about the administrative servlet, see the *Oracle Commerce Administrator's Guide*.

Performance logging is enabled for the core cartridges included with Tools and Frameworks. If you create a custom cartridge handler and wish to track its processing time, you must use the static `PerfUtil.start()` method to create a corresponding `Event`.

Example 3.4. Example

For example:

```
Event event = PerfUtil.start("com.example.ClassName_MyMethod");
try {
    /* cartridge handler logic */
    event.succeed();
} finally {
    event.failIfNotCompleted();
}
```

Note

A call to `PerfUtil.start` must include a corresponding call to either the `Event.succeed()` or `Event.fail()` method of the returned `Event` instance. Oracle recommends using the `Event.failIfNotCompleted()` helper method within a `finally{}` block to ensure proper resolution.

For more information about the `com.endeca.infront.perf` package, see the *Assembler API Reference (Javadoc)*.

Debugging MDEX Engine query results

The MDEX Engine provides several methods for understanding why certain results were returned for a query so that you can determine how to tune search features to provide the desired results.

Query debugging features

The MDEX Engine query debugging features include Why Match, Word Interpretation, Why Rank, and Why Precedence Rule Fired. Each feature provides information about a different aspect of search results.

Feature	Description
Why Match	Augments record results with information about which record properties were involved in search matching.

Feature	Description
Why Rank	Augments record results with information about which relevance ranking modules ordered the results and why a particular record was ranked in the way that it was.
Why Precedence Rule Fired	Augments root dimension values with information about how the precedence rule was triggered (explicitly or implicitly), which dimension ID and name triggered the precedence rule, and the type of precedence rule (standard, leaf, or default).
Word Interpretation	Reports word or phrase substitutions made during text search processing due to stemming, thesaurus expansion, or spelling correction.

Enabling query debugging features

You enable the query debugging features on an Assembler application via the `debugEnabled` constructor argument on your `MdexRequestBroker` object. In the Discover Electronics reference application, this is configured in the MDEX Resource section of the Spring context file for the Assembler.

When `debugEnabled` is set to `true`, it enables query debugging features to be applied to an Assembler request. When set to `false`, debugging features are turned off for every request. Debugging features are disabled by default.

In addition to the corresponding object configuration, Word Interpretation must be enabled via the `--wordInterp` Dgraph flag.

The following shows the default MDEX resource configuration in the Discover Electronics application:

```
<bean id="mdexRequestBuilder" scope="request"
 class="com.endeca.infront.navigation.request.MdexRequestBroker">
    <constructor-arg ref="mdexResource" />
    <!-- Debug Enabled Parameter. When set to true, allows debug information to be
 returned from the Assembler -->
    <constructor-arg value="false"/>
</bean>
```

URL parameters for query debugging features

All query debugging features except for Word Interpretation may be enabled on a per-query basis via URL parameters.

The following parameters take a value of `1` (for enabled) or `0` (for disabled):

- `whymatch`
- `whyrank`
- `whyprecedencerulefired`

The Word Interpretation feature can only be enabled at the level of an individual cartridge handler.

Note

If the debug constructor argument on the MDEX resource bean is set to `false`, all debugging features are disabled on every request regardless of URL parameters.

Query debugging results in the reference application

In Discover Electronics, the results of query debugging can be returned as part of the response model for the Results List, Search Adjustments, and Refinement Menu cartridges as appropriate. In the Discover Electronics reference application, these results can be enabled by removing comments from the corresponding properties in each cartridge handler.

The debugging results are returned as properties on returned records:

Feature	Results
Why Match	Returns information about why each record matched the query in a `Dgraph.WhyMatch` property on the record.
Why Rank	Returns information about why each record was ranked the way it was in a `Dgraph.WhyRank` property on the record.
Why Precedence Rule Fired	Returns information about precedence rules that fired on a query in a `DGraph.WhyPrecedenceRuleFired` property on each root dimension value.
Word Interpretation	Returns information about word or phrase substitutions as a map that can be accessed via `getInterpretedTerms()` on the `SearchAdjustments` model.

For details about the format of the debugging results, refer to the *MDEX Engine Developer's Guide*.

Note

The renderers in the Discover Electronics application do not include rendering logic to display the query debugging properties, but the information is available from the JSON or XML view.

The relevant configuration for the individual cartridge handlers in the Discover Electronics reference application is shown below:

- Results List — Why Match, Why Rank

```
<bean class="com.endeca.infront.cartridge.ResultsListConfig" scope="singleton">
    <!-- <property name="whyMatchEnabled" value="true"/> -->
    <!-- <property name="whyRankEnabled" value="true"/> -->
                          <!-- additional elements omitted from this example -->
</bean>
```

Enabling these settings overrides the default values specified for the `setWhyMatchEnabled` and `setWhyRankEnabled` methods on the `com.endeca.infront.cartridge.ResultsListConfig` object when the Tools Service is initialized.

- Refinement Menu — Why Precedence Rule Fired

```
<bean class="com.endeca.infront.cartridge.RefinementMenuConfig" scope="singleton">
    <property name="moreLinkText" value="More..."/>
    <!-- <property name="whyPrecedenceRuleFired" value="true"/> -->
</bean>
```

Enabling this setting overrides the default value specified for the `setWhyPrecedenceRuleFired` method on the `com.endeca.infront.cartridge.RefinementMenuConfig` object when the Tools Service is initialized.

- Search Adjustments — Word Interpretation

```
<bean class="com.endeca.infront.cartridge.SearchAdjustmentsConfig" scope="singleton">
    <!-- <property name="showWordInterp" value="true"/> -->
</bean>
```

Enabling this setting overrides the default value specified for the `setShowWordInterp` method on the `com.endeca.infront.cartridge.SearchAdjustmentsConfig` object when the Tools Service is initialized.

4 Optimizing Application URLs

This part provides information on optimizing application URLs.

About the URL optimization classes

This section provides an introduction to the URL optimization classes in the Assembler API.

Related links

- *Optimizing Application URLs* (page 99)
- Package contents (page 99)
- Introduction to URL optimization (page 99)
- Overview of URL optimization capabilities (page 100)
- URL canonicalization (page 101)

Package contents

The `com.endeca.soleng.urlformatter` package within `ToolsAndFrameworks\<version>\assembler\lib\endeca_assembler-<version>.jar` contains the classes and dependencies necessary for generating optimized URLs and canoncial links in your application.

To enable the API for the Discover Electronics reference application, the `endeca_assembler-<version>.jar` file is also included under the `ToolsAndFrameworks\<version>\reference\discover-electronics-authoring\WEB-INF\lib` directory.

Introduction to URL optimization

Dynamically created URLs that are composed of meaningless, randomly generated strings can lower your site's search engine ranking and make it harder for users to recognize your site. The Assembler API includes classes that enable you to create site links using directory-style URLs. These URLs include keywords and store the dynamic information in the base URL rather than in the query string.

The resulting URLs do not contain any URL query parameters. Instead, all of the necessary values are stored in the URL path, resulting in search engine-friendly URLs.

Note

The examples in this guide assume a sample Web application running on `http://localhost:8888` against a wine data set.

Overview of URL optimization capabilities

The URL optimization classes are designed to increase your search engine rankings by enabling you to create search engine-friendly URLs.

Integration of keywords into the URL string

Many search engines evaluate URL strings as part of their relevancy ranking strategy. Generating URLs that include keywords can increase your natural search engine ranking as well as create visitor-friendly URLs that are easier for front-end users to understand.

Using the URL optimization classes, you can configure the following strings to appear in the URL:

- Dimensions
- Dimension values
- Dimension ancestors
- Record properties
- Text search queries

For example, the base URL for a Merlot page in a wine application configured to include ancestors in the string could appear as:

```
http://localhost/ContentAssemblerRefApp/Content.aspx/Wine-Red-Merlot/
```

The optimized URL is more comprehensible to users and more search-engine friendly than the traditional URL, which contains no keywords:

```
http://localhost:8888/endeca_jspref/controller.jsp?
sid=122C7EA4C912&Ne=6200&enePort=15000&eneHost=localhost&N=8025
```

Canonicalizing the URL string

Dynamic sites often produce syntactically different URLs for the same page. Multiple variant URLs result in duplicate content and lower search engine ranking.

For example, users might be able to reach a Napa white wine page by first clicking on “Napa” and then clicking on “White”, or by first clicking on “White” and then “Napa.” This creates two syntactically unique links pointing to the same Napa White page:

- `http://localhost:8888/urlformatter_jspref/controller/Wine-White/Region-Germany/_/N-1z141vcZ66t`
- `http://localhost:8888/urlformatter_jspref/controller/Region-Germany/Wine-White/_/N-1z141vcZ66t`

To ensure that only one version of the URL per page is used in links throughout the site, the `com.endeca.soleng.urlformatter .NavStateCanonicalizer` interface provides options for creating a single "canonical" URL for a given location.

Configuring the word separator string

It is possible to customize the word separator for each keyword string in the URLs. By default, the word separator is the dash character "-":

```
http://localhost:8888/urlformatter_jspref/controller/Wine-White/Region-Germany/_/
N-1z141vcZ66t
```

Moving URL parameters out of the query string

In order to create directory-style URLs, you can limit the number of parameters in the query string by moving them from the query string and into the path-params section of the URL.

For example, the following URL has the parameters `N`, `Ntk`, `Ntt`, and `Ntx` in the query string:

```
http://localhost/ContentAssemblerRefApp/Content.aspx/Bordeaux?
N=4294966952&fromsearch=false&Ntk=All&Ntt=red&Ntx=mode%2bmatchallpartial
```

To optimize the URL, you can move parameters into the path-params section of the URL. For example, the following URL includes the `N` and `Ntt` parameters in the base URL:

```
http://localhost/ContentAssemblerRefApp/Content.aspx/Bordeaux/_/N-4294966952/Ntt-red?
fromsearch=false&Ntk=All&Ntx=mode%2bmatchallpartial
```

Encoding Parameters

In order to shorten URLs, the URL optimization classes allow base-36 encoding of parameters.

For example, the following URL for Vintage > 1996 contains the dimension value ID for 1996 (4294962059):

```
http://localhost/ContentAssemblerRefApp/Content.aspx/_/N-4294962059
```

By base-36 encoding the N parameter, you can shorten the URL:

```
http://localhost/ContentAssemblerRefApp/Content.aspx/_/N-1z13xxn
```

URL canonicalization

Dynamic sites often produce syntactically different URLs for the same page. Multiple variant URLs can lower the search engine ranking of a page. Canonicalizing URLs reduces the duplicate content and improves search engine ranking.

Many search engines base their relevancy ranking algorithms on the number and quality of links that point to a particular page. The more links there are that point to a particular page, the higher the page rank. Multiple URLs generated by a dynamic site can lower the ranking of a page because, to the search engine, each version of the URL appears to point to a different page.

For example, users might be able to reach a Napa Red wine page by first clicking on “Napa” and then clicking on “Red”, or by first clicking on “Red” and then “Napa.” This creates two syntactically unique links pointing to the same Napa Red page:

- `http://localhost:8888/urlformatter_jspref/controller/Wine-Red/Region-Napa/_/N-1z141vcZ66t`
- `http://localhost:8888/urlformatter_jspref/controller/Region-Napa/Wine-Red/_/N-1z141vcZ66t`

To the search engine, each version of the URL appears to be its own unique page with identical or near-identical content, and each page takes a portion of the link references.

To improve quality, search engines try to minimize the appearance of largely similar pages within results sets. Among other strategies, all indexed pages are evaluated for duplicates and near-duplicates before a page is selected to be displayed in the search results. In the case of the Napa Red page, only one of the two URLs would be selected -- and therefore only half of the link references are evaluated. This link dilution of the Napa Red page may result in a lower position within search results. Multiple parameters in URLs have the same effect.

In order to avoid multiple versions of URLs per page, links throughout the site should be standardized (canonicalized), and requests for a non-standard version of the URL should be redirected to the canonical version via a "301" (permanent) redirect.

By design, the URL optimization classes prevent the creation of syntactically different URLs by canonicalizing keywords, ensuring that equivalent pages have URLs with the same syntax even if they can be navigated to through different paths. You can choose from a number of configuration options to control the arrangement of keywords. For example, you can configure your `UrlFormatter` object to arrange dimensions alphabetically in an ascending order:

- `http://localhost:8888/urlformatter_jspref/controller/Region-Napa/Wine-Red/_/N-1z141vcZ66t`

Now even if a user navigates to "Red" before "Napa", the link still appears as `/Region-Napa/Wine-Red`.

Related links

- Canonicalization configuration options (page 133)

Working with Application URLs

Each of the user-facing pages in an Assembler application exists as a page with a corresponding navigation or record state; the combination of the page and its state results in a specific set of results or a set of record details. The Assembler API includes an `Action` class for storing these URL components and returning them as part of the output model produced by a cartridge handler.

Related links

- *Optimizing Application URLs* (page 99)
- About application URLs (page 102)
- About Actions (page 103)
- Working with URL parameters (page 107)
- URL configuration in the reference application (page 108)
- About working with canonical links (page 112)

About application URLs

Features in a front-end application can provide one or more links to other locations within a site. The information required for constructing these links is provided on the cartridge response model as an `Action` object that includes the components of a destination URL.

For example, a dimension refinement in a Refinement Menu cartridge has an associated action to select the refinement and add it to the end user's navigation state. A record in a Results List cartridge has an action to view the corresponding record detail page.

The Assembler API includes an `ActionPathProvider` interface that returns components of an application URL. For the Discover Electronics reference application, an implementation of this interface is configured in the `NavigationCartridgeHandler`.

Cartridge handlers in the reference application use this implementation to create `NavigationAction` paths to a certain navigation state (like the modified navigation state created when a user selects a dimension refinement), or `RecordAction` paths to specified records (such as a record select from the results list).

About Actions

An `Action` object allows you to construct a link that represents a decision by an end user. The included fields and values depend on the action that the user wishes to take; they can include the action label, the root site path, the path to the destination content within the site, and the site state.

The `Action` class does not include a complete URL to the resulting navigation state or record; instead, the URL resulting from an Action is generally created by combining fields. For details, see "Action fields."

The Assembler splits the class into three subclasses:

- `NavigationAction` — An Action that represents changing the current navigation state, such as through a search query or the addition of a dimension refinement. For example, the "See All" link on a `RecordSpotlight` object includes a `NavigationAction` for navigating to the refinement state represented by the spotlight.
- `RecordAction` — An Action that represents selecting a record or aggregate record. The individual records in a `RecordSpotlight` each include a `RecordAction` for selecting that record.
- `UrlAction` — An Action that represents following an arbitrary URL. The Media Banner cartridge includes a `UrlAction` for URLs that are manually specified in Experience Manager.

Note

For information about the Actions associated with each output model, refer to the *Assembler API Reference (Javadoc)* for the corresponding class.

Action fields

All Actions include the following fields:

Field	Description
Label	The label that displays to the application end-user for the specified action. For example, you might set this to a product name for a link from a results list to a record detail page, or it you might set it to a dimension refinement name when displaying a breadcrumb with an associated Action to remove the refinement and adjust the user's navigation state.
Site root path	The path that identifies the EAC application associated with the Action, such as `/sites/Discover`.

Field	Description
Content path	The path that identifies the content associated with the Action within the containing site. In the Discover Electronics reference application, this is the servlet that handles the specified content type, such as `/browse` or `/detail`.
Site state	Site State is an object that contains the `siteId`, `matchedUrlPattern`, and `contentPath` used to query the Assembler.

Additionally, certain types of Actions may include additional fields. A `NavigationAction` includes a field for the navigation state represented by the Action, while a `RecordAction` action includes a field for the corresponding record state.

Using action fields

To construct a useable link from an Action, the UI tier of your application (the cartridge renderers in the Discover Electronics reference application) must include logic for combining the Action fields. A typical use case consists of directly concatenating fields, depending on the type of page you wish to link to.

In the reference application, a link to a navigation state typically combines the content path and the desired navigation state:

```
String href = action.getContentPath() + action.getNavigationState();
```

A link to a record details page combines the content path with the appropriate record state:

```
String href = action.getContentPath() + action.getRecordState();
```

In an application with multiple sites where your site definition specifies URL pattern matching, a link to a navigation state combines the site state, the content path and the desired navigation state. In this example, `getMatchedUrlPattern` returns the portion of URL from the incoming request that matches with a pattern configured on a site.

```
String href = action.getSiteState().getMatchedUrlPattern() + action.getContentPath() +
 action.getNavigationState();
```

If the site definition in this application specifies domain pattern matching, then the link would be:

```
String href = action.getContentPath() + action.getNavigationState();
```

If it matches a domain pattern, `getMatchedUrlPattern()` is blank so you can use the following for either domain or URL pattern matching:

```
String href = action.getSiteState().getMatchedUrlPattern() + action.getContentPath() +
 action.getNavigationState();
```

This does not handle the case where the site ID is passed, such as preview passing the site ID. To handle all these cases, you can add `com.endeca.infront.site.SiteUtils.getSiteUrl` to return a site-specific URL.

For example:

```
SiteUtils.getSiteUrl(action.getSiteState(), action.getContentPath() +
 action.getNavigationState())
```

`FunctionTags` also has a `getSiteUrl` method so you can call this from a JSP file as well. For example, in the `userPanel.jsp` file:

```
<a href='<c:url value="${util:getSiteUrl(siteState, '/about-us')}" />'>
        About Us
</a>
```

A link to an arbitrary URL does not require combining fields, since the `UrlAction` object includes a method for directly retrieving a configured URL:

```
String href = action.getUrl();
```

Most of the Discover Electronics cartridge renderers use the `<discover:link>` tag, defined in `WEB-INF\tags\discover\link.tag`. The tag makes use of the `getUrlForAction` function declared in `WEB-INF\tlds\functions.tld` and defined in `WEB-INF\classes\com\endeca\infront\refapp\support\FunctionTags.java`.

About using Actions with the packaged services

The packaged services in Oracle Tools and Frameworks return specific actions for the included cartridges.

The following is an Experience Manager example of the results of a guided search service query for the URI `http://localhost:8006/assembler-authoring/json/services/guidedsearch?Ntt=pink+camera`, serialized to JSON:

```
{
    "@type": "GuidedSearchService",
    "name": "Guided Search Service",
    "navigation": { … },
    "breadcrumbs": { … },
    "resultsList": {
        "@type":"ResultsList",
        "totalNumRecs":228,
        "sortOptions":[
            {
                "@class":"com.endeca.infront.cartridge.model.SortOptionLabel",
                "selected":true,
                "navigationState":"?Ntt=pink+camera",
                "contentPath":"\/services\/guidedsearch",
                "siteRootPath":"\/pages",
                "siteState":{
                    "@class":"com.endeca.infront.site.model.SiteState",
                    "contentPath":"\/services\/guidedsearch",
                    "siteId":"\/DiscoverElectronics",
                    "properties":{
                                }
                            },
                "label":"Relevance"
            },
```

```
            { … }
        ],
        "firstRecNum":1,
        "lastRecNum":12,
        "pagingActionTemplate":{ ... },
        "recsPerPage":12,
        "records":[
            {
                "@class":"com.endeca.infront.cartridge.model.Record",
                "detailsAction":{
                    "@class":"com.endeca.infront.cartridge.model.RecordAction",
                    "recordState":"\/Canon\/Digital-IXUS-80-IS\/_\/A-1439032",
                    "contentPath":"\/services\/recorddetails",
                    "siteRootPath":"\/pages",
                    "siteState":{
                        "@class":"com.endeca.infront.site.model.SiteState",
                        "contentPath":"\/services\/guidedsearch",
                        "siteId":"\/DiscoverElectronics",
                        "properties":{
                                    }
                                }
                 },
                 "numRecords":3,
                 "attributes":{},
                 "records":[ ... ]
            },
            { content removed from this example }
        ]
    },
    "content removed from this example"
}
```

Note that the `sortOptions` returned for the Results List cartridge include the Action fields required to create a URL for the navigation state resulting from modifying the sort order. Sorting by Price (Ascending) requires constructing a URL with the appropriate `navigationState` and `siteState`, resulting in `http://localhost:8006/assembler-authoring/json/services/guidedsearch?Ns=product.price|0&Ntt=pink+camera`. Querying this URL returns the JSON response for the re-ordered results.

Similarly, each of the records returned in the Results List includes the Action fields for an associated record details page. Using the `/services/recorddetails` content root and the `recordState` for the Slim Camera Case results in the URL `http://localhost:8006/assembler-authoring/json/services/recorddetails/Kodak/Slim-Camera-Case/_/A-2707821`. Querying this URL returns the record details for the Slim Camera Case.

The following is an Oracle Commerce Guided Search (without Experience Manager) example of the results of the same guided search service query for the URI `http://localhost:8006/assembler-authoring/json/services/guidedsearch?Ntt=pink+camera`, serialized to JSON:

```
{
    "@type": "GuidedSearchService",
    "name": "Guided Search Service",
    "navigation": { … },
    "breadcrumbs": { … },
    "resultsList": {
        "@type":"ResultsList",
        "totalNumRecs":228,
        "sortOptions":[
            {
```

```
                "@class":"com.endeca.infront.cartridge.model.SortOptionLabel",
                "selected":true,
                "navigationState":"?Ntt=pink+camera",
                "contentPath":"\/guidedsearch",
                "siteRootPath":"\/services",
                "siteState":{
                    "@class":"com.endeca.infront.site.model.SiteState",
                    "contentPath":"\/guidedsearch",
                    "siteId":"\/DiscoverElectronics",
                    "properties":{
                                }
                            },
                "label":"Relevance"
            },
            { … }
        ],
        "firstRecNum":1,
        "lastRecNum":12,
        "pagingActionTemplate":{ ... },
        "recsPerPage":12,
        "records":[
            {
                "@class":"com.endeca.infront.cartridge.model.Record",
                "detailsAction":{
                    "@class":"com.endeca.infront.cartridge.model.RecordAction",
                    "recordState":"\/Canon\/Digital-IXUS-80-IS\/_\/A-1439032",
                    "contentPath":"\/recorddetails",
                    "siteRootPath":"\/services",
                    "siteState":{
                        "@class":"com.endeca.infront.site.model.SiteState",
                        "contentPath":"\/guidedsearch",
                        "siteId":"\/DiscoverElectronics",
                        "properties":{
                                    }
                                }
                 },
                 "numRecords":3,
                 "attributes":{},
                 "records":[ ... ]
            },
            { content removed from this example }
        ]
    },
    "content removed from this example"
}
```

Note the differences from the Experience Manager example for the `contentPath` and `siteRootPath` values.

Working with URL parameters

The `navigationStateBuilder` handles both Oracle-specific and non-Oracle URL parameters.

All URL parameters are parsed into the parameters map in the `NavigationState` object that represents the user's current navigation state. Oracle-specific parameters are used in constructing MDEX Engine queries. All other parameters are included in the navigation state or record state fields on the Action object in the output model. You can change this default behavior by specifying which parameters to remove when generating Actions:

Property	Description
removeAlways	A list of URL parameters that should be removed from all Actions.
removeOnUpdateFilterState	A list of URL parameters that should be removed from an Action when the Action represents a change in the filter (search or navigation) state.
removeOnClearFilterState	A list of URL parameters that should be removed from an Action when the user clears the filter state of all search and navigation selections.

URL configuration in the reference application

URL configuration in the Discover Electronics reference application is located in the Assembler context file, WEB-INF\assembler-context.xml. Configuration is divided between the navigationStateBuilder and the NavigationCartridgeHandler.

The configuration for the navigationStateBuilder specifies a urlFormatter to use when serializing a NavigationState:

```
<!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ Navigation state (including record state) and related config
-->

<bean id="navigationStateBuilder" scope="request"
      class="com.endeca.infront.navigation.url.UrlNavigationStateBuilder">
    <property name="urlFormatter" ref="seoUrlFormatter" />
    <property name="mdexRequestBroker" ref="mdexRequestBroker"/>
    <property name="defaultSearchKey" value="All" />
    <property name="defaultMatchMode" value="ALLPARTIAL" />
    <property name="defaultFilterState" ref="defaultFilterState"/>
        <!-- Filter state properties removed from this example -->
    </property>
```

Note

The seoUrlFormatter bean is defined in the imported endeca-seo-url-config file.

Configuring URL parameters

The configuration for the navigationStateBuilder also lets you specify the URL parameters to remove from the request URL when serializing a NavigationState or RecordState:

```
                    <property name="removeAlways">
        <list>
            <value>contentText</value>
            <value>Nty</value>
            <value>Dy</value>
            <value>collection</value>
```

```
        </list>
    </property>
    <property name="removeOnUpdateFilterState">
        <list>
            <value>No</value>
        </list>
    </property>
    <property name="removeOnClearFilterState">
        <list>
            <value>Ns</value>
            <value>Nrpp</value>
            <value>more</value>
        </list>
    </property>
</bean>
```

Configuration for navigation and record paths

The content paths that prefix navigation and record states when creating Action URLs are configured in the `actionPathProvider` of the `NavigationCartridgeHandler` as sets of key-value pairs:

```
<bean id="NavigationCartridgeHandler" abstract="true">
        <property name="navigationState" ref="navigationState" />
        <property name="mdexRequestBroker" ref="mdexRequestBroker" />
        <property name="actionPathProvider" ref="actionPathProvider"/>
        <property name="siteState" ref="siteState"/>
        <property name="userState" ref="${user.state.ref}"/>
    <bean id="actionPathProvider" scope="request"
 class="com.endeca.infront.refapp.navigation.BasicActionPathProvider">
    <constructor-arg index="0" ref="contentSource"/>
    <constructor-arg index="1" ref="httpServletRequest"/>
    <!-- navigationActionUriMap -->
    <constructor-arg index="2">
        <map>
            <entry key="^/pages/[^/]*/mobile/detail$"value="/mobile/browse"/>
            <entry key="^/pages/[^/]*/services/recorddetails/.*$"value="/services/
guidedsearch"/>
            <entry key="^/pages/[^/]*/detail$"value="/browse"/>
            <entry key="^/services/.*$"value="/services/guidedsearch"/>
        </map>
    </constructor-arg>
    <!-- recordActionUriMap -->
    <constructor-arg index="3">
        <map>
            <entry key="^/pages/[^/]*/mobile/.*$"value="/mobile/detail"/>
            <entry key="^/pages/[^/]*/services/.*$"value="/services/recorddetails"/>
            <entry key="^/pages/[^/]*/.*$"value="/detail"/>
            <entry key="^/services/.*$"value="/recorddetails"/>
        </map>
    </constructor-arg>
    <constructor-arg index="4" ref="siteState"/>
</bean>
```

URL formatter configuration

The Discover Electronics reference application serializes `NavigationState` objects through the use of a `UrlNavigationStateBuilder` configured with a `UrlFormatter`. By default, the application is

configured for search engine optimized (SEO) URLs using the `SeoUrlFormatter` class, but it also includes a `BasicUrlFormatter` for creating basic URLs.

The basic URL formatter

The following properties can be set on the `basicUrlFormatter` bean:

Property	Description
`defaultEncoding`	Specifies the default query encoding, for example, `UTF-8`.
`prependQuestionMark`	Specifies whether a question mark is prepended to the URL parameter portion of the URL, after the servlet path.

The configuration in `WEB-INF\endeca-url-config` is shown below:

```
<!--
    ########################################################################
    # BEAN: basicUrlFormatter
    #
    # This is an UrlFormatter that generates "classic" URLs.
    #
-->

<bean id="basicUrlFormatter"
 class="com.endeca.soleng.urlformatter.basic.BasicUrlFormatter">
    <property name="defaultEncoding">
        <value>UTF-8</value>
    </property>

    <property name="prependQuestionMark">
        <value>true</value>
    </property>
</bean>
```

The SEO URL formatter

The following properties can be set on the `seoUrlFormatter` bean:

Property	Description
`defaultEncoding`	Specifies the default query encoding, for example, `UTF-8`.
`pathSeparatorToken`	The separator token used to separate the path section of the URL from the parameter section.
`pathKeyValueSeparator`	The character used to separate key/value pairs in the parameter section of the URL.

Property	Description
pathParamKeys	Specifies the URL parameter keys for the following: • The parameter key used for record detail links. The default value is R. • The parameter key used for aggregate record detail links. The default value is A. • The parameter key used for navigation state. The default value is N.
navStateFormatter	The NavStateFormatter that maps navigation state information to URL path keywords.
ERecFormatter	The ERecFormatterthat maps Endeca record attributes to URL path keywords.
aggrERecFormatter	The AggrERecFormatter that maps aggregate record attributes to URL path keywords.
navStateCanonicalizer	Specifies the canonicalizer used to sort URL parameters to ensure that included parameters are arranged a specific order.
useNavStateCanonicalizer	Determines whether or not the canonicalizer specified in navStateCanonicalizer is used. The default value is true. This value is ignored if the canonicalLinkBuilder enables canonical links.
urlParamEncoders	A list of UrlParamEncoder objects to use for encoding URL parameters.

The configuration in WEB-INF\endeca-seo-url-config is shown below:

```
<!--
    ################################################################################
    # BEAN: seoUrlFormatter
    #
    # This is the SEO URL formatter, which is responsible for
    # transforming UrlState objects into URL strings.
    #
-->
    <bean id="seoUrlFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

    <property name="defaultEncoding">
        <value>UTF-8</value>
    </property>

    <property name="pathSeparatorToken">
        <value>_</value>
    </property>

    <property name="pathKeyValueSeparator">
        <value>-</value>
    </property>

    <property name="pathParamKeys">
```

```
        <list>
            <value>R</value>
            <value>A</value>
            <value>N</value>
        </list>
    </property>

    <property name="navStateFormatter">
        <ref bean="navStateFormatter"/>
    </property>

    <property name="ERecFormatter">
        <ref bean="erecFormatter"/>
    </property>

    <property name="aggrERecFormatter">
        <ref bean="aggrERecFormatter"/>
    </property>

    <property name="navStateCanonicalizer">
        <ref bean="navStateCanonicalizer"/>
    </property>

    <property name="useNavStateCanonicalizer">
        <value>false</value>
    </property>

    <property name="urlParamEncoders">
        <list>
            <ref bean="N-paramEncoder"/>
        </list>
    </property>
</bean>
```

About working with canonical links

Configure the Assembler to add canonical link support to the root content item.

The canonical link configuration in the Discover Electronics reference application is located in the Assembler context file, `WEB-INF\assembler-context.xml`. Configuration is handled by the `canonicalLinkBuilder` which constructs links for navigation state and record state URLs that include the canonical link element.

The Canonical Link Builder

The following properties can be set on the `canonicalLinkBuilder`:

Property	Description
`objectLocator`	Allows the retrieval of services without explicit injection. In this case, it is used to reference the framework for retrieving the `recordState` and `navigationState` for the current request.
`recordStateId`	The ID of the `recordState` being retrieved, not the actual `recordState`.

Property	Description
navigationStateId	The ID of the navigationState being retrieved, not the actual navigationState.
siteStateId	The ID of the siteState being retrieved, not the actual siteState.
includedParameters	The list of URL parameters that are included in the canonical link.

The configuration for the canonicalLinkBuilder specifies an objectLocator to use when creating canonical links:

```
<bean id="assemblerFactory"
 class="com.endeca.infront.assembler.spring.SpringAssemblerFactory">
...
    <constructor-arg>
        <list>
            ...
            <bean class="com.endeca.infront.navigation.url.event.CanonicalLinkBuilder">
                <property name="objectLocator" ref="springUtility"/>
                <property name="recordStateId" value="recordState"/>
                <property name="navigationStateId" value="navigationState"/>
                <property name="siteStateId" value="siteState"/>
                <property name="includedParameters">
                    <list>
                        <value>R</value>
                        <value>A</value>
                        <value>N</value>
                        <value>Ntt</value>
                    </list>
                </property>
            </bean>
        </list>
    </constructor-arg>
</bean>
```

Output content items

The Assembler API returns navigation state, record state, and site state content items as output from the CanonicalLinkBuilder. The following examples are JSON representations of the output.

NavigationState

```
{
    name: "Static Page Slot",
    ...,
    canonicalLink: {
        @class: "com.endeca.infront.cartridge.model.NavigationAction",
        navigationState: "/Canon/cameras/_/N-1z141xuZ1z141yaZ25y6Zej4?format=json",
        contentPath: "/browse",
        siteRootPath: "/pages",
        label: ""
```

```
    }
}
```

RecordState

```
{
    name: "Static Page Slot",
    ...,
    canonicalLink: {
        @class: "com.endeca.infront.cartridge.model.RecordAction",
        recordState: "/_/A-1318562?format=json",
        contentPath: "/detail",
        siteRootPath: "/pages",
        label: ""
    }
}
```

SiteState

```
canonicalLink:{
     @class":"com.endeca.infront.cartridge.model.NavigationAction",
     navigationState:"\/cameras\/_\/N-25y6",
     contentPath:"\/browse",
     siteRootPath:"\/pages",
     siteState":{"@class":"com.endeca.infront.site.model.SiteState",
     contentPath":"\/browse\/cameras\/_\/N-25y6",
     siteId:"\/DiscoverElectronics",
     properties:{}
     }
  label:""}
  }
}
```

For each of the content items, a JSP file can render output as in this example:

```
<link rel="canonical" href="<c:url
 value='${util:getUrlForAction(rootComponent.canonicalLink)}'/>"/>
```

Preparing your application

This section describes the basic requirements and recommendations for writing your application.

Related links

- *Optimizing Application URLs* (page 99)
- Preparing your dimensions (page 115)

Preparing your dimensions

If you intend to display dimensions or dimension values in your URLs, you must configure each of the dimensions to Show with record and Show with record list.

You only need to configure the dimensions you intend to include in URLs. Configuring all dimensions to Show with record and Show with record list may have performance implications.

To configure a dimension to Show with record and Show with record list:

1. Open your project in Developer Studio.
2. From the Project Explorer on the left, click Dimensions.

   The Dimensions dialog displays.
3. Select the dimension you need to edit.
4. Select the Show with record list checkbox.
5. Select the Show with record checkbox.
6. Click OK.
7. Save your changes.

For more information, please refer to the *Oracle Developer Studio Help*.

Preparing your properties

If you intend to display record properties in your URLs, you must configure each property to Show with record and Show with record list.

You only need to configure the properties you intend to include in URLs. Configuring all properties to Show with record and Show with record list may have performance implications.

To configure a property to Show with record and Show with record list:

1. Open your project in Developer Studio.
2. From the Project Explorer on the left, click Dimensions.

   The Dimensions dialog displays.
3. Select the dimension you need to edit.
4. Select the Show with record list checkbox.
5. Select the Show with record checkbox.
6. Click OK.

7. Save your changes.

For more information, please refer to the *Oracle Developer Studio Help*.

Handling images and external JavaScript files

When you modify your application to produce optimized URLs, it is important to ensure that the server can still locate resources requested by the application, such as image files, JavaScript files, and CSS files.

Relative URLs are partial URLs that omit host and port information. There are two types of relative URLs:

- "Site-relative" URLs are relative to the root directory on the site that hosts the Web page, for example: `/sitemap.htm`
- "Non-site-relative" URLs are relative to their parent pages, for example: `../sitemap.htm`

Because relative paths are relative to the URL that is requested, not the URL that is ultimately resolved, optimized URLs may create unresolved links when external resources are referenced. When using optimized URLs, Endeca recommends replacing non-site-relative URLs with site-relative URLs to ensure that links resolve properly.

URL transitioning

Managing redirects is an important aspect of search engine optimization. In order to maintain page rank for resources within your website, you need an effective strategy to manage URL changes.

As you transition from traditional URLs to optimized URLs, or when you change the configuration of your optimized URLs, it is important to ensure that:

- Links throughout your Web site are updated
- Links to external resources (such as image files, CSS, or Javascript files) are updated
- External links to your Web site are permanently redirected to the new URLs

Links throughout your own Web site and to your own external resources can simply be updated to the new URLs. However, external references to your site must be redirected in order to prevent unresolved links.

The URL optimization classes are responsible for transforming URLs into search and navigation queries, and vice-versa. They do not implement redirect logic. In order to redirect incoming requests, you must include the appropriate logic in your application controller. By comparing an inbound URL to the canonical (optimized) form, you can redirect to the canonical URL in cases where the inbound URL is different.

Oracle recommends including HTTP 301 redirects. Unlike HTTP 302 redirects, which collect ranking information and index content on a site against the source URL, 301 redirects apply this information to the destination URL.

Building optimized URLs

This section describes the basic tasks for using the URL optimization classes to build search engine-optimized URLs.

Related links

- *Optimizing Application URLs* (page 99)
- Core URL optimization classes (page 117)
- Overview of building URLs using the URL optimization classes (page 117)
- Parsing an incoming query and sending it to an MDEX Engine (page 118)
- Informing the UrlState of the navigation state (page 118)
- Creating link URLs from a UrlState (page 119)

Core URL optimization classes

The primary classes and interfaces of the URL Optimization API are `UrlState`, `UrlFormatter`, and `QueryBuilder`.

UrlState

A `UrlState` instance represents the URL, including any parameters, for a particular navigation state in Your application. You typically create a `UrlState` by using a `UrlFormatter` to parse a URL string. You then inform the `UrlState` of the navigation state that it represents by passing it a set of query results. When the `UrlState` is informed, you can modify it in order to generate URLs representing links to other states in your application, such as selecting refinements.

UrlFormatter

A `UrlFormatter` is responsible for parsing URL strings into `UrlState` objects and transforming `UrlState` objects back into URLs. The `SeoUrlFormatter` is a highly configurable implementation of `UrlFormatter` that parses and generates search engine-optimized URLs.

QueryBuilder

A `QueryBuilder` marshals `UrlState` objects into MDEX Engine queries. The `BasicQueryBuilder` is an implementation of `QueryBuilder` that creates `ENEQuery` objects from a given `UrlState`.

For more information about these and other classes, refer to the *Assembler API Reference (Javadoc)*.

Overview of building URLs using the URL optimization classes

Building optimized URLs with the Assembler API requires passing in the necessary configuration and instantiating the required objects.

The high-level process is as follows:

1. Set up your basic application configuration with a `BasicQueryBuilder` and `SeoUrlFormatter`.

   How you create and configure the `QueryBuilder` and `UrlFormatter` may vary depending on your application, but they should be should be scoped at a global or application level.

2. Handle requests by parsing the incoming query and sending it to an MDEX Engine.
3. Inform a `UrlState` object of the navigation state.
4. Modify the `UrlState` object by adding or removing URL parameters.

5. Generate a URL from the `UrlState`.

Parsing an incoming query and sending it to an MDEX Engine

Because it is possible for optimized URLs not to contain query string parameters (these parameters can be stored in the path), you cannot rely on the `UrlENEQuery` class to create an `ENEQuery` object from a URL.

Instead, use a `UrlFormatter` to parse the incoming request URL in order to populate the `UrlState` with the current URL query parameters, then use a `QueryBuilder` to create the `ENEQuery` from the `UrlState`.

To parse an incoming request and query an MDEX Engine, follow these steps:

1. Parse the request into a `UrlState` instance.

   For example:

```
UrlState requestUrlState = urlFormatter.parseRequest(request);
```

2. Build an `ENEQuery` based on the `UrlState`.

   For example:

```
ENEQuery eneQuery = queryBuilder.buildQuery(requestUrlState);
```

3. Execute the request and retrieve the results.

   For example:

```
HttpENEConnection conn = new HttpENEConnection(mdexHost, mdexPort);
ENEQueryResults eneQueryResults = conn.query(eneQuery);
```

Informing the UrlState of the navigation state

Informing is the process of providing the `UrlState` object with information about the current query results.

From this information, the `UrlState` object creates either a `NavStateUrlParam` if the query results are from a navigation query, an `ERecUrlParam` if the query results are from a record detail query, or an `AggrERecUrlParam` if the query results are from an aggregated record detail query.

The `SeoUrlFormatter` can use the extra information in these objects to generate customized URLs based on the current navigation state or properties and dimensions associated with these results.

To inform a `UrlState` of the current navigation state:

1. Add code similar to the following:

```
urlState.inform(eneQueryResults);
```

You can generate properly formatted URLs representing either the current navigation state, a record detail link, or an aggregated record detail link. Note that of these three possiblities, only the record detail link is guaranteed

to be complete when calling `inform` on an empty `UrlState`. A navigation URL would be correct but, without further modification, only reflects the selected dimension values (the N parameter values). An aggregated record detail URL would not work without adding the required An and Au parameters.

The intent of the `inform()` method is to give the `UrlFormatter` and `UrlState` access to property and dimension information, not to copy your query. In some cases a complete query URL can only be created through a combination of using `UrlFormatter.parseRequest()` on the initial request and calling `UrlState.setParam()` as needed in addition to using `inform()`.

Creating link URLs from a UrlState

To create link URLs on a particular page to different navigation states within your application, modify the `UrlState` and then transform the modified `UrlState` to a URL string.

This procedure requires that you have an informed `UrlState` representing the current navigation state of your page.

To create a link URL, follow these steps:

1. Modify the `UrlState` to reflect a different navigation state in your application.

   For example, the following statement creates a refinement link for a Guided Navigation component in your application:

   ```
   UrlState refinedUrlState =
          informedUrlState.selectRefinement(refDim, refDimVal, true);
   ```

   The final parameter indicates whether the modification should be performed on a cloned version of the current `UrlState`, and should typically be `true`. For instance, in the case of a Guided Navigation component, you would loop through the possible refinements and create a modified `UrlState` based on the current `UrlState` for each refinement link. If you wanted to select several refinements in the same URL, you would pass `false` as the value of this parameter.

   For further details about additional methods that can be used to modify a `UrlState`, please refer to the *Assembler API Reference (Javadoc)*.

2. Generate the URL string from the modified `UrlState`.

   ```
   String refinedUrl = refinedUrlState.toString();
   ```

   The `UrlState.toString()` method calls the `formatString()` method of the `UrlFormatter` that constructed the `UrlState` instance.

Configuring URLs

The following sections provide information about creating and using a URL configuration file to optimize your URLs. The information and examples provided in this section relate to basic URL configuration tasks, and do not cover the entire breadth of URL optimization capabilities. Oracle recommends consulting the API documentation as you develop your application.

Related links

- *Optimizing Application URLs* (page 99)
- Anatomy of an optimized URL (page 120)
- About the URL configuration file (page 121)
- Creating a URL configuration file (page 122)
- About optimizing the misc-path (page 125)
- Configuring the path-param-separator (page 145)
- About optimizing the path-params and query string (page 145)
- Using the URL configuration file with your application (page 150)

Anatomy of an optimized URL

An optimized Oracle Commerce Guided Search URL is made up of four configurable sections.

General URL References

When referring to URLs in general, the API documentation may use the terms "base URL" and "URL query parameters." The "base URL" is the part of the URL that precedes the question mark.

For example, in the URL:

```
http://www.example.com/pathparam1/pathparam2/pathparam3/results?queryparam=123
```

the base URL is the string that appears before the question mark:

```
http://www.example.com/pathparam1/pathparam2/pathparam3/results
```

Optimized URLs

For reference purposes, the documentation identifies four distinct sections of optimized URLs:

- misc-path
- path-param-separator
- path-params
- query string

For example, the following URL is broken down into subsections:

```
http://localhost:8888/controller[/Wine-Red-Merlot/Napa/Pine-Ridge/_/N-12ZafZfd?Ne=123]
```

The sections of the URL encased in square brackets can be broken down into the following components:

```
[/<misc-path>][/<path-param-separator>][/<path-params>][?<query-string>]
```

The components correspond to the following strings:

Section	String
misc-path	Wine-Red-Merlot/Napa/Pine-Ridge
path-param-separator	_
path-params	N-12ZafZfd
query string	Ne=123

misc-path

This section of the URL incorporates keywords into the URL in order to create user-friendly and search engine-optimized URLs. The misc-path section of the optimized URL can be generated based on dimension names, dimension values, ancestor names, and record properties. The misc-path component is largely ignored by the application.

path-param-separator

The path-param-separator component is used to identify the end of the misc-path and the starting point for path parameters. This string is configurable.

path-params

Together with the query string, the path-params segment of the URL represents the current state of the application. This may include the numerical representation of the navigation state or a specific record, as well as any other parameter key-value pairs that have an effect on the displayed content. This component can be configured to contain several parameters that would typically be included as part of the query string in traditional URLs, such as the `N`, `Ne`, `Ntt`, and `R` parameters.

query string

The query string component of the URL follows the question mark character. The combination of the path-params and query string represents the current state of the application. parameters that are not configured to appear in the path-params section of the URL – such as `N`, `Ne`, `Ntt`, and `R` – appear in the query string.

About the URL configuration file

The example application uses an XML file named `urlconfig.xml` to configure the format of the URLs that it generates.

The reference application uses the Spring Framework for this configuration file. Although the Assembler API does not require the Spring Framework, it supplies a convenient and flexible configuration mechanism. In addition, if you plan to use the Sitemap Generator with your application, Oracle recommends using a `urlconfig.xml` file to configure your optimized URLs, because the Sitemap Generator relies on the same format for configuration. If you need further information about the Spring Framework syntax, please consult the documentation provided with the Spring Framework.

The URL configuration file contains basic configurations for the following objects:

- A `BasicQueryBuilder` to transform `UrlState` objects into `ENEQuery` objects
- An `SeoUrlFormatter` to transform `UrlState` objects into optimized URL strings

By specifying settings for additional components in the configuration file, you can configure the following aspects of your URLs:

- the dimension values and properties to include in the misc-path
- canonicalization options for dimensions in the misc-path
- the path-param-separator
- parameters to be included in the path-params instead of the query string
- base-36 encoding for numeric parameters

Creating a URL configuration file

A URL configuration file defines a `BasicQueryBuilder` and a top-level `SeoUrlFormatter`.

To create a URL configuration file, follow these steps:

1. Create a basic query builder that invokes the `com.endeca.soleng.urlformatter.basic.BasicQueryBuilder` class:

   For example:

```
 <bean id="queryBuilder"
class="com.endeca.soleng.urlformatter.basic.BasicQueryBuilder">
 </bean>
```

2. Add the following properties:

Option	Description
`queryEncoding`	Specifies the query encoding. For example: `<value>UTF-8</value>`
`baseUrlENEQuery`	Sets the `baseUrLENEQuery`. This query is used to create the `UrlENEQuery` if the `UrlState` is not associated with a record or navigation state. If this value is `<null/>`, a new query is created.
`baseNavigationUrlENEQuery`	Sets the `baseNavigationUrlENEQuery`. This query is used to create the `UrlENEQuery` if the `UrlState` is associated with a navigation state (but not a record or aggregate record). If this value is `<null/>`, a new query is created.
`baseERecUrlENEQuery`	Sets the `baseERecUrlENEQuery`. This query is used to create the `UrlENEQuery` if the `UrlState` is associated with a record (but not an aggregate record). If this value is `<null/>`, a new query is created.

Option	Description
baseAggrERecUrlENEQuery	Sets the baseAggrERecUrlENEQuery. This query is used to create the UrlENEQuery if the UrlState is associated with an aggregate record. If this value is <null/>, a new query is created.
defaultUrlENEQuery	Sets the detaultUrlENEQuery. This query is used to create the UrlENEQuery if the UrlState contains no parameters.

For example:

```
 <bean id="queryBuilder"
class="com.endeca.soleng.urlformatter.basic.BasicQueryBuilder">

   <property name="queryEncoding">
     <value>UTF-8</value>
   </property>

   <property name="baseUrlENEQuery">
     <value><![CDATA[N=0&Ns=P_Price|1&Nr=8020]]></value>
   </property>

   <property name="baseNavigationUrlENEQuery">
     <value><![CDATA[N=0&Ns=P_Price|1&Nr=8020]]></value>
   </property>

   <property name="baseERecUrlENEQuery">
     <null/>
   </property>

   <property name="baseAggrERecUrlENEQuery">
     <value>An=0</value>
     <null/>
   </property>

   <property name="defaultUrlENEQuery">
     <value>N=0</value>
   </property>

 </bean>
```

3. Create a top-level seoUrlFormatter bean to invoke the com.endeca.soleng.urlformatter.seo.SeoUrlFormatter class:

   For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">
 </bean>
```

4. Add the following properties:

Option	Description
`defaultEncoding`	Specifies the default query encoding. For example: `<value>UTF-8</value>`
`pathSeparatorToken`	Specifies the character used to separate the misc-path from the path-params section in URLs.
`pathKeyValueSeparator`	Specifies the character used to separate key-value pairs in the path parameter section of the URL.

For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

   <property name="defaultEncoding">
     <value>UTF-8</value>
   </property>

   <property name="pathSeparatorToken">
     <value>_</value>
   </property>

   <property name="pathKeyValueSeparator">
     <value>-</value>
   </property>

<!-- additional elements deleted from this example --!>

  </bean>
```

5. Set any required properties to specify configuration beans.

   Note

   The instructions in this chapter explain which of beans are required for each task. You can set these properties on your `SeoUrlProvider` object as you work through the chapter.

   For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

   <property name="pathParamKeys">
     <list>
       <value>R</value>
       <value>A</value>
       <value>An</value>
       <value>Au</value>
       <value>N</value>
```

```
        <value>No</value>
        <value>Np</value>
        <value>Nu</value>
        <value>D</value>
        <value>Ntt</value>
        <value>Ne</value>
      </list>
    </property>

    <property name="navStateFormatter">
      <ref bean="navStateFormatter"/>
    </property>

    <property name="ERecFormatter">
      <ref bean="erecFormatter"/>
    </property>

    <property name="aggrERecFormatter">
      <ref bean="aggrERecFormatter"/>
    </property>

    <property name="navStateCanonicalizer">
      <ref bean="navStateCanonicalizer"/>
    </property>

    <property name="urlParamEncoders">
      <list>
        <ref bean="N-paramEncoder"/>
        <ref bean="Ne-paramEncoder"/>
        <ref bean="An-paramEncoder"/>
      </list>
    </property>

  </bean>
```

After you have created the basic URL configuration file, you create additional beans to specify further configuration for the misc-path and path-params. Follow the procedures in the sections below to complete your URL configuration.

Related links

- Using the URL configuration file with your application (page 150)

About optimizing the misc-path

You can configure dimensions, dimension values, record properties, and aggregate record properties to display in the misc-path of URLs. You can also specify the order in which dimension and dimension values display. The `urlconfig.xml` file provides a simple and convenient method for configuring these options.

navStateFormatter

The `navStateFormatter` bean invokes the `com.endeca.soleng.urlformatter.seo.SeoNavStateFormatter` class to define `dimLocationFormatters` for each dimension that you want to configure.

Using the dimLocationFormatters defined in the navStateFormatter bean, you can configure URLs for navigation pages to include dimension names, roots, ancestors, and dimension value names in the misc-path of URLs for navigation pages.

For example, the following URL is for the navigation state Region > Napa:

```
http://localhost:8888/endeca_jspref/controller.jsp?&Ne=8&N=4294967160
```

By optimizing the URL, it can be formatted as follows:

```
http://localhost:8888/urlformatter_jspref/controller/Napa/_/N-1z141vc/Ne-8
```

navStateCanonicalizer

The navStateCanonicalizer bean invokes the com.endeca.soleng.urlformatter.seo.SeoNavStateCanonicalizer to order the dimension and dimension value names included in the misc-path for navigation pages. For example, an end user can reach the Wine Type > Red, Region > Napa page by navigating first to Wine Type > Red and then to Region > Napa, or by navigating to Region > Napa and then Wine Type > Red. To avoid two syntactically different URLs for the same Wine Type > Red, Region > Napa page, you can use the navStateCanonicalizer to standardize the order of dimension and dimension values in the misc-path.

Note

By design, the URL optimization classes prevent the creation of syntactically different URLs by canonicalizing keywords. You can choose from a number of configuration options to control the arrangement of keywords, but the URLs are always canonicalized.

erecFormatter

URL optimization for record detail pages is configured separately from navigation pages and aggregate record details pages. The erecFormatter bean invokes the com.endeca.soleng.urlformatter.seo.SeoERecFormatter class to define dimLocationFormatters for each dimension that you want to configure.

The same options for including dimension names, roots, ancestors, and dimension value names are available for record detail pages as are available for navigation pages. While the urlconfig.xml configuration file uses the same dimLocationFormatters for the erecFormatter and the aggErecFormatter as are used for the navStateFormatter, this is not a requirement. You can create separate dimLocationFormatters for navigation pages, record detail pages, and aggregate record detail pages.

aggrERecFormatter

URL optimization for aggregate record detail pages is configured separately from navigation pages and record details pages as are available for navigation pages. The aggrERecFormatter bean invokes the com.endeca.soleng.urlformatter.seo.SeoAggrERecFormatter class to define dimLocationFormatters for each dimension that you want to configure. The same options for including dimension names, roots, ancestors, a nd dimension value names are available for aggregate record detail pages. While the urlconfig.xml configuration file uses the same dimLocationFormatters for the aggrERecFormatter and the erecFormatter as are used for the navStateFormatter, this is not a requirement. You can create separate dimLocationFormatters for navigation pages, record detail pages, and aggregate record detail pages.

Formatting misc-path strings in optimized URLs

The SeoNavStateFormatter, SeoERecFormatter, and SeoAggrERecFormatter use StringFormatter objects to format dimension and record property strings that display in URLs.

You can format the strings in the misc-path section of a URL by using string formatters that are predefined in the Assembler API. Formatting may include changing capitalization or applying a regular expression to replace portions of the string.

There are several `StringFormatter` objects in the Assembler API:

- `LowerCaseStringFormatter` — formats path-keyword data into lower case.
- `UpperCaseStringFormatter`— formats path-keyword data into upper case.
- `UrlEncodedStringFormatter` — URL-encodes strings.
- `RegexStringFormatter` — You can create a new `RegexStringFormatter` object and customize the `pattern`, `replacement`, and `replaceAll` properties to perform custom string formatting. For more information about the properties, please refer to the *Assembler API Reference (Javadoc)*.

To define `StringFormatter` objects in the `urlconfig.xml` file:

1. Create a bean to invoke a `StringFormatter` class.

   This example shows the configuration for a `RegexStringFormatter` that replaces all non-word character sequences with a single "`-`" character:

```
<bean class="com.endeca.soleng.urlformatter.seo.RegexStringFormatter">
  <property name="pattern">
    <value><![CDATA[[\W_&&[^\u00C0-\u00FF]]+]]></value>
  </property>

  <property name="replacement">
    <value>-</value>
  </property>

  <property name="replaceAll">
    <value>true</value>
  </property>
</bean>
```

2. Optionally, you can build a `StringFormatterChain` to apply more than one `StringFormatter` to a string in series.

   The following example shows the `defaultStringFormatterChain` that is used throughout the sample `urlconfig.xml` file.

```
<bean name="defaultStringFormatterChain"
    class="com.endeca.soleng.urlformatter.seo.StringFormatterChain">

  <property name="stringFormatters">
    <list>
      <!--
        ###################################################################
        # replace all non-word character sequences with a single '-'
        #
      -->
      <bean class="com.endeca.soleng.urlformatter.seo.RegexStringFormatter">
        <property name="pattern">
          <value><![CDATA[[\W_&&[^\u00C0-\u00FF]]+]]></value>
        </property>
```

```
          <property name="replacement">
            <value>-</value>
          </property>

          <property name="replaceAll">
            <value>true</value>
          </property>
        </bean>

        <!--
          ###################################################################
          # trim leading and trailing '-' characters (if any)
          #
        -->
        <bean class="com.endeca.soleng.urlformatter.seo.RegexStringFormatter">
          <property name="pattern">
            <value><![CDATA[^-?([\w\u00C0-\u00FF][\w-\u00C0-\u00FF]*[\w\u00C0-
\u00FF])-?$]]></value>
          </property>

          <property name="replacement">
            <value>$1</value>
          </property>

          <property name="replaceAll">
            <value>false</value>
          </property>
        </bean>

      </list>
    </property>
  </bean>
```

Note that because `StringFormatterChain` implements `StringFormatter`, you can nest chains. For example:

```
    <bean class="com.endeca.soleng.urlformatter.seo.StringFormatterChain">
      <property name="stringFormatters">
        <list>

          <!-- replace 'Wine Type' with 'Wine' -->

          <bean class="com.endeca.soleng.urlformatter.seo.RegexStringFormatter">
            <property name="pattern">
              <value>Wine Type</value>
            </property>

            <property name="replacement">
              <value>Wine</value>
            </property>

            <property name="replaceAll">
              <value>false</value>
            </property>
          </bean>

          <!-- execute the default string formatter chain -->

          <ref bean="defaultStringFormatterChain"/>
```

```
      </list>
    </property>
  </bean>
```

Optimizing URLs for navigation pages

Using URL optimization, you can include dimension and dimension value names in the misc-path of URLs. You can also choose to canonicalize these dimension and dimension value names in order to avoid duplicate content and to increase your natural search rankings.

Note

For dimensions to display properly in the URL, they must be enabled for display with the record list.

You must create a URL configuration file before completing this procedure.

To optimize URLs for navigation pages:

1. Open your URL configuration file.
2. Create a `navStateFormatter` bean to invoke the `com.endeca.soleng.urlformatter.seo.SeoNavStateFormatter`:

   For example:

```
 <bean id="navStateFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoNavStateFormatter">
 </bean>
```

3. Add a `navStateFormatter` property to your top-level `seoUrlFormatter` bean.

   For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

<!-- additional elements deleted from this example --!>

   <property name="navStateFormatter">
     <ref bean="navStateFormatter"/>
   </property>

 </bean>
```

4. Add a `useDimensionNameAsKey` property on the `navStateFormatter`.

   For example:

```
 <bean id="navStateFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoNavStateFormatter">

   <property name="useDimensionNameAsKey">
     <value>true</value>
```

```
  </property>
</bean>
```

Setting the `useDimensionNameAsKey` to `false` creates a key on the dimension ID numbers.

5. Add a `dimLocationFormatters` property and list each `dimLocationFormatter` bean you plan to define.

   For example:

```
 <bean id="navStateFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoNavStateFormatter">

  <property name="useDimensionNameAsKey">
    <value>true</value>
  </property>

  <property name="dimLocationFormatters">
    <list>
      <ref bean="wineTypeFormatter"/>
      <ref bean="regionFormatter"/>
      <ref bean="wineryFormatter"/>
      <ref bean="flavorsFormatter"/>
    </list>
  </property>

</bean>
```

6. Create a `dimLocationFormatter` for each of the dimensions in the `dimLocationFormatters` list.

   For example:

```
<bean id="regionFormatter"
    class="com.endeca.soleng.urlformatter.seo.SeoDimLocationFormatter">
</bean>
```

Note

The sample `urlconfig.xml` file uses the same `dimLocationFormatter` for navigation pages, record detail pages, and aggregate record detail pages. You can choose to create unique `dimLocationFormatters` for each page type.

7. Add the following properties to each `dimLocationFormatter`:

Property	Description
`key`	In the `navStateFormatter` bean, the `useDimensionNameAsKey` property sets the key type. If you set the `useDimensionNameAsKey` to true, then use the dimension name as the value for this property (for example `<value>Region</value>`). If you set the `useDimensionNameAsKey` to false, use the dimension ID number.

Property	Description
appendRoot	Specifies whether or not to append root dimension values to the URL. Set to true to append root dimension values.
appendAncestors	Specifices whether or not to append ancestor dimension values to the URL. Set to true to append ancestor dimension values.
appendDescriptor	Specifies whether or not to append the selected or descriptor dimension values to the URL. Set to true to append selected or descriptor dimension values.
separator	Specifies the character used to separate dimension roots, ancestors, and descriptor values.
rootStringFormatter	Specifies the bean to format the dimension name. The reference application uses a defaultStringFormatterChain bean to invoke the com.endeca.soleng.urlformatter.seo.StringFormatterCha
dimValStringFormatter	Specifies the bean to format the dimension value names. The reference application uses a defaultStringFormatterChain bean to invoke the com.endeca.soleng.urlformatter.seo.StringFormatterCha The examples below also use a defaultStringFormatterChain bean.

For example:

```
<bean id="regionFormatter"
    class="com.endeca.soleng.urlformatter.seo.SeoDimLocationFormatter">

  <property name="key">
    <value>Region</value>
  </property>

  <property name="appendRoot">
    <value>false</value>
  </property>

  <property name="appendAncestors">
    <value>false</value>
  </property>

  <property name="appendDescriptor">
    <value>true</value>
  </property>

  <property name="separator">
    <value>-</value>
```

```
    </property>

    <property name="rootStringFormatter">
      <ref bean="defaultStringFormatterChain"/>
    </property>

    <property name="dimValStringFormatter">
      <ref bean="defaultStringFormatterChain"/>
    </property>

  </bean>
```

8. Create a `navStateCanonicalizer` bean to invoke the `com.endeca.soleng.urlformatter.seo.SeoNavStateCanonicalizer` class.

   For example:

```
  <bean name="navStateCanonicalizer"
 class="com.endeca.soleng.urlformatter.seo.SeoNavStateCanonicalizer">
  </bean>
```

   Note

   Canonicalizing the dimension and dimension value names in the misc-path also changes the order in which they appear in the path-params section of the URL. For example, if Napa is configured to display before Red in the misc-path, the Napa dimension value ID displays before the Red dimension value ID in the path-params section.

9. Add a `navStateCanonicalizer` property to your top-level `seoUrlFormatter` bean.

   For example:

```
  <bean id="seoUrlFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

<!-- additional elements deleted from this example --!>

    <property name="navStateCanonicalizer">
      <ref bean="navStateCanonicalizer"/>
    </property>

  </bean>
```

10. Configure the `navStateCanonicalizer`.

    For example, the following configuration creates URLs sorted by dimension ID in descending order:

```
  <bean name="navStateCanonicalizer"
 class="com.endeca.soleng.urlformatter.seo.SeoNavStateCanonicalizer">

    <property name="sortByName">
      <value>false</value>
    </property>

    <property name="sortByDimension">
```

```
    <value>true</value>
  </property>

  <property name="ascending">
    <value>false</value>
  </property>

</bean>
```

Note

There a number of possible configuration options for canonicalization.

11.Save and close the file.

Related links

- Preparing your properties (page 115)
- Preparing your dimensions (page 115)
- URL canonicalization (page 101)
- Formatting misc-path strings in optimized URLs (page 126)

Canonicalization configuration options

You can customize the canonicalization of URLs for navigation pages by choosing a sort method, for example by dimension name or dimension ID, and then a sort direction.

The following example configurations use the dimensions:

- Wine Type (dimension ID: 6200)
- region (dimension ID: 8)

and the dimension values:

- red (dimension value ID: 8021)
- Napa (dimension value ID: 4294967160)

Sort direction

Sort Direction	Configuration	Example base URL (sorted by dimension ID)

Ascending	`<property name="ascending">`
	`  <value>true</value>`
	`</property>`
		`http://localhost/urlformatter_jspref/controller/region-Napa/Wine-red/`

Sort Direction	Configuration	Example base URL (sorted by dimension ID)

Descending	`<property name="ascending">`
	`  <value>false</value>`
	`</property>`
		`http://localhost/urlformatter_jspref/controller/Wine-red/region-Napa/`

Sort method

Sort by	Configuration	Example base URL (sort direction ascending)

Dimension name, case sensitive	`<property name="sortByName">`
	`  <value>true</value>`
	`</property>`
	
	`<property name="sortByDimension">`
	`  <value>true</value>`
	`</property>`
	
	`<property name="ignoreCase">`
	`  <value>false</value>`
	`</property>`
		`http://localhost/urlformatter_jspref/controller/Wine-red/region-Napa/`

Dimension name, case insensitive	`<property name="sortByName">`
	`  <value>true</value>`
	`</property>`
	
	`<property name="sortByDimension">`
	`  <value>true</value>`
	`</property>`
	
	`<property name="ignoreCase">`
	`  <value>true</value>`
	`</property>`
		`http://localhost/urlformatter_jspref/controller/region-Napa/Wine-red/`

Dimension ID	`<property name="sortByName">`
	`  <value>false</value>`
	`</property>`
	
	`<property name="sortByDimension">`
	`  <value>true</value>`
	`</property>`
		`http://localhost/urlformatter_jspref/controller/region-Napa/Wine-red/`

Sort by	Configuration	Example base URL (sort direction ascending)

Dimension value name, case sensitive	`<property name="sortByName"><value>true</value></property>`
	`<property name="sortByDimension"><value>false</value></property>`
	`<property name="ignoreCase"><value>false</value></property>`
		`http://localhost/urlformatter_jspref/controller/region-Napa/Wine-red/`

Dimension value name, case insensitive	`<property name="sortByName"><value>true</value></property>`
	`<property name="sortByDimension"><value>false</value></property>`
	`<property name="ignoreCase"><value>true</value></property>`
		`http://localhost/urlformatter_jspref/controller/region-Napa/Wine-red/`

Dimension value ID	`<property name="sortByName"><value>false</value></property>` `<property name="sortByDimension"><value>false</value></property>`	`http://localhost/urlformatter_jspref/controller/Wine-red/region-Napa/`

Example 1: the following code sample creates a canonicalized URL that sorts by dimension name, case sensitive, in an ascending order:

```
<bean name="navStateCanonicalizer"
 class="com.endeca.soleng.urlformatter.seo.SeoNavStateCanonicalizer">

    <property name="sortByName">
      <value>true</value>
    </property>

    <property name="sortByDimension">
      <value>true</value>
    </property>

    <property name="ascending">
```

```
      <value>true</value>
    </property>

    <property name="ignoreCase">
      <value>false</value>
    </property>

  </bean>
```

The resulting base URL: `http://localhost/urlformatter_jspref/controller/Wine-red/region-Napa/`

Example 2: the following code sample creates a canonicalized URL that sorts by dimension value ID in a descending order:

```
<bean name="navStateCanonicalizer"
 class="com.endeca.soleng.urlformatter.seo.SeoNavStateCanonicalizer">

    <property name="sortByName">
      <value>false</value>
    </property>

    <property name="sortByDimension">
      <value>true</value>
    </property>

    <property name="ascending">
      <value>false</value>
    </property>

  </bean>
```

The resulting base URL: `http://localhost/urlformatter_jspref/controller/region-Napa/Wine-red/`

Note

Canonicalizing the dimension and dimension value names in the misc-path changes the order in which they appear in the path-params section of the URL. For example, if Napa is configured to display before Red in the misc-path, the Napa dimension value ID displays before the Red dimension value ID in the path-params section.

Optimizing URLs for record detail pages

Using the URL optimization classes, you can include dimension names, dimension value names, and record properties in the misc-path of URLs for record detail pages.

Note

For dimensions to display properly in the URL, they must be enabled for display with the record list.

You must create a URL configuration file before completing this procedure.

To optimize URLs for record detail pages:

1. Open your URL configuration file.

2. Create an `erecFormatter` bean to invoke the `com.endeca.soleng.urlformatter.seo.SeoERecFormatter`:

   For example:

```
  <bean id="erecFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoERecFormatter">
  </bean>
```

3. Add an `ERecFormatter` property to your top-level `seoUrlFormatter` bean.

   For example:

```
  <bean id="seoUrlFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

<!-- additional elements deleted from this example --!>

    <property name="ERecFormatter">
      <ref bean="erecFormatter"/>
    </property>

  </bean>
```

4. Add a `useDimensionNameAsKey` property on the `erecFormatter`.

   For example:

```
  <bean id="erecFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoERecFormatter">

    <property name="useDimensionNameAsKey">
      <value>true</value>
    </property>

  </bean>
```

   Setting `useDimensionNameAsKey` to `false` creates a key on the dimension ID numbers.

5. Add a `propertyKeys` property to include record properties in the URLs of record details pages.

   For example:

```
  <bean id="erecFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoERecFormatter">

    <property name="useDimensionNameAsKey">
      <value>true</value>
    </property>

    <property name="propertyKeys">
      <list>
```

```
        <value>P_Name</value>
      </list>
    </property>

  </bean>
```

6. Add a `propertyFormatter` property to format record properties included in the URLs of record details pages.

   For example:

```
  <bean id="erecFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoERecFormatter">

    <property name="useDimensionNameAsKey">
      <value>true</value>
    </property>

    <property name="propertyKeys">
      <list>
        <value>P_Name</value>
      </list>
    </property>

    <property name="propertyFormatter">
      <ref bean="defaultStringFormatterChain"/>
    </property>

  </bean>
```

7. Add a `dimLocationFormatters` property and list each `dimLocationFormatter` bean you plan to define.

   For example:

```
  <bean id="erecFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoERecFormatter">

    <property name="useDimensionNameAsKey">
      <value>true</value>
    </property>

    <property name="dimLocationFormatters">
      <list>
        <ref bean="regionFormatter"/>
        <ref bean="wineryFormatter"/>
        <ref bean="wineTypeFormatter"/>
        <ref bean="vintageFormatter"/>
      </list>
    </property>

    <property name="propertyKeys">
      <list>
        <value>P_Name</value>
      </list>
    </property>

    <property name="propertyFormatter">
```

```
    <ref bean="defaultStringFormatterChain"/>
  </property>

</bean>
```

8. Create a `dimLocationFormatter` for each of the dimensions in the `dimLocationFormatters` list.

   For example:

```
<bean id="regionFormatter"
    class="com.endeca.soleng.urlformatter.seo.SeoDimLocationFormatter">
</bean>
```

   Note

   The sample `urlconfig.xml` file uses the same `dimLocationFormatter` for navigation pages, record detail pages, and aggregate record detail pages. You can choose to create unique `dimLocationFormatters` for each page type.

9. Add the following properties to each `dimLocationFormatter`:

Property	Description
`key`	In the `navStateFormatter` bean, the `useDimensionNameAsKey` property sets the key type. If you set the `useDimensionNameAsKey` to true, then use the dimension name as the value for this property (for example `<value>Region</value>`). If you set the `useDimensionNameAsKey` to false, use the dimension ID number.
`appendRoot`	Specifies whether or not to append root dimension values to the URL. Set to `true` to append root dimension values.
`appendAncestors`	Specifices whether or not to append ancestor dimension values to the URL. Set to `true` to append ancestor dimension values.
`appendDescriptor`	Specifies whether or not to append the selected or descriptor dimension values to the URL. Set to `true` to append selected or descriptor dimension values.
`separator`	Specifies the character used to separate dimension roots, ancestors, and descriptor values.
`rootStringFormatter`	Specifies the bean to format the dimension name. The reference application uses a `defaultStringFormatterChain` bean to invoke the `com.endeca.soleng.urlformatter.seo.StringFormatterCha`

Property	Description
`dimValStringFormatter`	Specifies the bean to format the dimension value names. The reference application uses a `defaultStringFormatterChain` bean to invoke the `com.endeca.soleng.urlformatter.seo.StringFormatterChain`. The examples below also use a `defaultStringFormatterChain` bean.

For example:

```
<bean id="regionFormatter"
    class="com.endeca.soleng.urlformatter.seo.SeoDimLocationFormatter">

  <property name="key">
    <value>Region</value>
  </property>

  <property name="appendRoot">
    <value>false</value>
  </property>

  <property name="appendAncestors">
    <value>false</value>
  </property>

  <property name="appendDescriptor">
    <value>true</value>
  </property>

  <property name="separator">
    <value>-</value>
  </property>

  <property name="rootStringFormatter">
    <ref bean="defaultStringFormatterChain"/>
  </property>

  <property name="dimValStringFormatter">
    <ref bean="defaultStringFormatterChain"/>
  </property>

</bean>
```

10.Save and close the file.

Related links

- Preparing your properties (page 115)
- Preparing your dimensions (page 115)
- Formatting misc-path strings in optimized URLs (page 126)

Optimizing URLs for aggregate record detail pages

Using the URL optimization classes, you can include dimension names, dimension value names, and record properties in the misc-path of URLs for aggregate record detail pages. These are configured separately from the optimizations for navigation pages.

Note

For dimensions to display properly in the URL, they must be enabled for display with the record list.

You must create a URL configuration file before completing this procedure.

To optimize URLs for aggregate record detail pages:

1. Open your URL configuration file.
2. Create an `aggrERecFormatter` bean to invoke the `com.endeca.soleng.urlformatter.seo.SeoAggrERecFormatter` class:

   For example:

```
 <bean id="aggrERecFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoAggrERecFormatter">
 </bean>
```

3. Add an `aggrERecFormatter` property to your top-level `seoUrlFormatter` bean.

   For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

<!-- additional elements deleted from this example --!>

   <property name="aggrERecFormatter">
     <ref bean="aggrERecFormatter"/>
   </property>

 </bean>
```

4. Add a `useDimensionNameAsKey` property on the `aggrERecFormatter`.

   For example:

```
 <bean id="aggrERecFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoAggrERecFormatter">

   <property name="useDimensionNameAsKey">
     <value>true</value>
   </property>
 </bean>
```

   Setting the `useDimensionNameAsKey` to `false` creates a key on the dimension ID numbers.

5. Add a `propertyKeys` property to include record properties in the URLs of record details pages.

   For example:

```
 <bean id="aggrERecFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoAggrERecFormatter">

   <property name="useDimensionNameAsKey">
     <value>true</value>
   </property>

   <property name="propertyKeys">
     <list>
       <value>P_Name</value>
     </list>
   </property>

 </bean>
```

6. Add a `propertyFormatter` property to format record properties included in the URLs of record details pages.

   For example:

```
 <bean id="aggrERecFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoAggrERecFormatter">

   <property name="useDimensionNameAsKey">
     <value>true</value>
   </property>

   <property name="propertyKeys">
     <list>
       <value>P_Name</value>
     </list>
   </property>
   <!-- use default string formatter chain -->

   <property name="propertyFormatter">
     <ref bean="defaultStringFormatterChain"/>
   </property>

 </bean>
```

7. Add a `dimLocationFormatters` property and list each `dimLocationFormatter` bean you plan to define.

   For example:

```
 <bean id="aggrERecFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoAggrERecFormatter">

   <property name="useDimensionNameAsKey">
     <value>true</value>
   </property>

   <property name="dimLocationFormatters">
```

```
    <list>
      <ref bean="regionFormatter"/>
      <ref bean="wineryFormatter"/>
    </list>
  </property>

  <property name="propertyKeys">
    <list>
      <value>P_Name</value>
    </list>
  </property>

  <property name="propertyFormatter">
    <ref bean="defaultStringFormatterChain"/>
  </property>

</bean>
```

Note

The sample `urlconfig.xml` file uses the same `dimLocationFormatter` for navigation pages, record detail pages, and aggregate record detail pages. You can choose to create unique `dimLocationFormatters` for each page type.

8. Create a `dimLocationFormatter` for each of the dimensions in the `dimLocationFormatters` list.

   For example:

```
<bean id="regionFormatter"
    class="com.endeca.soleng.urlformatter.seo.SeoDimLocationFormatter">
</bean>
```

9. Add the following properties to each `dimLocationFormatter`:

Property	Description
`key`	In the `navStateFormatter` bean, the `useDimensionNameAsKey` property sets the key type. If you set the `useDimensionNameAsKey` to true, then use the dimension name as the value for this property (for example `<value>Region</value>`). If you set the `useDimensionNameAsKey` to false, use the dimension ID number.
`appendRoot`	Specifies whether or not to append root dimension values to the URL. Set to `true` to append root dimension values.
`appendAncestors`	Specifices whether or not to append ancestor dimension values to the URL. Set to `true` to append ancestor dimension values.

Property	Description
appendDescriptor	Specifies whether or not to append the selected or descriptor dimension values to the URL. Set to `true` to append selected or descriptor dimension values.
separator	Specifies the character used to separate dimension roots, ancestors, and descriptor values.
rootStringFormatter	Specifies the bean to format the dimension name. The reference application uses a `defaultStringFormatterChain` bean to invoke the `com.endeca.soleng.urlformatter.seo.StringFormatterChain`.
dimValStringFormatter	Specifies the bean to format the dimension value names. The reference application uses a `defaultStringFormatterChain` bean to invoke the `com.endeca.soleng.urlformatter.seo.StringFormatterChain`. The examples below also use a `defaultStringFormatterChain` bean.

For example:

```
<bean id="regionFormatter"
    class="com.endeca.soleng.urlformatter.seo.SeoDimLocationFormatter">

  <property name="key">
    <value>Region</value>
  </property>

  <property name="appendRoot">
    <value>false</value>
  </property>

  <property name="appendAncestors">
    <value>false</value>
  </property>

  <property name="appendDescriptor">
    <value>true</value>
  </property>

  <property name="separator">
    <value>-</value>
  </property>

  <property name="rootStringFormatter">
    <ref bean="defaultStringFormatterChain"/>
  </property>

  <property name="dimValStringFormatter">
    <ref bean="defaultStringFormatterChain"/>
```

```
    </property>
  </bean>
```

10.Save and close the file.

Related links

- Preparing your properties (page 115)
- Preparing your dimensions (page 115)
- Formatting misc-path strings in optimized URLs (page 126)

Configuring the path-param-separator

You can customize the string that displays between the misc-path and the path-params components of URLs.

The sample `urlconfig.xml` file uses an underscore to separate the misc-path from the path-params in URLs. For example: `http://localhost/urlformatter_jspref/controller/Wine-Red-Pinot-Noir/_/N-66w`

You must create a URL configuration file before completing this procedure.

To change the path-param-separator string:

1. Locate the top-level URL formatter bean in your URL configuration file.

   For example:

```
  <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">
  </bean>
```

2. Customize the value of the `pathSeparatorToken` property:

   For example:

```
  <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">
    <property name="pathSeparatorToken">
      <value>separator</value>
    </property>
  </bean>
```

The new URL displays as: `http://localhost/urlformatter_jspref/controller/Wine-Red-Pinot-Noir/separator/N-66w`

About optimizing the path-params and query string

The URL optimization classes provide functionality for encoding path parameters and moving path parameters from the query string into the path-params section of the URL.

Moving parameters out of the query string

In order to create directory-style URLs, you can limit the number of parameters in the query string by configuring a list of parameters to move from the query string and into the path-params section of the URL. For example, the following URL has the parameters N, Ntk, Ntt, and Ntx in the query string:

```
http://localhost/ContentAssemblerRefApp/Content.aspx/Bordeaux?
N=4294966952&fromsearch=false&Ntk=All&Ntt=red&Ntx=mode%2bmatchallpartial
```

Using the URL Optimization API, you can move parameters into the path-params section of the URL. For example, the following URL includes the N and Ntt parameters in the base URL:

```
http://localhost/ContentAssemblerRefApp/Content.aspx/Bordeaux/_/N-4294966952/Ntt-red?
fromsearch=false&Ntk=All&Ntx=mode%2bmatchallpartial
```

Note

To ensure the best possible natural search-engine ranking, it is recommended that you limit the number of parameters you include in the path-params section.

Encoding parameters

In order to shorten URLs, the Assembler API allows base-36 encoding of parameters.

For example, the following URL for Region > Napa contains the dimension value ID for Napa (4294966952):

```
http://localhost/ContentAssemblerRefApp/Content.aspx/Napa/_/N-4294966952
```

By base-36 encoding the N parameter, you can shorten the URL:

```
http://localhost/ContentAssemblerRefApp/Content.aspx/Napa/_/N-1z141pk
```

Note

Only the numeric parameters can be encoded:

- `N`
- `Ne`
- `An`
- `Dn`

Removing session-scope parameters

In order to simplify the URLs, session-scope parameters should be removed from the URL string and stored as session objects. This might include any parameters that do not change value during the session, such as the session ID or MDEX host and port values.

Passing non-parameters to the API

You can add non-parameters to URLs by passing them through the API.

Moving parameters out of the query string

In order to create directory-style URLs, you can limit the number of parameters in the query string by configuring a list of parameters to move from the query string and into the path-params section of the URL.

You must create a URL configuration file before completing this procedure.

To move parameters out of the query string and into the path-params section of the URL:

1. In your URL configuration file, locate the top-level URL formatter.

   For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

   <property name="defaultEncoding">
     <value>UTF-8</value>
   </property>

   <property name="pathSeparatorToken">
     <value>_</value>
   </property>

   <!-- additional elements deleted from this example --!>

 </bean>
```

2. Add a `pathParamKeys` property.

   For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

   <property name="pathParamKeys">
   </property>

 </bean>
```

3. Add a `list` attribute containing all of the parameters you want moved from the query string.

   For example:

```
 <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">

   <property name="pathParamKeys">
     <list>
       <value>R</value>
       <value>A</value>
       <value>An</value>
     </list>
   </property>

 </bean>
```

Encoding parameters

You can apply base-36 encoding to numeric parameters.

You must create a URL configuration file before completing this procedure.

Only the numeric parameters can be encoded:

- `N`
- `Ne`
- `An`
- `Dn`

The following procedure provides instructions for applying base-36 encoding to the `An` parameter. You can apply base-36 encoding to any numeric parameter, but each parameter requires a separately configured `paramEncoder` bean.

To encode numeric parameters:

1. Open your URL configuration file.
2. Create a `paramEncoder` bean to invoke the `com.endeca.soleng.urlformatter.seo.SeoNavStateEncoder`:

   For example:

```
  <bean name="An-paramEncoder"
 class="com.endeca.soleng.urlformatter.seo.SeoNavStateEncoder">
  </bean>
```

3. Add a `paramKey` property to specify which numeric parameter to encode.

   For example:

```
  <bean name="An-paramEncoder"
 class="com.endeca.soleng.urlformatter.seo.SeoNavStateEncoder">
    <property name="paramKey">
      <value>An</value>
    </property>
  </bean>
```

4. Repeat steps one and two for each parameter you want to encode.
5. Locate the top-level URL formatter bean in your URL configuration file.

   For example:

```
  <bean id="seoUrlFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">
  </bean>
```

6. Add a `urlParamEncoders` property:

```
  <bean id="seoUrlFormatter"
 class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">
```

```
    <property name="urlParamEncoders">
    </property>
  </bean>
```

7. Add a `list` attribute and specify each of the parameter encoder beans.

   For example:

```
  <bean id="seoUrlFormatter"
class="com.endeca.soleng.urlformatter.seo.SeoUrlFormatter">
    <property name="urlParamEncoders">
      <list>
        <ref bean="N-paramEncoder"/>
        <ref bean="Ne-paramEncoder"/>
        <ref bean="An-paramEncoder"/>
      </list>
    </property>
  </bean>
```

8. Save and close the file.

Removing session-scope parameters

In order to simplify the URLs, session-scope parameters should be removed from the URL string and stored as session objects.

This might include any parameters that do not change value during the session, such as the session ID or MDEX host and port values. For example, the following URL contains information about the MDEX host and port:

`http://localhost:8888/endeca_jspref/controller.jsp?N=0&eneHost=localhost&enePort=15002`

You can remove the MDEX host and port values from the URL and store them as session objects. The resulting URL is simplified:

`http://localhost:8888/endeca_jspref/controller.jsp`

The following procedure provides instructions for removing the MDEX host and port values from the URL, but this procedure can be adapted as necessary to remove other session-scope parameters.

To remove the MDEX host and port values from the URL and store them as session attribute values:

1. To set the attribute, use the following code:

```
session.setAttribute("eneHost", eneHost);
```

2. To retrieve the attribute value, use the following code:

```
eneHost = (String)session.getAttribute("eneHost");
```

About passing non-parameters to the API

You can add non-parameters to URLs by passing them through the API.

For example, you could add information about how many records per page should display in each results set:

In the reference application's `controller.jsp` file, find the following section:

```
UrlState baseUrlState = urlFormatter.parseRequest(request);

ENEQuery usq = queryBuilder.buildQuery(baseUrlState);
```

and add code similar to the following:

```
baseUrlState.setParam("records_per_page", "25");
```

Note

Oracle recommends limiting the number of parameters that display in URLs. It is recommended that session-scope parameters be removed from the URL and stored as session objects.

Using the URL configuration file with your application

Before you can create optimized URLs with your own application, you need to include the URL configuration file in your application's classpath.

To use the URL configuration file with your application:

1. Stop the HTTP service.
2. Locate your URL configuration file.
3. Copy the URL configuration file into the `WEB-INF` subdirectory of your Web application directory.

   For example: `C:\Endeca\ToolsAndFrameworks\<version>\reference\discover-electronics-authoring\WEB-INF`
4. Start the HTTP service.

To verify that the URL configurations are working properly, open a Web browser and navigate to your Web application. Check that the URLs display as you configured them with the URL configuration file.

Related links

- Creating a URL configuration file (page 122)
- Creating a URL configuration file (page 122)

Integrating with the Sitemap Generator

The Sitemap Generator creates an index of your Web site based on information stored in your MDEX Engine, not information stored on your application server. Because of this, you need to ensure that the URLs produced

by the Sitemap Generator match the URLs in your application. To make certain that the URLs match, you need to configure the Sitemap Generator's `urlconfig.xml` file to make the same customizations to URLs as those configured for the Assembler API.

Related links

- *Optimizing Application URLs* (page 99)
- The Sitemap Generator urlconfig.xml file (page 151)
- Using the URL configuration file with the Sitemap Generator (page 151)

The Sitemap Generator urlconfig.xml file

The Sitemap Generator uses a URL configuration file that must mirror your URL configurations in order to output a sitemap that matches your Web application.

The Sitemap Generator creates a site map by issuing a single bulk query against the MDEX Engine to retrieve the necessary record, dimension, and dimension value data. It uses this information to build an index of pages. The formatting of the URLs it creates is controlled by the `urlconfig.xml` file located in the `conf` subdirectory of your Sitemap Generator installation directory. For example: `C:\Endeca\SEM\SitemapGenerator\<version>\conf`

To ensure that the URLs in the sitemap are consistent with the URLs produced by the Assembler, configuration in the URL configuration file must correspond to the Sitemap Generator's `urlconfig.xml` file.

Because the `urlconfig.xml` file included with the Sitemap Generator uses the same format as the sample `urlconfig.xml` file for the Assembler API, you can copy the `urlconfig.xml` file for sitemap generation.

Using the URL configuration file with the Sitemap Generator

You can use the same `urlconfig.xml` file you created for URL optimization as the URL configuration file for sitemap generation.

To use the URL configuration file with the Sitemap Generator:

1. Open the `conf.xml` file located in the `conf` subdirectory of your Sitemap Generator installation directory.

   For example: `ToolsAndFrameworks\<version>\sitemap_generator\conf`

2. Locate the `URL_FORMAT_FILE`:

   For example:

   ```
   <URL_FORMAT_FILE>urlconfig.xml</URL_FORMAT_FILE>
   ```

3. Edit the `<URL_FORMAT_FILE>` value so that it points to the `urlconfig.xml` file you created with the URL Optimization API.

   For example:

```
<URL_FORMAT_FILE>C:\Endeca\ToolsAndFrameworks\<version>\reference\discover-
electronics-authoring\WEB-INF\urlconfig.xml</URL_FORMAT_FILE>
```

4. Save and close the `conf.xml` file.

Related links

- Creating a URL configuration file (page 122)
- About the URL configuration file (page 121)

5 Extending the Assembler

This part provides information on extending the Assembler.

Extending and Developing Cartridges

If your application requires functionality that is not covered by the core cartridges and navigation cartridges included in Tools and Frameworks, you can extend the existing cartridges or develop your own.

Related links

Cartridge Basics

This section introduces the basic components of a cartridge by examining how they work together in a "Hello, World" example cartridge.

First steps with a new cartridge

This section describes how to define a new cartridge and use Workbench to configure it to appear on a page.

To create and configure a basic "Hello, World" cartridge, follow these steps:

1. Navigate to the templates directory of your application, and create a subdirectory named "HelloWorld." This directory name is the template ID for your template.

   For example: `C:\Endeca\apps\Discover\config\import\templates\HelloWorld`.

2. Create a cartridge template JSON file.

   a. Open a new plain text file.

   b. Type or copy the following into the contents of the file:

```
{
    "ecr:type": "template",
    "@group": "SecondaryContent",
    "@description": "A sample cartridge that can display a simple message",
    "@thumbnailUrl": "thumbnail.jpg",
    "defaultContentItem": {
        "@name": "Hello cartridge",
        "message": "",
        "messageColor": ""
     },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
      {
                "editor": "GroupLabel",
                "label": "${group.contents.label}"
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "message",
                "label": "Message"
                },
            {
                "editor": "editors/StringEditor",
                "propertyName": "messageColor",
                "label": "Color"
            }
        ]
    },
    "typeInfo": {
        "message": {"@propertyType": "String"},
        "messageColor": {"@propertyType": "String"}
    }
}
```

   c. Save the file with the name `_.json` in the `HelloWorld` directory of your Discover Electronics application, for example: `C:\Endeca\apps\Discover\config\import\templates\HelloWorld`.

3. Upload the template to Workbench.

   a. Open a command prompt and navigate to the `control` directory of your deployed application, for example, `C:\Endeca\apps\Discover\control`.

   b. Run the `set_templates` command.

```
C:\Endeca\apps\Discover\control>set_templates.bat
Removing existing cartridge templates for Discover
Setting new cartridge templates for Discover
Finished setting templates

C:\Endeca\apps\Discover\control>
```

4. Add the cartridge to a page.
   a. Open Workbench in a Web browser.

      The default URL for Workbench is `http://<workbench-host>:8006`.

   b. From the launch page, select Experience Manager.
   c. In the tree on the left, select Search and Navigation Pages under the Content section, then select the Default Page.
   d. In the Edit Pane on the right, select the right column section from the Content Tree.
   e. Click Add.

      The cartridge selector dialog displays.

   f. Select the Hello cartridge and click OK.
   g. Select the new Hello cartridge from the Content Tree on the left and configure the color as #FF00000.
   h. Click Save Changes.
5. Try to view the cartridge in the Discover Electronics application.
   a. In a Web browser, navigate to `http://<workbench-host>:8006/discover-authoring/`.

      An error displays because we have not yet created a renderer for the Hello cartridge.

The following shows the JSON representation of the page with most of the tree collapsed, highlighting the data for the cartridge that we just added.

```
{
    "workflowState": "ACTIVE",
    "ecr:lastModifiedBy": "admin",
    "ecr:lastModified": "2016-09-12T13:38:32.258+05:30",
    "priority": 100,
    "ecr:createDate": "2016-09-12T13:38:32.258+05:30",
    "ecr:type": "content-item",
    "contentItem": {
        "@name": "Three-Column Page",
        "metaKeywords": "camera cameras electronics",
        "@type": "ThreeColumnPage",
        "title": "Discover Electronics",
        "metaDescription": "Oracle Commerce reference application.",
        "redirectGroup": "",
        "headerContent":{...},
        "leftContent": {...},
        "mainContent": {...},
        "rightContent":[
```

```
                { … },
                { … },
                {
                     "@name": "Hello World",
                         "@type": "HelloWorld",
                         "message": "Hello, World",
                         "messageColor": "#FF0000"
                }
            ]
        }
    ],
    …

}
```

In the next section, we'll create a simple renderer that displays the message based on the values configured in Experience Manager.

Adding a basic renderer

While there is no one way to write rendering code for an application, in this example we'll write a simple JSP renderer for our basic cartridge.

To write a basic "Hello, World" renderer:

1. Create a new JSP page and type or copy the following:

```
<%@page language="java" pageEncoding="UTF-8"
  contentType="text/html;charset=UTF-8"%>

<%@include file="/WEB-INF/views/include.jsp"%>
<div style="border-style: dotted; border-width: 1px;
  border-color: #999999; padding: 10px 10px">
  <div style="font-size: 150%;
    color: ${component.messageColor}">${component.message}
  </div>
</div>
```

2. Save the renderer to `discover-electronics-authoring/WEB-INF/views/desktop/Hello/Hello.jsp`.
3. Refresh the Discover Electronics authoring application at `http://<workbench-host>:8006/discover-authoring/` to see the result.

Elements of the example cartridge

As we have seen, the high-level workflow for creating a basic cartridge is:

1. Create a cartridge template and upload it to Workbench.

2. Use Experience Manager to create and configure and instance of the cartridge.

3. Add a renderer to the front-end application.

Step 2 is necessary during development in order to have a cartridge instance with which to test. However, once the cartridge is complete, the business user is typically responsible for creating and maintaining cartridge instances in Experience Manager.

In the following sections, we'll describe each of these elements of the cartridge in greater detail.

The cartridge template

The template defines the configuration that the business user can specify in Workbench using Experience Manager.

The template contains these main sections: the `defaultContentItem` element, the `EditorPanel` element, and the `typeInfo` element..

The *content item* is a core concept in Assembler applications that can represent both the configuration model for a cartridge and the response model that the Assembler returns to the client application. A content item is a map of properties, or key-value pairs. End users know content items as **rules**. The `defaultContentItem` element in the template contains properties with default values,

The `EditorPanel` defines the interface that can be used in Experience Manager to configure the properties of the content item. The *editor panel* is composed of a number of *editors*. The editors provide the UI controls that the business user can use to specify the property values for a particular instance of that cartridge.

The `typeInfo` element defines property type information.

In our example template, we defined two string properties named `message` and `messageColor` and attached two simple string editors to those properties.

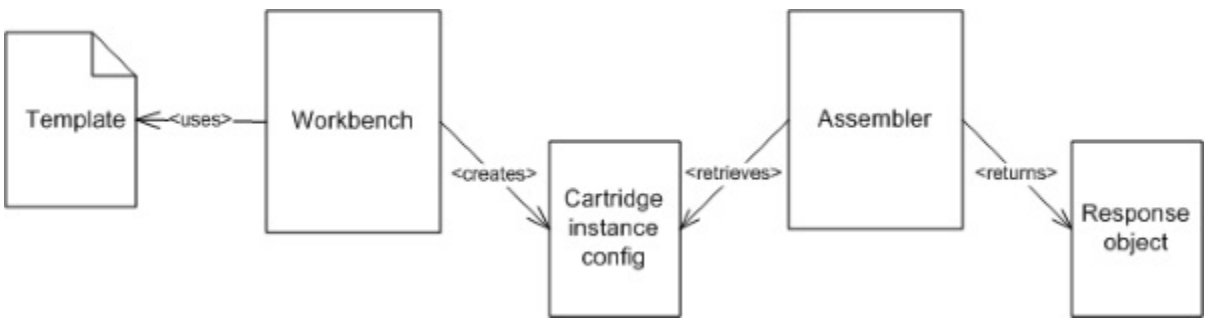
For more information about creating and managing cartridge templates, see About creating templates (page 27).

The cartridge instance configuration

The business user creates and configures instances of cartridges in Experience Manager based on a template. During cartridge development you need to create at least one instance of a cartridge for testing.

Experience Manager writes this cartridge instance configuration as JSON.

The Assembler retrieves this configuration at runtime and uses it to build the response model that it returns to the client application.



For any given cartridge, the default behavior is for the Assembler to do no processing on the configuration and simply return the configuration content item as a map of properties. That is, the response object is the same as the configuration object unless specific processing logic is defined in the Assembler for that cartridge.

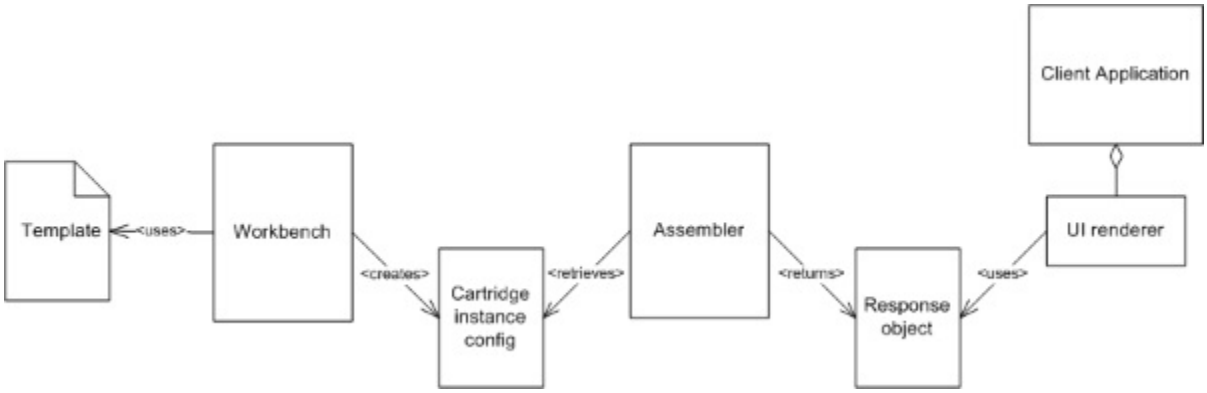
The cartridge renderer

As a best practice, the client application should be composed of modular rendering components, each corresponding to a particular cartridge.

Recall the contents of the Assembler response object corresponding to the example cartridge:

```
"@name": "Hello World",
 "@type": "HelloWorld",
 "message": "Hello, World",
 "messageColor": "#FF0000"
```

For each cartridge, the `@type` of the response object corresponds to the folder name of the template that was used to create it. The Discover Electronics application uses this type to identify the appropriate renderer to use for this content item.

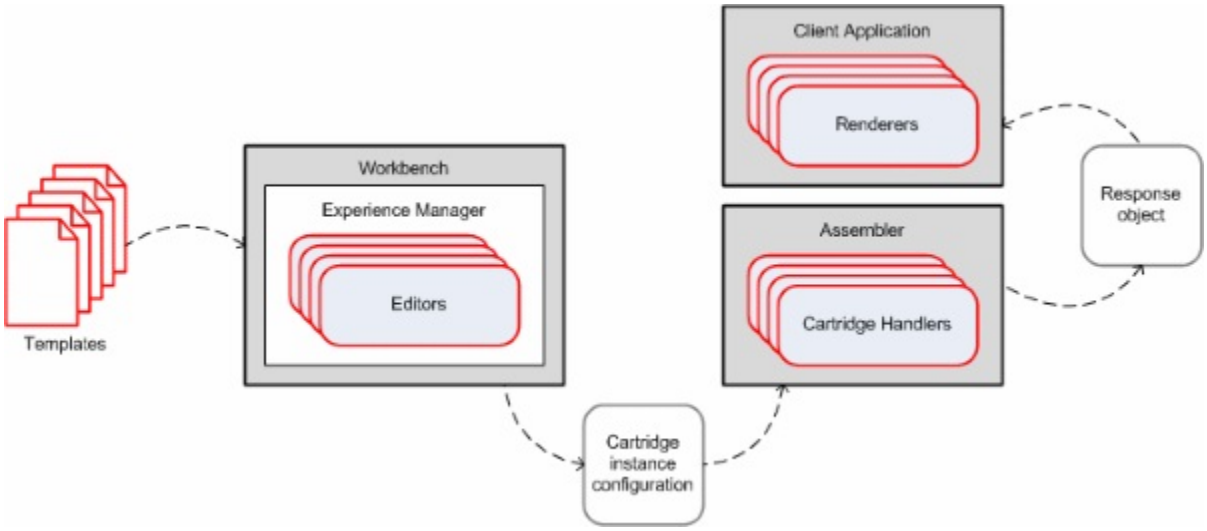


The logic for mapping response objects to the appropriate renderer is contained in `include.tag` in the reference application.

Overview of cartridge extension points

Cartridges are made up of several components that may be customized for specific purposes.

The following diagram shows the parts of a cartridge and where they fit within the overall architecture:



The cartridge *template* defines the configuration options that are available to the business user in Workbench. The Experience Manager interface is composed of *editors* that provide UI controls for specifying property values. Experience Manager produces the *cartridge instance configuration* that is consumed by the Assembler. During the processing of a query, the Assembler may invoke *cartridge handlers* that define specific processing logic for particular cartridges. Using these cartridge handlers, the Assembler produces the *response object* that it returns to the client application. Typically, the client application includes modular *renderers* that are intended to handle a particular cartridge.

We created a basic template and renderer in the example cartridge. We also inspected the cartridge instance configuration generated in Workbench and the response returned by the Assembler. In the example cartridge, both the configuration and the response model were generic content items that are simple maps of properties. Many of the core cartridges have strongly typed configuration models and response objects associated with them that extend from the basic content item. This makes it easier to understand the expected input to and output from the core cartridge handlers, and also enables reuse of the models for the core cartridges. Strongly typed configuration beans also make it possible to configure default values for cartridge properties via Spring. Creating strongly typed model objects for the Assembler configuration and response is not required when developing cartridges.

In the following sections, we discuss how to customize the Experience Manager interface using editors, and how to define custom processing logic in the Assembler using cartridge handlers.

Customizing the Experience Manager interface

Experience Manager provides a set of standard editors that you can use in cartridge templates as well as the ability to develop custom editors.

Adding embedded user assistance to a cartridge

You can provide embedded assistance for the business user in the Experience Manager interface by specifying it in the cartridge template.

In our example cartridge, we provided two simple text fields for the business user to enter a message and the desired color. This user interface makes it unclear what values are allowed or expected for those fields. The template schema for configuring editors allows you to supply a short descriptive label for each field, but

sometimes additional context can be helpful. For such cases, you can use the `bottomLabel` attribute to provide further information.

To add additional guidance for the business user to the example cartridge:

1. Open the template file (`HelloWorld\_.json`) that you previously created.

2. Add a `bottomLabel` attribute to each editor in the `EditorPanel`, as in the example below:

```
"editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.contents.label}"
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "message",
                "label": "Message",
                "bottomLabel": "Enter a message to display. HTML is allowed."
                },
            {
                "editor": "editors/StringEditor",
                "propertyName": "messageColor",
                "label": "Color"
            }
        ]
    },
```

   This additional label text can be configured for all editors built using the Experience Manager SDK, including all the standard editors. For the full content of the updated template, see the example below. If your implementation uses multiple locales, see About multiple locales (page 38) for information about localizing strings.

3. Save and close the template.

4. Upload the template by running the `set_templates` script.

The following shows the complete content of the updated template:

```
{
    "ecr:type": "template",
    "@group": "SecondaryContent",
    "@description": "A sample cartridge that can display a simple message",
    "@thumbnailUrl": "thumbnail.jpg",
    "defaultContentItem": {
        "@name": "Hello cartridge",
        "message": "",
        "messageColor": ""
     },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.contents.label}"
```

```
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "message",
                "label": "Message",
                "bottomLabel": "Enter a message to display. HTML is allowed."
                },
            {
                "editor": "editors/StringEditor",
                "propertyName": "messageColor",
                "label": "Color"
            }
        ]
    },
    "typeInfo": {
        "message": {"@propertyType": "String"},
        "messageColor": {"@propertyType": "String"}
    }
}
```

For more information about label options for Experience Manager editors, see the Editor label configuration reference (page 212).

Using the core Experience Manager editors

Experience Manager provides a set of editors that can configure primitive property types as well as Oracle Commerce-specific features. You specify which editor to use to configure which properties in the `<EditorPanel>` portion of the template.

Even with additional user assistance text, asking the business user to type a hex code into a text field does not provide a very user-friendly experience. One of the standard editors included with Experience Manager is a combo box that can be used to specify a set of valid values for a string property. In this example, we provide a set of colors from which the business user can choose. This not only relieves the business user from typing in a hex code, but it can also ensure that the selected color matches the site's color scheme.

To update the example cartridge to use a combo box editor:

1. Open the template file, `HelloWorld\_.json`, that you previously created.
2. Replace the string editor configuration for the `messageColor` property with the following:

```
"editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.contents.label}"
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "message",
                "label": "Message",
                "bottomLabel": "Enter a message to display. HTML is allowed."
            },
            {
                "editor": "editors/ChoiceEditor",
                "label": "Color",
```

```
                "propertyName": "messageColor",
                "choices": [
                    {
                        "label": "Red",
                        "value": "#FF0000"
                    },
                    {
                        "label": "Green",
                        "value": "#00FF00"
                    },
                    {
                        "label": "Blue",
                        "value": "#0000FF"
                    }
                ]
            }
        ]
    }
```

For the full content of the updated template, see the example below.

3. Upload the template by running the `set_templates` script.

Depending on the option that the business user selects, the value of the property is set to the appropriate hex code. You can change the value and refresh the application to see the change.

The following shows the complete content of the updated template:

```
{
    "ecr:type": "template",
    "@group": "SecondaryContent",
    "@description": "A sample cartridge that can display a simple message",
    "@thumbnailUrl": "thumbnail.jpg",
    "defaultContentItem": {
        "@name": "Hello cartridge",
        "message": "",
        "messageColor": ""
     },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.contents.label}"
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "message",
                "label": "Message",
                "bottomLabel": "Enter a message to display. HTML is allowed."
                },
            {
                "editor": "editors/ChoiceEditor",
                "label": "Color",
                "propertyName": "messageColor",
                "choices": [
                    {
                        "label": "Red",
                        "value": "#FF0000"
```

```
                    },
                    {
                        "label": "Green",
                        "value": "#00FF00"
                    },
                    {
                        "label": "Blue",
                        "value": "#0000FF"
                    }
                ]
            }
        ]
    },
    "typeInfo": {
        "message": {"@propertyType": "String"},
        "messageColor": {"@propertyType": "String"}
    }
}
```

For more information about the standard Experience Manager editors and their configuration, refer to the *Template Property and Editor Reference* (page 209).

About custom editors

If none of the standard editors meet your needs, you can develop your own editors using the Experience Manager Editor SDK.

You may want to develop an editor if:

- You want to allow the business user to configure more advanced properties such as lists or maps of key-value pairs.
- You want to provide a more advanced interface for the business user, such as a list that enables drag-and-drop.
- You want the editor options to be populated dynamically from an external system rather than configured in the template.
- You want the behavior of one editor or UI control to be linked to the state of another.

For more information about the Experience Manager Editor SDK and developing Experience Manager editors, see Developing Custom Editors (page 191).

About Cartridge Handlers and the Assembler

This section provides an overview of the Assembler. It describes the Assembler processing model and core interfaces as well as how to implement a cartridge handler.

About the CartridgeHandler interface

A cartridge handler takes a content item representing the cartridge instance configuration as input and is responsible for returning the response as a content item.

The `CartridgeHandler` interface defines three methods: `initialize()`, `preprocess()`, and `process()`.

The `initialize()` method provides an opportunity for the cartridge handler to augment the cartridge instance configuration specified in Experience Manager with configuration from other sources. This can be used

to define default behavior for a cartridge in the case where there is no Experience Manager configuration, or to override the Experience Manager configuration for the current query. The `initialize()` method should return a content item containing the complete configuration for the cartridge from all possible configuration sources. This augmented configuration item can either be the mutated input content item or a new instance of `ContentItem`, and is used as input to both the `preprocess()` and `process()` methods.

Because the `preprocess()` method is called on all cartridges before `process()` is called on any cartridges, it provides an opportunity to coordinate processing between cartridges. Many of the core cartridges make use of this mechanism in order to consoldiate queries to an MDEX Engine among several cartridges during the course of a single assembly cycle.

The `process()` method is responsible for returning a `ContentItem` that represents the cartridge response.

A cartridge handler need not define any behavior for `initialize()` or `preprocess()`. The `AbstractCartridgeHandler` class exists to simplify the task of implementing the `CartridgeHandler` interface. It provides empty implementations for `initialize()` and `preprocess()`. Subclasses of `AbstractCartridgeHandler` need only implement the `process()` method to return the response object. They can optionally override the `initialize()` and `preprocess()` methods.

About initializing the cartridge configuration

The `initialize()` phase in the cartridge processing life cycle enables the cartridge handler to synthesize the complete configuration for the cartridge from several sources.

The configuration content item that is passed in to the assembly process is the cartridge instance configuration from Experience Manager, however, any given cartridge may also have other configuration sources.

In a typical scenario, a cartridge has some default behavior that can be specified as a property value in a Spring context file. A business user can specify a value for a specific instance of a cartridge using Experience Manager. The site visitor may also have the ability to override either the default or the cartridge instance setting from the client application. For example, in the Results List cartridge, the default value for records per page is 10. The business user can set this value to 25 in Experience Manager, and the site visitor can choose to display 50 records by selecting the appropriate option on the site.

The Assembler API includes the `ConfigInitializer` utility class with the method `initialize()`. The default implementation of `initialize()` layers the cartridge configuration in the following order (from lowest to highest):

1. **Default configuration**, typically defined in the Spring configuration for the cartridge handler
2. **Cartridge instance configuration**, typically created in Experience Manager and passed in as the configuration content item
3. **Request-based configuration** parsed from the HTTP request parameters, using the `RequestParamMarshaller` helper class

The `ConfigInitializer` class also provides methods for additional layering of configuration. Subclasses can override `ConfigInitializer` to define custom layering behavior, for example, to incorporate configuration saved in the session state.

About the NavigationCartridgeHandler class

The core cartridges that make queries to an MDEX Engine use cartridge handlers that extend from `NavigationCartridgeHandler`.

The `NavigationCartridgeHandler` makes use of the two-pass Assembler processing model to consolidate MDEX Engine queries across cartridges.

In the `preprocess()` phase, the cartridge handler calls `createMdexRequest()` but does not execute the request. In subsequent calls to `createMdexRequest()` by other handlers, the MDEX resource broker determines whether the new request can be consolidated with an existing request in order to minimize the number of queries to the MDEX Engine for a single assembly cycle.

During the `process()` phase, the handler calls `executeMdexRequest()` to retrieve the results. The actual query to the MDEX Engine is executed when the first handler in the assembly cycle calls `executeMdexRequest()` and the results are cached for all subsequent handlers that try to execute the same request.

You can use a similar approach if you have multiple cartridges that need to make requests to the same external resource and can achieve efficiencies by consolidating requests across cartridges.

For further information about the `NavigationCartridgeHandler` class, refer to the *Assembler API Reference (Javadoc)*.

Implementing a cartridge handler

You add a cartridge handler by writing a Java class that implements the `CartridgeHandler` interface and configuring the Assembler to use the new handler in the Spring context file.

In this example, we update our "Hello, World" cartridge to do some simple string manipulation on the message that was specified in Experience Manager. Because this cartridge does not use any configuration other than the cartridge instance configuration from Experience Manager and does not need to do any preprocessing, we can extend `AbstractCartridgeHandler`.

To add a cartridge handler to the example cartridge:

1. Create a new Java class in the package `com.endeca.sample.cartridges` and type or copy the following:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.AbstractCartridgeHandler;
import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.assembler.ContentItem;

public class UppercaseCartridgeHandler extends AbstractCartridgeHandler
{
  //========================================================================
  // The cartridge handler 'process' method
  public ContentItem process(ContentItem pContentItem) throws
CartridgeHandlerException
  {
    // Get the message property off of the content item.
    final String message = (String) pContentItem.get("message");
    // If the message is non-null, uppercase it.
    if (null != message) {
       pContentItem.put("message", message.toUpperCase());
    }
    return pContentItem;
  }
}
```

2. Compile the cartridge handler and add the compiled class to your application, for example, by saving it in `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring\WEB-INF\classes`.

3. Configure the Assembler to use the `UppercaseCartridgeHandler` for the Hello cartridge.

a. Navigate to the `WEB-INF` directory of your application, for example, `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring\WEB-INF`.

b. Open the `assembler-context.xml` file.

c. Add the following in the `CARTRIDGE HANDLERS` section:

```
<!--
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ BEAN: CartridgeHandler_Hello
-->
<bean id="CartridgeHandler_Hello"
class="com.endeca.sample.cartridges.UppercaseCartridgeHandler"
scope="prototype" />
```

d. Save and close the file.

4. Restart the Tools Service.

5. Refresh the authoring instance of the application.

The message now appears in all-uppercase letters.

Cartridge handler development scenarios

You should write a cartridge handler in cases where you need to perform some processing on the cartridge instance configuration before sending the response to the client application.

It is always possible to do processing in the client application, but encapsulating the business logic in an extension to the Assembler provides several advantages:

- It makes the rendering code cleaner and easier to maintain.
- It centralizes the processing in one place so that the results can be consumed by multiple client applications including across multiple channels such as desktop, mobile, and others.
- It provides an opportunity for coordinating processing across multiple cartridges before returning the response to the client application.

Depending on what the cartridge handler needs to accomplish, your implementation approach may vary. Cartridge handlers must always implement the `process()` method to return the response model.

Scenario	Implementation approach	Example cartridge
Update properties from the cartridge instance configuration in place (data cleansing or manipulation scenario)	Extend `AbstractCartridgeHandler` and override `process()` to update the property values in the input content item	"Hello, World" with UppercaseCartridgeHandler
Use information from the cartridge instance configuration to query an external resource for the information to display	Extend `AbstractCartridgeHandler` and override `process()` to query the resource and insert the results in the output content item	RSS Feed cartridge

Scenario	Implementation approach	Example cartridge
Query an external resource, consolidating queries between cartridges within a single assembly cycle for improved performance	Take advantage of the two-pass assembly model with `preprocess()` and `process()` and implement a resource broker that can consolidate queries and manage their execution	NavigationCartridgeHandler
Augment the results from a core cartridge with additional information from a non-MDEX resource	Extend the core cartridge and override `process()` to query the resource and add additional properties to the MDEX query results before returning the response	Custom Record Details with availability information
Customize a core cartridge to modify the MDEX Engine query parameters	Extend the core cartridge and override either `initialize()` or `preprocess()` to modify the query before it is executed	Custom Results List with recommendations
Combine multiple sources of cartridge configuration before processing results	Extend `AbstractCartridgeHandler` or implement the `CartridgeHandler` interface and override `initialize()`, making use of the `ConfigInitializer` and `RequestParamMarshaller` helper classes to generate the complete configuration model	"Hello, World" with layered color configuration

About using event listeners to extend the navigation cartridges

You can use the Assembler eventing framework as an extension point for navigation cartridges in cases where extending an existing cartridge handler is insufficient.

If you are making modifications to the navigation cartridges, you can trigger processing logic based on Assembler events instead of subclassing the core cartridge handlers.

Using an event listener instead of extending a cartridge handler introduces the following considerations:

- Unlike extending a cartridge handler, logic included in an event listener is evaluated for every cartridge handler.
- Event listeners do not have access to the current Assembler request or to the navigation state.
- Event listeners must be thread safe.

Related links

- Assembler event framework reference (page 12)

Creating an event listener

The Assembler provides an empty implementation of the `AssemblerEventListener`, `AssemblerEventAdapter`. You can extend this implementation to create a listener that triggers on an Assembler event.

To create an event listener:

1. Create a new Java class that extends the `AssemblerEventAdapter`.

   For example:

```
public class ResultsListListener extends AssemblerEventAdapter {
}
```

2. Override the methods that correspond to the events for which you wish to trigger custom processing logic:

```
public class ResultsListListener extends AssemblerEventAdapter {
    @Override
    public void cartridgePreprocessStarting(AssemblerEvent event){
    ...
    }

    @Override
    public void cartridgeProcessComplete(AssemblerEvent event){
    ...
    }
}
```

   For a list of Assembler events, see the Assembler event framework reference (page 12) or refer to the *Assembler API Reference (Javadoc)*.

3. Add conditional logic to restrict processing to a specific cartridge handler:

```
public class ResultsListListener extends AssemblerEventAdapter {
    ...

    @Override
    public void cartridgeProcessComplete(AssemblerEvent event){
        if(event.getContentItem() != null &&
 "ResultsList".equals(event.getContentItem().getType()){

            ...
        }
    }
}
```

4. Add processing logic.

   The example below prefixes the `max_price` property on a record with a dollar sign:

```
public class ResultsListListener extends AssemblerEventAdapter {

    ...

    @Override
    public void cartridgeProcessComplete(AssemblerEvent event){
        if(event.getContentItem() != null &&
 "ResultsList".equals(event.getContentItem().getType()){
            ResultsList resultsList = (ResultsList) event.getContentItem();
            for(Record record : resultsList.getRecords()){
                Attribute price = record.getAttributes().get("product.max_price");
```

```
                if(price != null){
                    for(int i = 0 ; i < price.size(); i++){
                        price.set(i, "$" + price.get(i).toString());
                    }
                }
            }
        }
    }
```

After creating a new listener, you must register it by including it in the list of listeners for the `assemblerFactory` object.

About registering an event listener

You must register all event listeners with the `AssemblerFactory` object.

The `AssemblerFactory` takes event listeners as constructor arguments. These listeners are instantiated with each `Assembler` object created by the factory class.

Optionally, you may also choose to use the `Assembler.addAssemblerEventListener()` method to add a listener for a single assembly request.

Example 5.1. Example

The example below uses the `ResultsListListener` defined in the previous topic, registered in the Discover Electronics reference application.

The reference application uses the Assembler context file to configure global application properties. The configuration bean for the `AssemblerFactory` includes a list of listeners as constructor arguments:

```
<bean id="assemblerFactory"
 class="com.endeca.infront.assembler.spring.SpringAssemblerFactory"
    scope="singleton">
    <constructor-arg>
        ...
    </constructor-arg>
    <constructor-arg>
        <!-- List of listeners registered in the assembler -->
        <list>
            <bean class="com.endeca.infront.ResultsListListener" />
            <bean class="com.endeca.infront.logger.SLF4JAssemblerEventLogger" />
            <bean
 class="com.endeca.infront.assembler.event.request.ContentItemAugmentAdapter">
                <constructor-arg ref="springUtility"/>
            </bean>
            ...
        </list>
    </constructor-arg>
</bean>
```

Sample Cartridges

This section contains sample cartridge customizations that demonstrate how to use the various cartridge extension mechanisms to address different use cases.

About using the sample cartridges

The sample cartridges are intended to demonstrate the cartridge extension mechanisms and provide a model for your own cartridge customizations.

The sample code provided is written to be generic and easy to follow, rather than production-quality code. Oracle recommends that you follow a few best practices when working with the examples:

- Set up a new instance of the Discover Electronics application to use as a sandbox for deploying the sample cartridges. This isolates the samples from the out-of-the-box configuration for Discover Electronics as well as your own application.
- Within your sandbox application, create a separate Spring context file for the custom cartridge handlers described in this guide.
- When copying and pasting examples from this guide, pay attention to the end-of-line marker (¬) that indicates that a long line of text has been wrapped. Ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

The steps described for creating and deploying the components of the sample cartridges correspond to the steps described in previous sections for the "Hello, World" cartridge. If you need additional information to complete a particular step in deploying one of the sample cartridges, refer to the more detailed procedures for the "Hello, World" example.

Setting up a test application based on Discover Electronics

Oracle recommends that you use a test application to test the sample cartridges instead of deploying them in Discover Electronics or your own application.

Because a test application is for development use only, we do not need to deploy a live instance of the application.

To deploy a copy of Discover Electronics to use as a test for the sample cartridges:

1. Deploy a new test application using the Deployment Template.
   a. From a command prompt, navigate to `%ENDECA_TOOLS_ROOT%\deployment_template\bin` (on Windows) or `$ENDECA_TOOLS_ROOT/deployment_template/bin` (on UNIX).
   b. Run the deploy script:
      - On Windows: **`deploy.bat --app ..\..\reference\discover-data\deploy.xml`**
      - On UNIX: **`deploy.sh --app ../../reference/discover-data/deploy.xml`**
   c. Specify the application name **`Test`** and specify the following ports when prompted:

Port	Recommended value
Live Dgraph	15100
Authoring Dgraph	15102

Port	Recommended value
LogServer	15110

2. Provision the test application.
   a. Ensure that the HTTP Service and Tools Service are running.
   b. From a command prompt, navigate to `<APP-DIR>\control` (on Windows) or `<APP-DIR>/control` (on UNIX).
   c. Run `initialize_services`.
   d. Run `load_baseline_test_data`.
   e. Run `baseline_update`.
3. Deploy a copy of the authoring instance of the Discover Electronics application.
   a. Navigate to `%ENDECA_TOOLS_ROOT%\reference` (on Windows) or `$ENDECA_TOOLS_ROOT/reference` (on UNIX).
   b. Make a copy of the directory `discover-electronics-authoring` and save the copy with the name `sandbox` in the same parent directory.
   c. Navigate to the `test` directory and then to the `WEB-INF` subdirectory.
   d. Open `assembler-context.xml` in a text editor.
   e. Locate the `CARTRIDGE SUPPORT` section:

```
<!--
    ########################################################################
    # CARTRIDGE SUPPORT
    #
    # The following section configures managers and other supporting objects.
    #
-->
```

   f. In the `mdexResource` bean, update the Dgraph port:

```
<bean id="mdexResource" scope="request"
class="com.endeca.infront.navigation.model.MdexResource">
    <property name="appName" value="${workbench.app.name}" />
    <property name="host" value="localhost" />
    <property name="port" value="15102" />
    <property name="sslEnabled" value="${mdex.sslEnabled}" />
    <property name="recordSpecName" value="common.id" />
</bean>
```

   g. Locate the `Content Sources` section:

```
<!--
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~ Content Sources
```

```
-->
```

h. In the `ContentSource` bean, update the application name:

```
<bean id="ContentSource""
 class="com.endeca.infront.content.source.WorkbenchContentSource"
scope="singleton" init-method="init" destroy-method="destroy">
    <property name="storeFactory" ref="storeFactory"/>
    <property name="defaultSiteRootPath" ref="defaultSiteRootPath" />
    <property name="appName" value="${workbench.app.name}"/>
    <property name="siteManager" ref="siteManager"/>
</bean>
```

i. In the `authoringMediaSources` bean, update the application name:

```
<bean id="authoringMediaSources" class="java.util.ArrayList" lazy-init="true">
    <constructor-arg>
        <list>
            <bean class="com.endeca.infront.cartridge.model.MediaSourceConfig">
                <property name="sourceName" value="IFCRSource" />
                <property name="sourceValue" value="http://localhost:8006/ifcr/
sites/Test/media/" />
            </bean>
            <bean class="com.endeca.infront.cartridge.model.MediaSourceConfig">
                <property name="sourceName" value="default" />
                <property name="sourceValue" value="http://localhost:8006/ifcr/
sites/Test/media/" />
            </bean>
        </list>
    </constructor-arg>
</bean>
```

j. Save and close the file.

k. Navigate to `%ENDECA_TOOLS_CONF%\conf\Standalone\localhost` (on Windows) or `$ENDECA_TOOLS_CONF/conf/Standalone/localhost` (on UNIX).

l. Make a copy of `discover-authoring.xml` and save the copy with the name `test` in the same directory.

m. Open `test.xml` in a text editor.

n. Change the value of `docBase` as follows:

```
docBase="${catalina.base}/../../reference/test"
```

o. Restart the Tools Service.

4. Validate your new sandbox application:

a. Navigate to `http://<WorkbenchHost>:8006/login` and verify that **`Test`** displays as an option in the Application drop-down.

b. Select the **`Test`** application and verify that the sample page content from Discover Electronics is available in Experience Manager.

c. In a separate browser window, navigate to the newly deployed sandbox application, at `http://<WorkbenchHost>:8006/test` and verify that it displays.

5. Optionally, update the Workbench configuration to use the test Web application for preview.

   a. Ensure that you are logged in to the Test application in Workbench.

   b. Select Application Configuration.

   c. Specify the URL to the sandbox application (for example, `http://<WorkbenchHost>:8006/test`) as the Preview URL.

   d. Preview a page from Experience Manager by selecting a page or content item and clicking Preview in the upper right.

Creating a Spring context file for sample cartridges

Oracle recommends that you specify the configuration for the sample cartridges in a separate Spring context file from the core cartridges.

To create a Spring context file for the sample cartridges:

1. Navigate to `%ENDECA_TOOLS_ROOT%\reference\sandbox\WEB-INF` (on Windows) or `$ENDECA_TOOLS_ROOT/reference/sandbox/WEB-INF` (on UNIX).

2. Open `assembler-context.xml` in a text editor.

3. At the top of the file, add the following `import`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
">
    <bean
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>WEB-INF/assembler.properties</value>
            </list>
        </property>
    </bean>

    <import resource="endeca-url-config.xml"/>
    <import resource="perf-logging-config.xml"/>
  <import resource="sample-cartridge-config.xml" />
```

4. Delete the configuration for the "Hello, World" sample cartridge that we added in an earlier example.

```
    <!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ BEAN: CartridgeHandler_Hello
    -->
    <bean id="CartridgeHandler_Hello"
    class="com.endeca.sample.cartridges.UppercaseCartridgeHandler"
```

```
    scope="prototype" />
```

5. Save and close the file.
6. Create a new file named `sample-cartridge-config.xml` in the same directory with the following contents:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ BEAN: CartridgeHandler_Hello
    -->
    <bean id="CartridgeHandler_Hello"
    class="com.endeca.sample.cartridges.UppercaseCartridgeHandler"
    scope="prototype" />

</beans>
```

7. Save and close the file.
8. Validate the new configuration by adding the "Hello, World" cartridge to your new sandbox application.
   a. Copy the "Hello, World" directory and its contents (`HelloWorld\`) from the Discover Electronics application (`<APP-DIR>\config\cartridge_templates`) to the sandbox application.
   b. Upload the template to Workbench using the `set_templates` script.
   c. Using Experience Manager, add the cartridge to the default page of the sandbox application and save your changes.
   d. Verify that the `Hello.jsp` renderer and `UppercaseCartridgeHandler` are present in the sandbox Web application. (These should have been included when you copied the Discover Electronics authoring application.)
   e. Refresh the sandbox application (`http://<WorkbenchHost>:8006/sandbox`) and verify that the text you entered in Experience Manager displays, and has been converted to all-uppercase letters.

RSS Feed cartridge

In this example, we build a cartridge that displays items from an RSS feed.

This cartridge enables a business user to specify some basic information about an existing RSS feed in Experience Manager. The cartridge handler fetches the RSS results and returns an output model to the client suitable for rendering.

It demonstrates the following use cases:

- Using a cartridge handler to fetch information from a source other than an MDEX Engine.
- Using the business user configuration from Experience Manager as input into the assembly process and returning a different output model from the configuration model.

In this cartridge, we create the following components:

Component	Description
cartridge template	Enables the business user to specify the URL to an RSS feed and the number of entries to display.
cartridge handler	Fetches results from the RSS feed and returns a number of entries up to the value specified by the business user or the number of entries in the feed, whichever is lower.
cartridge renderer	Displays the name of the feed with a link to the channel URL, and the title and description of each entry with a link to the entry on the original site.

Creating the cartridge template

The business user needs to be able to configure the RSS Feed with a URL and the number of entries to display.

To create the RSS Feed template and add it to your application:

1. Create a new template based on the example below.

   Since the number of entries is expected to be an integer, the example uses a `NumericStepperEditor` for this property. In the example, we specify a default value of `5` for the number of entries.

2. Create a directory with the name `RssFeed` in the templates directory of your application.

3. Save the template with the name `_.json` to the `RssFeed` directory of your application.

4. Upload the template using the `set_templates` script.

5. Add the cartridge to the default search and navigation page.

   Note

   The sample renderer for this cartridge works best with RSS feeds that have brief descriptions with no images or advertisements in the description field. A possible enhancement to this cartridge would be to make displaying the description configurable.

6. Save your changes to the page.

The cartridge instance configuration is saved as JSON. At this point, because there is no cartridge handler specified for this cartridge, the same configuration is passed to the client as the response from the Assembler.

The following shows the sample template for the RSS Feed cartridge:

```
{
   "ecr:type": "template",
   "@group": "SecondaryContent",
   "@description":"A cartridge that displays entries from an RSS feed.",
   "@thumbnailUrl":"/ifcr/tools/xmgr/img/template_thumbnails/sidebar_content.jpg",
   "defaultContentItem":{
      "@name":"RSS cartridge",
      "feedUrl": "",
      "numEntries": "5"
   },
   "editorPanel":{
      "editor":"editors/DefaultEditorPanel",
      "children":[
```

```
         {
            "editor": "GroupLabel",
            "label": "${group.contents.label}"
            },
         {
            "editor":"editors/StringEditor",
            "label":"Feed URL",
            "propertyName":"feedUrl",
            "bootomLabel":"The address of the RSS feed, such as http://www.oracle.com/
us/corporate/press/rss/rss-pr.xml"
         },
         {
            "editor":"editors/NumericStepperEditor",
            "label":"Number of entries to display",
            "minValue":1,
            "maxValue":15,
            "propertyName":"numEntries"
         }
      ]
   },
    "typeInfo": {
        "feedUrl": {"@propertyType": "String"},
        "numEntries": {"@propertyType": "String"}
    }
}
```

Creating the cartridge handler

The cartridge handler fetches the RSS results and returns an output model to the client suitable for rendering.

To create the RSS Feed cartridge handler and add it to the application:

1. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `AbstractCartridgeHandler`.
2. Compile the cartridge handler and add the compiled class to your application.
3. Configure the Assembler to use the `RssFeedHandler` for the RSS Feed cartridge by adding the following to the Spring context file:

```
      <!--
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      ~ BEAN: CartridgeHandler_RssFeed
      -->
      <bean id="CartridgeHandler_RssFeed"
      class="com.endeca.sample.cartridges.RssFeedHandler"
      scope="prototype" />
```

4. Restart the Tools Service.
5. Refresh the application.

The RSS feed does yet appear because we have not created the renderer. Nevertheless, you can validate that the response model has been populated with the information that we want to display in the JSON view:

```
{

    "@type": "RssFeed",
```

```
    "name": "RSS cartridge",
    "feedUrl": "http://www.wired.com/reviews/feed/",
    "numEntries": "5",
    "chanTitle": "Product Reviews",
    "chanUrl": "http://www.wired.com/reviews",
    "entries": [
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        }
    ]

}
```

The following shows the code for the sample RSS Feed cartridge handler:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.AbstractCartridgeHandler;
import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.assembler.ContentItem;
import com.endeca.infront.assembler.BasicContentItem;
import java.net.URL;
import java.util.ArrayList;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.CharacterData;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;

public class RssFeedHandler extends AbstractCartridgeHandler {
```

```
public ContentItem process(ContentItem pContentItem)
  throws CartridgeHandlerException {

 final String urlString = (String) pContentItem.get("feedUrl");
 final int numEntries =
   Integer.parseInt((String)pContentItem.get("numEntries"));

 try {
  URL url = new URL(urlString);
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder docBuilder = factory.newDocumentBuilder();
  Document RssContents = docBuilder.parse(url.openStream());

  // get the channel info
  Element channel =
   (Element)RssContents.getElementsByTagName("channel").item(0);
  pContentItem.put("chanTitle", getElementValue(channel, "title"));
  pContentItem.put("chanUrl", getElementValue(channel, "link"));

  // get the entries and add them to a list
  ArrayList<ContentItem> entries = new ArrayList<ContentItem>(numEntries);
  NodeList nodes = RssContents.getElementsByTagName("item");
  for(int i=0; i<numEntries; i++) {
   Element element = (Element)nodes.item(i);
   if (element!=null) {
    ContentItem entry = new BasicContentItem("rssEntry");
    entry.put("itemTitle", getElementValue(element, "title"));
    entry.put("itemUrl", getElementValue(element, "link"));
    entry.put("itemDesc", getElementValue(element, "description"));
    entries.add(entry);
   }
  }
  pContentItem.put("entries", entries);
 }
 catch (Exception e) {
  throw new CartridgeHandlerException(e);
 }

 return pContentItem;
}

private static String getCharacterDataFromElement(Element e) {
 try {
  Node child = e.getFirstChild();
  if(child instanceof CharacterData) {
   CharacterData cd = (CharacterData) child;
   return cd.getData();
  }
 }
 catch(Exception ex) {
 }
 return "";
}

private static String getElementValue(Element parent, String label) {
 return getCharacterDataFromElement(
   (Element)parent.getElementsByTagName(label).item(0));
}
```

```
}
```

Creating the cartridge renderer

The renderer displays a summary of the results with links that take the site visitor to the site that originated the RSS feed.

To create a renderer for the RSS feed:

1. Create a new JSP page based on the example below.
2. Save the renderer to `/WEB-INF/views/desktop/RssFeed/RssFeed.jsp`.
3. Refresh the application to see the result.

The results from the RSS feed display in the right sidebar.

The following shows the code for the sample RSS Feed renderer in JSP:

```
<%@page language="java" pageEncoding="UTF-8"
contentType="text/html;charset=UTF-8"%>

<%@include file="/WEB-INF/views/include.jsp"%>

<div style="padding:2ex 0">
<b><a href="${component.chanUrl}">${component.chanTitle}</a></b>
<c:forEach var="rssEntry" items="${component.entries}">
    <p><a href="${rssEntry.itemUrl}">${rssEntry.itemTitle}</a><br/>
    ${rssEntry.itemDesc}</p>
</c:forEach>
</div>
```

Custom Record Details cartridge with availability information

In this example, we extend the Record Details cartridge to display information about the availability of a product.

It demonstrates the following use cases:

- Extending one of the core cartridges
- Combining results from the MDEX Engine with information from another source during the `process()` phase of the assembly cycle
- Configuring a third-party service through Spring

In this cartridge, we create the following components:

Component	Description
cartridge handler	Extends the `RecordDetailsHandler` to add a property to the response model containing availability information.

Component	Description
mock "availability service"	Stands in for a real source of availability information such as an inventory system.

Because this cartridge does not introduce any change in the configuration options for the business user, there are no template changes for this cartridge. To enable the full functionality of this cartridge, the renderer should be updated to display the availability information, however that is not demonstrated in this guide.

Creating the cartridge handler and supporting classes

The `AvailabilityRecordDetailsHandler` extends the core `RecordDetailsHandler` to call a simple mock availability service to retrieve availability information about a particular record.

To create a cartridge handler that calls an availability service:

1. Create the following classes: `Availability`, `AvailabilityService`, and `FixedAvailabilityService` based on the examples below.

   The `AvailabilityService` defines an interface that returns availability information based on a record identifier, and `FixedAvailabilityService` provides a basic implementation of the interface.

2. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `RecordDetailsHandler`.

   The handler takes the results of the MDEX Engine query and adds an additional property that represents the product availability.

3. Compile the classes and add them to your application.

4. Configure the Assembler to use the `AvailabilityRecordDetailsHandler` for the Record Details cartridge by editing the Spring context file as in the following example.

   Note

   If you have created a `sample-cartridge-config.xml` file for configuring the example cartridges, copy the `CartridgeHandler_ResultsList` bean from `assembler-context.xml` to your sample context file, comment out the version in `assembler-context.xml`, and then modify the version in your sample context file as indicated below.

```
<bean id="CartridgeHandler_RecordDetails"
  class="com.endeca.sample.cartridges.AvailabilityRecordDetailsHandler"
  parent="NavigationCartridgeHandler" scope="prototype" >
    <property name="recordState" ref="recordState" />
    <property name="availabilityService" ref="availabilityService" />
    <property name="recordSpec" value="common.id" />
    <property name="availabilityPropertyName"
      value="product.availability" />
</bean>

<bean id="availabilityService"
  class="com.endeca.sample.cartridges.FixedAvailabilityService"
  scope="singleton" >
  <!-- Implementation-specific configuration for the service
       could be specified here -->
```

```
  </bean>
```

5. Restart the Tools Service.
6. Refresh the application and then click on any record to view its details page.

The availability property is now returned as part of the record details information:

```
{
    "@type": "RecordDetailsPageSlot",
    "name": "Record Details Page",
    "contentCollection": "Record Details Pages",
    "ruleLimit": "1",
    "contents": [
        {
            …
            },
            "recordDetails": {
                "@type": "RecordDetails",
                "record": {
                   "@class": "com.endeca.infront.cartridge.model.Record",
                   "numRecords": 1,
                   "attributes": {
                        …
                        "product.availability": [
                            "BACKORDER"
                        ],
                        …
                   },
                   "records": [ … ]
                },
                "name": "Record Details"
            }
        }
    ],
    …
}
```

The renderer can now be updated to display availability information based on the value of this property.

The following shows the code for the availability service and its supporting classes:

```
package com.endeca.sample.cartridges;

public enum Availability {
    IMMEDIATE,
    WEEK,
    DROP_SHIP,
    BACKORDER;

}
```

```
package com.endeca.sample.cartridges;

public interface AvailabilityService {
```

```
    Availability getAvailabilityFor(String identifier);
}
```

```
package com.endeca.sample.cartridges;

public class FixedAvailabilityService implements AvailabilityService {

    public Availability getAvailabilityFor(String identifier) {
        try {
            return Availability.valueOf(identifier);
        } catch (IllegalArgumentException e) {
            return Availability.BACKORDER;
        }
    }
}
```

The following shows the code for the custom cartridge handler:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.cartridge.RecordDetails;
import com.endeca.infront.cartridge.RecordDetailsConfig;
import com.endeca.infront.cartridge.RecordDetailsHandler;
import com.endeca.infront.cartridge.model.Attribute;
import org.springframework.beans.factory.annotation.Required;

public class AvailabilityRecordDetailsHandler extends RecordDetailsHandler {

    private AvailabilityService availabilityService;
    private String recordSpec;
    private String availabilityPropertyName;

    @Required
    public void setAvailabilityService(
      AvailabilityService availabilityService_) {
        availabilityService = availabilityService_;
    }

    @Required
    public void setRecordSpec(String recordSpec_) {
        recordSpec = recordSpec_;
    }

    @Required
    public void setAvailabilityPropertyName(
      String availabilityPropertyName_) {
        availabilityPropertyName = availabilityPropertyName_;
    }

    @Override
    public RecordDetails process(RecordDetailsConfig detailsConfig)
      throws CartridgeHandlerException {
        RecordDetails details = super.process(detailsConfig);
        if (null == details) return null;
        Attribute attr =
          details.getRecord().getAttributes().get(recordSpec);
```

```
        if (null == attr || 1 != attr.size()) {
            throw new CartridgeHandlerException("No record spec
              available on record, or spec is multiassign");
        }
        Attribute<Availability> availability =
          new Attribute<Availability>();
        availability.add(
          availabilityService.getAvailabilityFor(attr.toString()));
        details.getRecord().getAttributes().put(availabilityPropertyName,
          availability);
        return details;
    }

}
```

Custom Results List with recommendations

In this example, we extend the Results List cartridge to boost certain products based on information from a recommendation engine.

It demonstrates the following use cases:

- Extending one of the core cartridges
- Using data from another source to modify the query to the MDEX Engine created during the `preprocess()` phase of the assembly cycle
- Configuring a third-party service through Spring

In this cartridge, we create the following components:

Component	Description
cartridge handler	Extends the `ResultsListHandler` to retrieve a set of items to boost from a recommendations engine and add a boost stratum to the MDEX Engine query.
mock recommendations service	Stands in for a real source of recommendations.

Because this cartridge does not introduce any change in the configuration options for the business user, there are no template changes for this cartridge. Additionally, the response model for the customized cartridge is the same as the default Results List (only with the records in a different order), so there is no need for changes to the default renderer.

Creating the cartridge handler and supporting classes

The `RecommendationsResultsListHandler` extends the core `ResultsListHandler` to call a simple mock recommendations service and boosts the recommended products.

To create a cartridge handler that boosts recommended records:

1. Create the interface `RecommendationService` and the concrete implementation `TestRecommendationService` based on the examples below.

As a proof of concept, the recommendations service always returns the same recommendations from the Discover Electronics data set.

2. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `ResultsListHandler`.

   The handler retrieves a list of recommended records from the service and adds them to a boost stratum for the MDEX Engine query. If the records are present in the results set, they are boosted to the top of the results list.

3. Compile the classes and add them to your application.

4. Configure the Assembler to use the `RecommendationsResultsListHandler` for the Results List cartridge by editing the Spring context file as follows:

   Note

   If you have created a `sample-cartridge-config.xml` file for configuring the example cartridges, copy the `CartridgeHandler_ResultsList` bean from `assembler-context.xml` to your sample context file, comment out the version in `assembler-context.xml`, and then modify the version in your sample context file as indicated below.

```
<bean id="CartridgeHandler_ResultsList"
  class="com.endeca.sample.cartridges.RecommendationsResultsListHandler"
  parent="NavigationCartridgeHandler" scope="prototype">
    <property name="contentItemInitializer">
      <!-- additional elements omitted from this example -->
    </property>
    <property name="sortOptions">
      <!-- additional elements omitted from this example -->
    </property>
    <property name="recommendationService" ref="recommendationService" />
    <property name="recordSpec" value="common.id"/>
</bean>

<bean id="recommendationService"
  class="com.endeca.sample.cartridges.TestRecommendationService"
  scope="singleton" >
  <!-- Implementation-specific configuration for the service
       could be specified here -->
</bean>
```

5. Restart the Tools Service.

6. Refresh the application.

The recommended records are boosted to the top of the results.

The following shows the code for the recommendations service interface and concrete implementation:

```
package com.endeca.sample.cartridges;

import java.util.List;

public interface RecommendationService {
    public List<String> getRecommendedProductIds();
```

```
}
```

```
package com.endeca.sample.cartridges;

import java.util.Arrays;
import java.util.List;

public class TestRecommendationService
  implements RecommendationService {
    public static final List<String> IDS =
      Arrays.asList("5891932", "6001963", "1438066", "1581692",
        "2708142", "1235424", "3422480");

    public List<String> getRecommendedProductIds() {
        return IDS;
    }

}
```

The following shows the code for the custom cartridge handler:

```
package com.endeca.sample.cartridges;

import java.util.ArrayList;
import java.util.List;

import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.cartridge.ResultsListConfig;
import com.endeca.infront.cartridge.ResultsListHandler;
import com.endeca.infront.navigation.model.CollectionFilter;
import com.endeca.infront.navigation.model.PropertyFilter;

public class RecommendationsResultsListHandler extends ResultsListHandler {
    private RecommendationService recommendationService;
    private String recordSpec;

    public String getRecordSpec() {
        return recordSpec;
    }

    public void setRecordSpec(String recordSpec_) {
        this.recordSpec = recordSpec_;
    }

    public void setRecommendationService(
      RecommendationService recommendationService_) {
        recommendationService = recommendationService_;
    }

    /**
     * This cartridge will get the list of recommended products
     * (by record spec) and explicitly boost each one of them using
     * a PropertyFilter.
     */
    @Override
    public void preprocess(ResultsListConfig pContentItem)
      throws CartridgeHandlerException {
```

```
        List<String> ids =
          recommendationService.getRecommendedProductIds();
        List<CollectionFilter> boostFilters =
          new ArrayList<CollectionFilter>(
                ids.size());
        for (String s : ids) {
            boostFilters.add(new CollectionFilter(new PropertyFilter(
                    recordSpec, s)));
        }

        pContentItem.setBoostStrata(boostFilters);
        super.preprocess(pContentItem);
    }

}
```

"Hello, World" cartridge with layered color configuration

In this example, we extend the "Hello, World" example cartridge to demonstrate the layering of configuration from several sources.

In this scenario, we can define a default color for the message in our "Hello, World" cartridge, which the business user can override on a per-instance basis in Experience Manager. The site visitor can also select a preferred color from the client application.

It demonstrates the following use cases:

- Combining the default cartridge configuration, cartridge instance configuration, and request-based configuration using the `ConfigInitializer` and `RequestParamMarshaller` helper classes
- Using a cartridge configuration bean

In this cartridge, we create the following components:

Component	Description
cartridge handler	Uses the `ColorConfigInitializer` to layer multiple sources of configuration for message color.
cartridge configuration bean	Provides a means of specifying default values for this cartridge via Spring.
cartridge renderer	Provides a drop-down list from which the site visitor can choose a color for the message.

Because this cartridge does not introduce any change in the configuration options for the business user, there are no template changes for this cartridge.

Creating the cartridge handler and supporting classes

The cartridge handler combines the various sources of configuration for message color using the `ConfigInitializer` and `RequestParamMarshaller` helper classes.

To create the "Hello, World" cartridge handler with color configuration and add it to the application:

1. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `AbstractCartridgeHandler`.
2. Create a configuration bean for this cartridge based on the example below. This enables us to define default values for the cartridge properties in the Spring context file.
3. Compile the cartridge handler and configuration bean and add them to your application.
4. Configure the Assembler to use the `ColorConfigHandler` for the "Hello, World" cartridge by editing the Spring context file as follows:

```
<bean id="CartridgeHandler_Hello"
  class="com.endeca.sample.cartridges.ColorConfigHandler"
  scope="prototype">
  <property name="contentItemInitializer">
    <bean class="com.endeca.infront.cartridge.ConfigInitializer"
      scope="singleton">
      <property name="defaults">
        <bean class="com.endeca.sample.cartridges.ColorConfig"
          scope="singleton">
          <property name="messageColor" value="#FF6600"/>
        </bean>
      </property>
      <property name="requestParamMarshaller">
        <bean
          class="com.endeca.infront.cartridge.RequestParamMarshaller"
          scope="singleton">
          <property name="httpServletRequest" ref="httpServletRequest"/>
          <property name="requestMap">
            <map>
              <entry key="color" value="messageColor"/>
            </map>
          </property>
        </bean>
      </property>
    </bean>
  </property>
  <property name="colorOptions">
    <map>
      <entry key="Red" value="#FF0000"/>
      <entry key="Green" value="#00FF00"/>
      <entry key="Blue" value="#0000FF"/>
      <entry key="Black" value="#000000"/>
    </map>
  </property>
</bean>
```

5. Restart the Tools Service.
6. Refresh the application.

The color options do not display yet because we have not updated the renderer, but you can validate that the response model has been populated with the information that we want the renderer to use in the JSON view:

```
{

    "@type": "Hello",
    "name": "Hello cartridge",
```

```
    "message": "Hello, color world!",
    "messageColor": "#0000FF",
    "colorOptions": [
        {
            "@type": "colorOption",
            "hexCode": "#FF0000",
            "label": "Red"
        },
        {
            "@type": "colorOption",
            "hexCode": "#00FF00",
            "label": "Green"
        },
        {
            "@type": "colorOption",
            "hexCode": "#0000FF",
            "label": "Blue"
        },
        {
            "@type": "colorOption",
            "hexCode": "#000000",
            "label": "Black"
        }
    ]

}
```

The following shows the code for the sample "Hello, World" cartridge handler with color configuration:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.AbstractCartridgeHandler;
import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.assembler.ContentItem;
import com.endeca.infront.assembler.BasicContentItem;
import com.endeca.infront.assembler.ContentItemInitializer;
import com.endeca.sample.cartridges.ColorConfig;
import java.util.ArrayList;
import java.util.Map;

public class ColorConfigHandler extends AbstractCartridgeHandler {

 private ContentItemInitializer mInitializer;
 private Map<String, String> mColorOptions;

  public void setContentItemInitializer(ContentItemInitializer initializer) {
    mInitializer = initializer;
  }

  public void setColorOptions(Map<String, String> colorOptions) {
    mColorOptions = colorOptions;
  }

  /**
   * Returns the merged configuration based on Spring defaults,
   * Experience Manager configuration, and request parameters
   */
  @Override
  public ContentItem initialize(ContentItem pContentItem) {
```

```
    // If any configuration from Experience Manager is empty, remove
    // that property so we can use the default value
    for (String key: pContentItem.keySet()) {
      if (((String)pContentItem.get(key)).isEmpty())
        pContentItem.remove(key);
    }
      return mInitializer == null ? new ColorConfig(pContentItem) :
        mInitializer.initialize(pContentItem);
    }

  /**
   * Returns the merged configuration and information about the color options
   * available to the site visitor.
   */
  @Override
  public ContentItem process(ContentItem pContentItem)
    throws CartridgeHandlerException {
    int numColors = mColorOptions.size();
    ArrayList<ContentItem> colors =
      new ArrayList<ContentItem>(numColors);
    if (mColorOptions != null && !mColorOptions.isEmpty()) {
      for (String key: mColorOptions.keySet()) {
        ContentItem color = new BasicContentItem("colorOption");
        color.put("label", key);
        color.put("hexCode", mColorOptions.get(key));
        colors.add(color);
      }
      pContentItem.put("colorOptions", colors);
    }
    return pContentItem;
  }
}
```

The following code implements a basic bean that enables us to specify a default value for the message color in the Spring configuration:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.BasicContentItem;
import com.endeca.infront.assembler.ContentItem;

public class ColorConfig extends BasicContentItem {

  public ColorConfig() {
    super();
  }

  public ColorConfig(final String pType) {
    super(pType);
  }

  public ColorConfig(ContentItem pContentItem) {
    super(pContentItem);
  }

  public String getMessageColor() {
    return getTypedProperty("messageColor");
  }
```

```
  public void setMessageColor(String color) {
    this.put("messageColor", color);
  }
}
```

Creating the cartridge renderer

In this example we update the "Hello, World" renderer to add a control for the site visitor to select a color for the message.

To add a drop-down for the site visitor to select a message color based on the options configured for this cartridge:

1. Create a new JSP page based on the example below, or update the renderer you previously created by adding the section in bold.
2. Save the renderer to `/WEB-INF/views/desktop/Hello/Hello.jsp`.
3. Refresh the application to verify that the drop-down menu displays.

The following shows the code for the sample "Hello, World" renderer with color choice drop-down in JSP:

```
<%@page language="java" pageEncoding="UTF-8"
contentType="text/html;charset=UTF-8"%>

<%@include file="/WEB-INF/views/include.jsp"%>
<div style="border-style: dotted; border-width: 1px;
border-color: #999999; padding: 10px 10px">
  <div style="font-size: 150%;
    color: ${component.messageColor}">${component.message}
  </div>
  <div style="font-size: 80%; padding: 5px 0px">
    <select onchange="location = this.options[this.selectedIndex].value">
      <option value="">Select a color</option>
      <c:forEach var="colorOption" items="${component.colorOptions}">
        <c:url value="<%= request.getPathInfo() %>" var="colorAction">
          <c:param name="color" value="${colorOption.hexCode}" />
        </c:url>
        <option value="${colorAction}">${colorOption.label}</option>
      </c:forEach>
    </select>
  </div>
</div>
```

Testing the "Hello, World" cartridge with layered color configuration

We can validate that the cartridge handler applies the different sources of configuration properly by incrementally populating each source of the configuration.

To test the "Hello, World" cartridge:

1. In Experience Manager, remove any previously created instance of the Hello cartridge.
2. Insert a new instance of the cartridge on the default page and specify a message string, but do not select a color.
3. Save the page.
4. Refresh the application.

   The message displays using the default color, orange.
5. Going back to Experience Manager, now select a message color for this instance of the cartridge.
6. Refresh the application.

   The message displays using the color configured in Experience Manager.
7. Using the drop-down list on the cartridge, select another color.

   The drop-down control adds a `color` parameter to the URL, which is parsed by the `RequestParamMarshaller` into the `messageColor` property.

Developing Custom Editors

This chapter covers steps for developing your own Experience Manager editors using the Editor SDK.

The Editor SDK

The Editor SDK is included with your Tools and Frameworks installation. It provides a framework for developing Experience Manager custom editors.

The Editor SDK is based on Oracle JavaScript Extension Toolkit (JET) and Knockout.

- Oracle JET is a modular open source toolkit based on modern JavaScript, CSS and HTML design and development principles. Oracle Jet contains a collection of open source JavaScript libraries and Oracle-contributed JavaScript libraries that make it as simple as possible to build Experience Manager editors.

  To learn more about Oracle JET, see http://www.oracle.com/webfolder/technetwork/jet/index.html.
- Knockout is a JavaScript library that helps you to create editor user interfaces. It is based on the Model-View-ViewModel (MVVM) architecture pattern. Knockout provides data binding between your data model and the editor user interface. Changes to the data model are automatically reflected in the editor user interface and any changes to the editor user interface are automatically reflected to the data model.

  To learn more about Knockout, see http://knockoutjs.com/.

The custom editors that you build using this SDK consist of the following components:

- `editor.js`, a Knockout view model that defines the business logic. This is where you extend the base Editor class and implement required life cycle methods.

- `editor.html`, a HTML template that defines the editor user interface using Oracle JET components.
- `_.json`, a resource node definition file with an `ecr:type` of `editor`. This file lets you register editors with the system by importing `editor.js`, `editor.html` along with any configuration.

Editor API

The Editor SDK provides **Editor**, an abstract base JavaScript extension. The Editor extension defines the life cycle for executing a custom editor.

Custom editors extend the base Editor class.

Editor(pConfig) constructor

The `Editor(pConfig)` constructor is invoked when a cartridge template corresponding to an asset is loaded into Experience Manager. The framework instantiates every editor in the order that they are mapped in the template's editor panel. The framework passes the configuration object that is built from various configurations defined at the cartridge template and editor levels to the editor's constructor.

The `Editor(pConfig)` constructor instantiates all the editor-level properties and helper objects.

pConfig is the editor's configuration that is supplied in a specific EAC application context. This is the configuration that you define in the editor's _.json file using `config` attribute.

Life cycle methods

This section describes the following life cycle methods. Note that override is optional for all of them.

initialize(pTemplateConfig, pContentItem)

The framework invokes the initialize life cycle method immediately after the instantiation of the editor that passes the template configuration object and content item.

- Parses meta information configured in the cartridge template and populates the same into the editor object. This meta information can be specific to:
  - Label. For example, localized label name, label position, localized tool-tip information and so on.
  - Generic validations. For example, whether or not the property is required, or the editor is enabled.
  - Validation meta information specific to editor. For example, the NumericStepperEditor might have a number range validator that can be turned on based on minimum or maximum values.
  - Additional information specific to the editor. For example, the ChoiceEditor has a list of choices to be displayed.
- Subscribes to notifications that are pertinent to the editor.
- Initializes custom controllers if there are any specific to editor.
  - The initialize method in the base Editor parses `TemplateConfig` and populates attributes like localized label text, tool tips, or `enabledExpression` that are common for all Experience Manager editors.
  - If a custom editor needs any additional meta information specific to that editor, it must override the initialize method and populate the same.
  - If overridden, this method must explicitly call its initialize method from its super class Editor.

    ```
    Editor.prototype.initialize.call(this, pTemplateConfig, pContentItem)
    ```

Parameter	Description
pTemplateConfig	Object that holds the cartridge template-level configuration associated with the editor.
pContentItem	`ContentItem` object that has the property objects that work with this editor. `ContentItem` that has the property instances created using the property API For more information on the property API, see Property API (page 195).

When initialization is finished, the framework binds editors to the view and moves them to Loading phase. The actual data binding occurs in the `editorReady` method.

editorReady(pProperty)

Once initialization is successful, the framework invokes the `editorReady` life cycle method and passes the appropriate property instance created by the framework.

An editor can override this method, if it needs to wrap the property using any custom extension to the property API or any computations regarding the property value. If overridden, this method must explicitly call `editorReady` from it's super class Editor: `Editor.prototype.editorReady.call (this, pProperty)`

Parameter	Description
pProperty	Property object from the content item which is mapped to this editor in the cartridge template's editor panel. An editor can work with multiple properties, but at any given time, only one property can be mapped against attribute `propertyName` in the editor panel of a cartridge template. Only that property instance whose name has been mapped would be passed to the `editorReady` method. Based on the property's value type, the framework wraps the property in the appropriate class provided by the framework.

For more information on the property API, see Property API (page 195).

In the `editorReady` method of an editor, you can load any data needed by an editor asynchronously. For example, DimensionListEditor needs to load dimension names from MDEX. You should ensure that the

`editorReady` method in the super class is called only when all asynchronous calls have completed. This allows the framework to call `editorReady` on individual editors asynchronously. When an editor is ready, the framework removes the Loading phase and renders the user interface using the editor's own HTML template.

editorError(pLocalizedMessage)

The framework invokes the `editorError` life cycle method when there are errors during the `initialize()` and `editorReady ()` life cycle phases. This method can also be explicitly invoked to handle error scenarios.

- Binds the error template to the editor that displays the localized error message with default styling.
- Editor must unsubscribe for notifications (if any), that it has subscribed to in the `initialize()` phase.

There is no need for an editor to override `editorError` until a different error template must be bound to the editor in error scenarios.

Parameter	Description
pLocalizedMessage	String value of the localized error message

dispose()

The framework invokes this method before de-referencing (marking) this editor instance for garbage collection.

- All the subscriptions made by this editor are un-subscribed here.
- If a subclass extending this editor does it's own subscriptions, it must override this method to un-subscribe it's own subscriptions and invoke the dispose method of it's superclass.

handleAttached()

The framework invokes this method internally by Oracle JET when a workbench user re-visits a previously visited mini-tree node. The method is invoked on each editor instance mapped to that particular node content.

When a page or rule is edited, and after loaded editor instances are cached (when a Workbench user clicks a specific mini-tree node for the first time), all the editors that are mapped to the node's content are instantiated and cached. From this point forward, whenever a Workbench user attempts to revisit the same node, editors are loaded from cache. This action happens because any form errors or modifications to a rule's or page's properties should be retained, even when a user navigates through the mini-tree's nodes.

This is a hook method to perform any post activities once an editor has been reloaded from cache. For example, you might want to re-subscribe for any topics at the editor level. In such a scenario, a specific editor can override this method and implement the appropriate post processing logic.

Note: This will not be called when editor is instantiated and loaded for the first time.

handleDetached()

The framework invokes this method internally by Oracle JET when a workbench user exits or switches from the current mini-tree node. The method is invoked on each editor instance mapped to that particular node content.

When a page or rule is edited, and after loaded editor instances are cached (when a Workbench user clicks a specific mini-tree node for the first time), all the editors that are mapped to the node's content are instantiated and cached. From this point forward, whenever a workbench user attempts to revisit the same node editors are

loaded from cache. This action happens because any form errors or modifications to a rule's or page's properties should be retained, even when a user navigates through the mini-tree's nodes. However, there might be some events that can trigger changes in a rule's or page's property even when not in view.

This is a hook method to perform any post activities, once an editor has been removed from the document DOM. For example, you might want to unsubscribe any topics at the editor level. In such a scenario, a specific editor can override this method and implement the appropriate logic.

Property API

This JavaScript-based property API is used for the following functions:

- Binding a content item property to an editor.
- Wrapping property raw data (this how a property is stored in the ECR) in a property instance which is easily consumed by an editor's Knockout view model and html template.
- When editing is complete, converting property instance back into a format in which it has to be stored into the ECR.
- Notifying other JavaScript components about changes to property values. This can be used in reevaluating enable expressions when a dependent property value changes, for example, enabling/disabling user interface controls and so on.

Class	Description	Methods
Property	The core property class that wraps the property's raw JSON data. It also defines custom Knockout subscriptions to track changes made to a property value. Whenever there is a change in a property value, a EditorEvents.PROPERTY_CHANGED event is triggered.. The property reference that was modified is passed as the event payload. Every editor has an observable property that is set to the property reference that editor works with.	**copyFrom (pProperty)** - Accepts raw the JSON object that represents a valid property with a name, type, and an optional value. Used to update any bindings that are needed by the user interface. Basic functionality is to set value as a Knockout observable or observableArray. **toJSON()** - Converts the property instance to JSON, in a format to be stored in the ECR. For example, a property instance will have observable or observableArray to bind value to the editor, which has to be converted to normal value.
StringProperty	A subclass of Property class. Deals with property objects. Value type is a String.	

Class	Description	Methods
BooleanProperty	A subclass to Property class. Deals with property objects. Value type is a Boolean.	Overrides the **copyFrom** method, and wraps property's value in a Boolean.
JSONProperty	A subclass to Property for Item and List types. This is used to wrap properties whose values are either JSONObject or JSONArray. When the framework parses a content item, in case of a list, it would not be possible to distinguish whether the value is a primitive list or an object list. Also in case of Item, it is not possible to determine value is of which object type. In these scenarios, the framework wraps a property in JSONProperty and passes it to the editorReady method of the corresponding editor. Within the editorReady method, the corresponding property instance (ObjectProperty, ListProperty, ObjectListProperty, or any extension to Property that customers might develop and plug in) can be created as an editor will know about the value type that it is working with. This is also used when there is no type specified for a property.	Overrides **copyFrom** method and sets property's value as an observable or observableArray based on the type. Overrides **toJSON** method and converts the property instance to JSON, in a format to be stored in ECR.

Class	Description	Methods
ObjectProperty	A subclass to Property class for property objects whose value is an object. This is needed to manipulate internal attributes of the object. Otherwise JSONProperty works with Object. For example in case of media property in MediaBanner cartridge, whose value is an object, there are attributes which are computed based on some other attributes in the object.	Overrides the following methods: **copyFrom**: Copies property's value into corresponding ObjectValue instance. **toJSON**: Converts the property's value into JSON format, in a format to be stored in ECR.

	<pre>"media": {
	"@class":
	"com.endeca.infront.
	
	cartridge.model.MediaObject",
	"uri":
	"banner_bags_761x225.jpg",
	"contentWidth": "761",
	"contentHeight": "225",
	"contentBytes": "3546313",
	"contentType": "Image",
	"contentSrcKey": "default"
	}</pre>	

	In this object, based on the image that we select, for example the uri attribute, other attributes like contentWidth, contentHeight, contentBytes, contentType are computed. We can define a Image.js model object that uses a Knockout computed observables concept to compute these attributes based on the uri attribute. This Image must be wrapped using ObjectProperty.	
ObjectListProperty	A subclass to Property for property objects whose value is an array or a list built from objects. As described for ObjectProperty, this is needed only when performing custom manipulations on an object's attributes. Otherwise use JSONProperty.	Overrides the following methods: **copyFrom**: Copies each object in the list into corresponding ObjectValue instance. **toJSON**: Converts each object in the value list into JSON format, in a format to be stored in ECR.
ContentItemProperty	A subclass to ObjectProperty for property objects whose value is an object of type ContentItem.	

Class	Description	Methods
ContentItemListProperty	A subclass to ObjectListProperty for property objects whose value is an array or a list built from objects, of type ContentItem. This also stores data in a tree structure.	
ObjectValue	A wrapper to property value which is an object. Provides method signatures to be implemented by any object values.	**copyFrom(pPropertyValue)**: Copies plain JSON object into the ObjectValue. For example, in SortOption, there can be an observable label, which would be populated under the copyFrom(..) method. **toJSON()**: In Experience Manager, the implementation converts the Knockout object back to a plain JSON object.

About developing custom editors

Keep in mind the following information as you develop your custom editors:

- Editors must be placed in the following directory:

```
<app dir>/config/import/editors/<optional namespace>/<editor name>
```

For example:

```
Discover/config/import/editors/custom/StringEditor
```

- Editors must refer to a Property object which holds primitive or complex property values.
- The Editor SDK uses `requireJS` to resolve and load dependent JavaScript modules.
- The Editor SDK provides `TemplateEngine.js`, an extension to the Knockout native template engine to refer templates dynamically from the view model.

Building custom editors

Follow these instructions to build your custom editor:

1. Load dependencies using the `requireJS` library

   For example:

```
Discover/config/import/editors/custom/StringEditor/editor.js
```

JavaScript libraries are resolved relative to the following path mappings:

Library	Path mapping	Example
jQuery	jquery	//In editor.js ... var $ = require("jquery"); ...
Knockout	knockout	//In editor.js ... var ko = require("knockout"); ... this.min = ko.observable(); ...
Hammer JS	hammerjs	// In editor.js var lib = require("hammerjs");

Oracle JET	ojs	// In editor.js // Load OJET component for input text
		require("ojs/ojcomponents");
		require("ojs/ojinputtext");

Underscore JS	underscore	// In editor.js var _ = require("underscore");
Promise	promise	// in editor.js var p = require("promise");

JavaScript modules provided by the framework must be loaded according to the JavaScript documentation installed with Tools and Frameworks.

```
define(["app/xmgr/view/editors/Editor",
 "core/TemplateEngine",
 "template!editors/custom/StringEditor/editor.html"
function(Editor, TemplateEngine, EditorTemplate) {
}
}
```

In addition to `editor.js`, if a custom editor depends on any other custom JavaScript extensions, then these extensions can be resolved within `editor.js` using the relative paths from the current editor context or the paths relative to the editors folder. For example, if a custom editor needs three more JavaScript extensions that are available at these locations:

- `Discover/config/import/editors/MediaEditor/model/Media.js,`,
- Discover/config/import/editors/MediaEditor/model/Image.js
- `Discover/config/import/editors/MediaEditor/model/Video.js`

The `editor.js` references them like in the following example:

```
define(["app/xmgr/view/editors/Editor",
 "core/TemplateEngine",
 "template!editors/custom/StringEditor/editor.html",
 "editors/MediaEditor",editors/MediaEditor/model/Image"],
 function(Editor, TemplateEngine, EditorTemplate, MediaEditor,Image) {
});

                                (OR)

define(["app/xmgr/view/editors/Editor",
 "core/TemplateEngine",
 "template!editors/custom/StringEditor/editor.html",
 "editors/MediaEditor",./model/Image"],
 function(Editor, TemplateEngine, EditorTemplate, MediaEditor,Image) {
});
```

2. The custom editor must extend the base Editor class.

   According to the JavaScript module pattern, to extend the Editor, the custom editor must define a constructor method with Configuration as a parameter and invoke the super class constructor. You must create an Editor object using a prototypical instance and copy this Editor prototype to the custom editor prototype. For example:

```
...
var StringEditor = function(pConfig) {
    var self = this;
    Editor.call(self, pConfig);
};
StringEditor.prototype = Object.create(Editor.prototype);
...
```

3. Optional. Override the `initialize()` life cycle method to implement any additional parsing that is specific to the custom editor.

   For example, ChoiceEditor would have an array of choices. In this case, parse the cartridge template and populate choices under the initialize method. If overridden, this method should explicitly call the `Editor::initialize()` super class method.

```
...
ChoiceEditor.prototype.initialize = function(pTemplateConfig, pContentItem) {
    var self = this;
    //Populate choices from the pTemplateConfig object
    Editor.prototype.initialize.call(self, pTemplateConfig);
};
...
```

4. Optional. Override the `editorReady()` life cycle method to implement any additional manipulations regarding a property or properties within which an editor works.

5. Define an `editor.html` template. In the following example, `Discover/config/import/editors/custom/StringEditor/editor.html` uses `ojInputText`, the Oracle JET component, and `oj-flex`, the default Oracle JET responsive layout. To ensure uniqueness generate a unique ID for each of the editor form controls.

```
<!-- Copyright (c) 2013, 2017, Oracle and/or its affiliates. All rights reserved. -->
<div class="oj-flex">
    <!-- To ensure uniqueness, generated unique id for label and input tags. -->
    <!-- Example: Generated Id will be in the format editor_<number> -->
    <div class="oj-flex-item customLabel customEditorLabel">
        <label data-bind="attr:{for: editorId}"><span data-bind="text : labelText"/
></label>
    </div>
    <div class="oj-flex-item editorInput">
        <!-- invalidComponentTracker is required in case of any validation needed for
 editor. So if an editor has validations defined, then
          add  invalidComponentTracker: $parents[3].tracker as a property of
 ojComponent-->
       <input title="" data-bind="attr : {'id': editorId ,'aria-label' :
 editorBundle.getMessage('editor.aria.label')}, ojComponent: { component:
 'ojInputText', value : property().value , disabled : !isEnabled() || isReadOnly(),
                                                                        help :
{definition :helpText},

 rootAttributes: {style:'max-width:68.5rem'},

 displayOptions: {messages: 'inline'},

 optionChange: handleInputKey.bind($data),
                                                                        validators :
 [
                                                                        {
                                                                           type :
 'regExp',
                                                                           options :
 {

   pattern : requiredPattern

  }
                                                                        }
                                                                        ],

 invalidComponentTracker: $parents[3].tracker
                                                                        }"/><br>
       <span data-bind="text: bottomLabel" class="bottomLabel"></span>
   </div>
   </div>
</div>
```

6. Once a template is defined, it can be loaded using the TemplateEngine JavaScript module provided by the framework.

   In the following example, `Discover/config/import/editors/custom/StringEditor/editor.js`, the template can be loaded by specifying a path relative to the editors folder. The template can also be resolved by specifying a path relative to `editor.js`, for example, `require("template!./editor.html")`.

```
var TemplateEngine  = require("core/TemplateEngine");
...
var TEMPLATE_TEXT = require("template!editors/custom/StringEditor/editor.html");
...
```

```
var StringEditor = function(pConfig) {
   var self = this;
   ...
   self.template(TemplateEngine.addSource(TEMPLATE_TEXT));
}
...
```

7. Define a `_.json` with an `ecr:type` of `editor`. If you need a configuration for this editor, that can be defined in the `config` property. See the following example of `Discover/config/import/editors/custom/StringEditor/_.json`:

```
{
  "ecr:type": "editor",
    "config":{
      "pattern":"[A-Za-z0-9_]{5,}"
   }
}
```

8. If you need to define a custom CSS for your editor, place at the following location: `<app dir>/editors/custom/<editor name>/css/<name>.css` file.

9. If you created a custom CSS in the previous step, add a link to the CSS in your custom editor. For example:

```
<link rel="stylesheet" href="sites/Discover/editors/custom/StringEditor/css/
styles.css"/>
<div class="oj-flex">
 <div class="editorLabel">
        <label data-bind="attr:{for: uniqueId}"><span data-bind="text : labelText"/
></label>
    </div>
    <div class="editorInput">
<input>
</div>
<div>
```

Example: StringEditor editor.js file

Here is an example of the completed `editor.js` for the String Editor used in the previous topics:

```
// Copyright (c) 2013, 2016, Oracle and/or its affiliates. All rights reserved.
define(["app/xmgr/view/editors/Editor", "core/TemplateEngine", "template!editors/
custom/StringEditor/editor.html", "ojs/ojcomponents", "ojs/ojknockout", "ojs/
ojinputtext"], function (Editor, TemplateEngine, EditorTemplate) {
  "use strict";

  var TOOL_TIP = "tooltip";

  /**
   * @alias StringEditor
   * @classdesc Special Editor for editing properties whose value is a String.
   * Editor allows values only those which are in a specific pattern.
   * This pattern can be configured using a regular expression in editor's
   * configuration file i.e. _.json.
   * @extends app/xmgr/view/editors/Editor
   * @constructor
```

```
 */
var StringEditor = function (pConfig) {
  var self = this;
  Editor.call(self, pConfig);
  /**
   * Pattern allowed to enter through editor. Default is null.
   * @type {null}
   */
  self.requiredPattern = pConfig.editorConfig.pattern;

  self.template(TemplateEngine.addSource(EditorTemplate));
};

StringEditor.prototype = Object.create(Editor.prototype);

/**
 * This method initializes the editor instance. This should not be called explicitly,
 * as framework internally calls the same as part of editor's life cycle.
 * Should call initialize method of it's super class, passing the configuration
object.
 * Will override this method to instantiate and initialize any custom controllers,
 * subscribe for any custom notifications etc.
 * @param pTemplateConfig Configuration object that holds configuration specified in
cartridge template for this editor.
 * @param pContentItem ContentItem object which has the property that this editor
works with.
 */
StringEditor.prototype.initialize = function (pTemplateConfig, pContentItem) {
  var self = this;
  pTemplateConfig[TOOL_TIP] = self.editorBundle.getMessage("tooltip.information",
self.requiredPattern);
  Editor.prototype.initialize.call(this, pTemplateConfig, pContentItem);
};

/**
 * This method binds the property object, whose value will be edited through this
editor.
 * This should not be called explicitly, as framework internally calls the same as
part of
 * editor's life cycle. Based on the type that you give for a property in cartridge
template,
 * framework wraps property in an appropriate property class :
 * Allowed types are String, Boolean, Item, List, ContentItem, ContentItemList.
Corresponding to these types framework wraps property using
 * StringProperty, BooleanProperty, JSONProperty (for both Item and List),
ContentItemProperty, ContentItemListProperty.
 * Can override this method, if there is a need to a wrap property in a custom
extension to the Property API.
 * For example in case of NumericStepperEditor, StringProperty can be wrapped using
NumberProperty in editorReady.
 * If overridden, should call editorReady method of it's super class, passing the
property object.
 */
StringEditor.prototype.editorReady = function (pProperty) {
  var self = this;
  Editor.prototype.editorReady.call(self, pProperty);
};

return StringEditor;
```

```
});
```

Registering custom editors

You register new editors with your Experience Manager application by adding your custom editor to the editor registry of the application. The registry is located in the `<app dir>/config/import/editors` directory.

When you complete your custom editor, you must place it in the `<app dir>/config/import/editors` directory in a folder with the custom editor's name. For example, if you want to register a custom editor named TextAreaEditor, then you must create a folder named `<app dir>/config/import/editors/TextAreaEditor`. Place a JSON file the with the `ecr:type` of **editor** in this folder. If a custom editor needs any configuration, that can be specified in the JSON file, too. The folder must also contain the following files:

- `editor.js` - the knockout view model for the editor.
- `editor.html` - the template file that defines user interface layout for the editor using one or more OJET components.

If the custom editor has the same name as an existing editor in your application, you can create a unique namespace for your custom editor, see *Overriding an existing editor with a custom editor*.

To register a custom editor:

1. Run the following command to export the editor registry of your deployed application: `runcommand.bat|sh IFCR exportContent editors <directory>`, where `<directory>` indicates the directory on the file system to which the editors registry should be exported.

   For example:

   ```
   runcommand.bat IFCR exportContent editors D:\backup\editors
   ```

2. Navigate to the directory where you exported the editor registry.

   For example:

   ```
   D:\backup\editors
   ```

3. Create a folder for your custom editor. Remember the folder must have the custom editor's name.

   For example:

   ```
   D:\backup\editors\TextAreaEditor
   ```

4. Add a JSON file, `_.JSON` with your editor's configuration to the folder. The `ecr:type` must be **editor**; `config` is optional.

   For example:

   ```
   {
       "ecr:type":"editor",
   ```

```
}
```

5. Add the `editor.js` and `editor.html` files for your editor to the folder.
6. Run the following command to import the updated editor registry of your deployed application: `runcommand.bat|sh IFCR importContent editors <directory>`, where `<directory>` indicates the directory on the file system from which editors registry should be imported.

   For example:

```
runcommand.bat IFCR importContent editors D:\backup\editors
```

Overriding an existing editor with a custom editor

If the custom editor that you need to add to your editor registry has the same name as an existing editor in your application, you can create a unique namespace for your custom editor. This lets you override an existing editor in the Experience Manager installation with a custom editor.

You do not need to edit every occurrence of the existing editor in cartridge templates to refer to the custom editor. Once the existing Experience Manager editor is overridden with a custom editor, all the mappings that exist in cartridge templates work as if the custom editor has been mapped in place of an editor in the Experience Manager installation.

Follow these steps to override an existing editor.

1. Run the following command to export the editor registry of your deployed application: `runcommand.bat|sh IFCR exportContent editors <directory>`, where `<directory>` indicates the directory on the file system to which the editors registry should be exported.

   For example:

```
runcommand.bat IFCR exportContent editors D:\backup\editors
```

2. Navigate to the `D:\backup\editors` directory.
3. If you have not done so already, create a namespace folder for your custom editors that have the same names as existing editors in your registry.

   For example, `\editors\custom`
4. In the namespace folder, create a folder for your custom editor. Remember the folder must have the custom editor's name.

   For example if you want to override StringEditor in Experience Manager (`editors\StringEditor`) with a custom StringEditor then create a folder named `editors\custom\StringEditor`
5. Add a JSON file, `_.JSON` with your editor's configuration to the folder. The `ecr:type` must be **editor**; `config` is optional.

   For example:

```
{
```

```
    "ecr:type":"editor",
}
```

6. Add the `editor.js` and `editor.html` files for your editor to the custom editor folder.
7. Copy the `editor.js`, `editor.html`, and `_.json` files from the custom editor folder `editors/<namespace>/<custom editor>` and paste them into the existing editor folder `editors/<editor>`.

   For example, copy the `editor.js`, `editor.html`, and `_.json` files from the `<app dir>/config/import/editors/custom/StringEditor` folder to the `<app dir >/config/import/editors/StringEditor` folder.
8. Run the following command to import the updated editor registry of your deployed application: `runcommand.bat|sh IFCR importContent editors <directory>`, where `<directory>` indicates the directory on the file system from which editors registry should be imported.

   For example:

```
runcommand.bat IFCR importContent editors D:\backup\editors
```

9. Clear the browser cache and restart Experience Manager. The system begins using the custom editor in place of the existing editor.

   For example, the system starts picking custom StringEditor in place of the existing StringEditor.

Reusing custom editors across multiple applications

You can use a custom editor that you have developed for an application in multiple applications.

Reuse the custom editor by copying an editor's folder in the editor registry from one application to one or more other applications. Next, import the updated editor registries into their applications.

Follow these steps to reuse custom editors.

1. Navigate to the editor-registry of the application that contains the custom editor that you want to reuse: `<app-one dir>/config/import/editors/<custom editor>`.
2. Copy the folder of the custom editor
3. Run the following command to export the editor registry of the deployed application to which you want to add the custom editor: `runcommand.bat|sh IFCR exportContent editors <directory>`, where `<directory>` indicates the directory on the file system to which the editors registry should be exported.

   For example:

```
runcommand.bat IFCR exportContent editors D:\backup2\editors
```

4. Navigate to the `D:\backup2\editors` directory.
5. Paste the folder of the custom editor that you previously copied into the editors directory.
6. Run the following command to import the updated editor registry of the application to which you added the custom editor: : `runcommand.bat|sh IFCR importContent editors <directory>`, where `<directory>` indicates the directory on the file system from which the editors registry should be imported.

For example:

```
runcommand.bat IFCR importContent editors D:\backup2\editors
```

7. Map the custom editor in the required cartridge templates of the application.

About creating and uploading a cartridge template

To use your custom editors in Experience Manager, you need to create and upload a cartridge template that includes the new editors. You can choose to create a new cartridge, or to modify an existing cartridge template.

After creating or modifying a cartridge to include your custom editors, you must upload it to your application. You can accomplish this by moving the template to your deployed application's `\config\import\templates` directory and running the `control\set_templates` batch or shell script.

About custom editors in multiple locales

If your implementation supports multiple locales, you can localize your custom editors.

You must do the following:

- Create resource properties files that contain localized strings
- Retrieve localized content from from the resource property files for your custom editors.

Creating resources properties files

You can create resource property files for each locale for storing localized strings.

Each resource property file name must follow this format: `Resources_<locale>.properties` where `<locale>` is the ISO language code. For example `Resources_fr.properties` indicates that French values are stored in it. Place these files in a locales folder for your custom editor: `<app dir>\config\import\editors\<editor name>\locales` or `<app dir>\config\import\editors\custom\<editor name>\locales`.

Here is an example of the contents of the `Resources_en.properties` file in the under `<app dir>/config/import/editors/custom/StringEditor/locales` directory.

```
...
sample.message = This message is loaded from resource bundle
error.message.summary = Incorrect format for the input text
error.message.detail = Value must match this pattern: {0}
...
```

Retrieving localized content for your custom editor

Your custom editor obtains localized content using the jQuery.i18n.properties plugin to retrieve the content from the resource properties file.

When a custom editor has property-based resource bundles in it's locales folder, the system loads the bundle corresponding to the Workbench user's locale during editor instantiation and makes the localized content available in a data attribute, `editorBundle`, defined within the editor instance. This bundle provides a method, `getMessage(resourceKey, arguments...)`, to load localized text.

To retrieve localized content from resource properties files, follow these steps

1. Load localized messages for the custom editor, using the following syntax in your `editor.js`:

```
var message1 = this.editorBundle.getMessage('<resource-key-to-localized-message>');

// Loading a parameterized message.
var message2 = this.editorBundle.getMessage('<resource-key-to-localized-message>',
 <param1>, <param2>);
```

For example:

```
var message1 = this.editorBundle.getMessage('error.message.summary');

// Loading a parameterized message.
var message2 = this.editorBundle.getMessage('error.message.detail', '[a-b0-9]\{6}');
```

2. Load localized messages for the custom editor, use the following syntax in your `editor.html`:

```
<label data-bind="text: editorBundle.getMessage('editor.proeprty.label')"/>

// Loading a parameterized message.
<span data-bind="text: editorBundle.getMessage('error.message.detail', '[a-
b0-9]\{6}')"/>
```

6 Template Property and Editor Reference

This section describes how to define basic content properties and associated editing interfaces in Experience Manager templates.

Experience Manager editors mapping reference

The following editors are included in the Oracle Experience Manager:

Editor	Property Type	Functionality
`BooleanEditor`	`Boolean`	Displays as a checkbox that the content administrator selects or de-selects. Optionally, the editor may be set to a read-only state.
`BoostBuryEditor`	`List`	Displays as a three-pane, drag-and-drop interface consisting of a central pane that lists available dimension refinements, a left pane for boosted refinements, and a right pane for buried refinements. The content administrator can filter the list of available dimensions by searching against a text string. The editor populates two `List` properties, one for boosted dimension refinements and one for buried dimension refinements.

Editor	Property Type	Functionality
`BoostBuryRecordEditor`	`List`	Displays as two panes, Boosted Records and Buried Records, each with an Edit List button that launches the Select Records dialog. The content administrator uses the Select Records dialog to populate the lists of boosted and buried records. The editor populates two `List` properties, one for boosted records and one for buried records.
`ChoiceEditor`	`String`	Displays as a dropdown with an optional default value. The content administrator selects from a set of pre-defined values.
`DimensionListEditor`	`List`	Displays as two panels, one with a list of available dimensions and one with a list of selected dimensions. The content administrator can drag values back and forth between the two lists.
`DimensionSelectorEditor`	`String`	Displays as a dropdown. The content administrator selects a value from the list of available dimensions retrieved from the MDEX Engine. The editor populates two `String` properties, one for the dimension name and one for the ID.
`DimvalListEditor`	`List`	Displays as two panels, one with a list of available dimension refinements and one with a list of selected refinements. The content administrator can drag values back and forth between the two lists. Additionally, the list of available refinements includes a search box for finding specific refinements in a large data set.
`DynamicSlot Editor`	`String`	Displays as a drop-down list for specifying a valid content collection, and a numeric stepper for setting the evaluation limit for that collection.
`GuidedNavigationEditor`	`ContentItemList`	Displays as a button for launching the Generate Guided Navigation wizard, which allows a content administrator to select and order a set of dimensions in order to create multiple Refinement Menu cartridges at once.
`ImagePreview`	(None)	Displays an image from a specified URL.

Editor	Property Type	Functionality
`LinkBuilderEditor`	`Item`	Displays two radio buttons, one for specifying an External link via a text field, and one for specifying an Internal (Relative) link. The content administrator specifies a relative link by selecting a servlet from a dropdown list, then launching the Select Records dialog to navigate to a specific record or a navigation state. The editor populates a class `com.endeca.infront.cartridge.model.LinkBuilder` item property. For more information, see "Adding a Link Builder."
`MediaEditor`	`Item`	Displays as a Media URL field, with an associated preview box and Select and Clear buttons for launching the media editor or clearing the current URL. The content administrator can browse through media in the configured source repository, and generate a link to a selected asset.
`NumericStepperEditor`	`String`	Displays as a one-line text field with a pair of arrow buttons for increasing or decreasing the value by a set amount. The content administrator inputs or adjusts the value to any number within the minimum and maximum boundaries defined in the editor.
`RichTextEditor`	`String`	Displays as a text area with a configurable formatting toolbar. The content administrator enters arbitrary string values and can include markup to add text formatting and hyperlinks.
`SortEditor`	`Item`	Displays as a dropdown. The content administrator selects a sort order from those configured in the editor. The editor includes multiple class `com.endeca.infront.navigation.model.SortOption` item properties that each specify an available sort option. For more information, see "Adding a Sort editor."
`SpotlightSelectionEditor`	`Item`	Displays as a button that launches the Select Records dialog and allows the content administrator to select the navigation state or list of records that populates a class `com.endeca.infront.cartridge.RecordSpotlightSelection` record selection property.
`StringEditor`	`String`	Displays as a text field or text area. The content administrator enters arbitrary string values. Optionally, the editor may be set to a read-only state to display a fixed, default value.

Related links

- Basic content properties (page 212)
- Complex property editors (page 226)

Editor label configuration reference

All editors share a set of common attributes that can be used to configure the appearance of the editor in Experience Manager.

When adding an editor to a template, you can configure its appearance by setting the following attributes:

Attribute	Description
`label`	This attribute enables you to specify a more descriptive label for the editor in Experience Manager. If no label is specified, the value of the associated `propertyName` is used by default.
`labelPosition`	The position of the label text. Valid values are `"left"` (the default) and `"top"`.
`bottomLabel`	This attribute allows you to specify a descriptive label that appears below the editor.
`tooltip`	This attribute allows you to specify mouseover text for the editor.

Basic content properties

Content items properties must be one of several basic types. All configuration models are composed of the same primitive property types.

The basic content property types are:

- `String`
- `Boolean`
- `List`
- `Item`

The following example shows a several properties of various types.

```
"typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"},
        "recordsPerPage": {"@propertyType": "String"},
        "relRankStrategy": {"@propertyType": "String"},
        "sortOption": {"@propertyType": "Item"}
    }
```

Adding a string property

String properties are very flexible and can be used to specify information such as text to display on a page, URLs for banner images, or meta keywords for search engine optimization.

To add a string property to a template:

1. In the`typeInfo` section, insert a `String` element inside a `@propertyType` element.
2. In `defaultContentItem`, specify the default value for the property for the content for each element that you specified in `typeInfo`.

The following example shows a variety of string properties:

```
{
    "@description": "${template.description}",
    "@group": "Navigation",
    "ecr:createDate": "2016-09-12T17:33:58.404+05:30",
    "@thumbnailUrl": "thumbnail.jpg",
    "ecr:type": "template",
    "defaultContentItem": {
        "lessLinkText": "Show Less Refinements...",
        "numRefinements": "10",
        "@name": "Dimension Navigation",
        "dimensionId": "",
        "moreLinkText": "Show More Refinements...",
        "maxNumRefinements": "200",
        "sort": "default",
        "showMoreLink": false,
        "dimensionName": ""
    },
<!-- additional elements omitted from this example -->
   "typeInfo": {
        "boostRefinements": {"@propertyType": "List"},
        "buryRefinements": {"@propertyType": "List"},
        "dimensionId": {"@propertyType": "String"},
        "dimensionName": {"@propertyType": "String"},
        "lessLinkText": {"@propertyType": "String"},
        "maxNumRefinements": {"@propertyType": "String"},
        "moreLinkText": {"@propertyType": "String"},
        "numRefinements": {"@propertyType": "String"},
        "showMoreLink": {"@propertyType": "Boolean"},
        "sort": {"@propertyType": "String"}
    }
}
```

Adding a string editor

You add a string editor to enable configuration of string properties. The string editor displays in the Experience Manager interface as a text field or text area depending on the configuration.

String editors enable content administrators to supply arbitrary values for a string property. If you want to constrain the input to a specific enumeration of values, use a choice editor.

To add a string editor to a template:

1. Insert an `editors/StringEditor` element within `editors/DefaultEditorPanel`.
2. Specify label attributes and additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The `name` of the string property that this editor is associated with. This property must be declared in the same template as the string editor.
`label`	The label for the editor.
`enabled`	If set to `false`, this attribute makes the property read-only so that the value of the property displays in the Content Details Panel in Experience Manager, but cannot be edited. Set this to `false` only if you specify a default value in the definition of the string property. Editors are enabled by default.

The following example shows a variety of editing options for string properties:

```
{
    "@description": "${template.description}",
    "@group": "Navigation",
    "ecr:createDate": "2016-09-12T17:33:58.404+05:30",
    "@thumbnailUrl": "thumbnail.jpg",
    "ecr:type": "template",
    "defaultContentItem": {
        "lessLinkText": "Show Less Refinements...",
        "numRefinements": "10",
        "@name": "Dimension Navigation",
        "dimensionId": "",
        "moreLinkText": "Show More Refinements...",
        "maxNumRefinements": "200",
        "sort": "default",
        "showMoreLink": false,
        "dimensionName": ""
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
<!-- additional elements omitted from this example -->
            {
                "editor": "editors/StringEditor",
```

```
                "propertyName": "moreLinkText",
                "label": "${property.moreLinkText.label}",
                "enabled": true
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "lessLinkText",
                "label": "${property.lessLinkText.label}",
                "enabled": true
            }
        ]
<!-- additional elements omitted from this example -->
    }
    "typeInfo": {
        "boostRefinements": {"@propertyType": "List"},
        "buryRefinements": {"@propertyType": "List"},
        "dimensionId": {"@propertyType": "String"},
        "dimensionName": {"@propertyType": "String"},
        "lessLinkText": {"@propertyType": "String"},
        "maxNumRefinements": {"@propertyType": "String"},
        "moreLinkText": {"@propertyType": "String"},
        "numRefinements": {"@propertyType": "String"},
        "showMoreLink": {"@propertyType": "Boolean"},
        "sort": {"@propertyType": "String"}
    }
}
```

Note

Neither Experience Manager nor the Assembler applies HTML escaping to strings. This enables content administrators to specify HTML formatted text in Experience Manager and have it rendered appropriately. If you intend to treat a string property as plain text, be sure to add HTML escaping to your application logic in order to avoid invalid characters and non-standards-compliant HTML.

Adding a choice editor

A choice editor enables the user to select from predefined string values for a property that are presented in a drop-down list. Choice editors affect the value of a string property.

To add a choice editor:

1. Insert an `editors/ChoiceEditor` element within `editors/DefaultEditorPanel`.
2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The `name` of the string property that this editor is associated with. This property must be declared in the same template as the choice editor.
`editable`	If set to `true`, this attribute allows Experience Manager users to specify custom string values. By default, choice editors are not editable.

Attribute	Description
`enabled`	If set to `false`, the choice editor displays in Experience Manager but the value cannot be changed by the user. By default, choice editors are enabled.
`prompt`	Specifies a custom prompt. The default prompt is an empty string. It is enabled only when editable is set to `true`.
`tooltip`	If present, specifies optional help text to display in a tool tip window. The default behavior is no tool tip.
`width`	The width, in pixels, of the choice editor. By default, the width of the editor adjusts to fit the longest choice in the editor. Use this attribute if you want to set a fixed width for the editor.

3. Insert an `choices` element within the `editors/ChoiceEditor element`.

4. Specify one or more menu options for the choice editor by adding elements that takes the following attributes:

Attribute	Description
`value`	Required. The string value to assign to the associated property if this choice is selected.
`label`	This attribute allows you to specify a more descriptive label for this option in the drop down list. If no label is specified, the `value` is used by default. You must either specify a `label` for all of the choices or none of them. You cannot have labels for some choices and not others. **Note** If you choose to make a choice editor editable (so that users can enter arbitrary strings), you should not use the `label` attribute for choices. Instead, the choice editor should display the raw value of the string so that users entering custom values can see the expected format of the string property.

5. Optionally, set a default value in the corresponding `defaultContentItem` property.

   For example,:

```
"defaultContentItem": {
```

```
        "sort": "default",
```

Note

Ensure that the default value for the property is one of the options defined for the choice editor in a element.

The following example shows a choice editor configured with a default value.

```
{
    "@description": "${template.description}",
    "@group": "Navigation",
    "ecr:createDate": "2016-09-12T17:33:58.404+05:30",
    "@thumbnailUrl": "thumbnail.jpg",
    "ecr:type": "template",
    "defaultContentItem": {
        "lessLinkText": "Show Less Refinements...",
        "numRefinements": "10",
        "@name": "Dimension Navigation",
        "dimensionId": "",
        "moreLinkText": "Show More Refinements...",
        "maxNumRefinements": "200",
        "sort": "default",
        "showMoreLink": false,
        "dimensionName": ""
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
<!-- additional elements omitted from this example -->
            {
                "editor": "editors/ChoiceEditor",
                "propertyName": "sort",
                "label": "${property.sort.label}",
                "choices": [
                    {
                        "label": "${property.sort.default.label}",
                        "value": "default"
                    },
                    {
                        "label": "${property.sort.static.label}",
                        "value": "static"
                    },
                    {
                        "label": "${property.sort.dynRank.label}",
                        "value": "dynRank"
                    }
                ]
            },
        ]
<!-- additional elements omitted from this example -->
    }
    "typeInfo": {
        "boostRefinements": {"@propertyType": "List"},
        "buryRefinements": {"@propertyType": "List"},
        "dimensionId": {"@propertyType": "String"},
        "dimensionName": {"@propertyType": "String"},
        "lessLinkText": {"@propertyType": "String"},
```

```
        "maxNumRefinements": {"@propertyType": "String"},
        "moreLinkText": {"@propertyType": "String"},
        "numRefinements": {"@propertyType": "String"},
        "showMoreLink": {"@propertyType": "Boolean"},
        "sort": {"@propertyType": "String"}
    }
}
```

About numeric properties

Numeric properties should be specified as string properties in the template.

Properties that are expected to have numeric values can be associated with editors that are designed to work with numbers. These editors guarantee that the property is assigned a numeric value.

Adding a numeric stepper

A numeric stepper enables content administrators to select a numeric value from a set of possible values by stepping through values or typing into an input field.

The numeric stepper provides a single-line input text field and a pair of arrow buttons for stepping through values. If a user enters number that is not a multiple of the `stepSize` property or is not in the range between the maximum and minimum properties, this property is set to the nearest valid value.

To add a numeric stepper to a template:

1. Insert an `editors/NumericStepperEditor` element within `editors/DefaultEditorPanel`.
2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The `name` of the string property that this editor is associated with. This property must be declared in the same template as the string editor.
`enabled`	If set to false, the numeric stepper editor displays in Experience Manager but the value cannot be changed by the user. By default, numeric stepper editor are enabled.
`width`	The width, in pixels, of the editor. The default width is 60.
`height`	The height, in pixels, of the editor. The default height is 24.
`minValue`	The minimum value of the property bound to this editor. The `minValue` can be any number, including a fractional value. The default minimum value is 0.

Attribute	Description
maxValue	The maximum value of the property bound to this editor. The maxValue can be any number, including a fractional value. The default maximum value is 10.
stepSize	The increment by which the property value is increased or decreased when a user clicks on the up or down arrows. The value must be a multiple of this number. The default step size is 1.

The following example shows the configuration for a numeric stepper:

```
{
  "ecr:type":"template",
  "@group":"Navigation",
  "@description":"${template.description}",
  "@thumbnailUrl":"thumbnail.jpg",
  "typeInfo":{
<!-- additional elements omitted from this example -->
    "numRefinements":{
      "@propertyType":"String"
    },
<!-- additional elements omitted from this example -->
  },
  "defaultContentItem":{
    "@name":"Dimension Navigation",
    "dimensionName":"",
    "dimensionId":"",
    "sort":"default",
    "showMoreLink":false,
    "moreLinkText":"Show More Refinements...",
    "lessLinkText":"Show Less Refinements...",
    "numRefinements":"10",
    "maxNumRefinements":"200",
    "boostRefinements":[

    ],
    "buryRefinements":[

    ]
  },
  "editorPanel":{
    "editor":"editors/DefaultEditorPanel",
    "children":[
<!-- additional elements omitted from this example -->
      {
        "editor":"editors/NumericStepperEditor",
        "enabled":true,
        "label":"${property.numRefinements.label}",
        "maxValue":10000,
        "propertyName":"numRefinements"
      },
        ]
    }
<!-- additional elements omitted from this example -->
}
```

Adding a Boolean property

Boolean properties represent a true or false value and can be used to enable or disable features in your application.

To add a Boolean property to a template:

1. In `typeInfo` element, insert a `Boolean` element inside a `@propertyType` element.

```
"typeInfo": {
        "showMoreLink":{
          "@propertyType":"Boolean"
        }
    }
```

2. Specify the default value for the property.

```
 "defaultContentItem": {
        "showMoreLink":false
    },
```

Any value other than the string "`false`" (case insensitive) defaults to a value of `true`.

The following example shows the configuration of a Boolean property:

```
{
  "ecr:type":"template",
  "@group":"Navigation",
  "@description":"${template.description}",
  "@thumbnailUrl":"thumbnail.jpg",
  "typeInfo":{
    "dimensionName":{
      "@propertyType":"String"
    },
    "dimensionId":{
      "@propertyType":"String"
    },
    "sort":{
      "@propertyType":"String"
    },
    "showMoreLink":{
      "@propertyType":"Boolean"
    },
    "moreLinkText":{
      "@propertyType":"String"
    },
    "lessLinkText":{
      "@propertyType":"String"
    },
    "numRefinements":{
      "@propertyType":"String"
    },
    "maxNumRefinements":{
      "@propertyType":"String"
```

```
    },
    "boostRefinements":{
      "@propertyType":"List"
    },
    "buryRefinements":{
      "@propertyType":"List"
    }
  },
  "defaultContentItem":{
    "@name":"Dimension Navigation",
    "dimensionName":"",
    "dimensionId":"",
    "sort":"default",
    "showMoreLink":false,
    "moreLinkText":"Show More Refinements...",
    "lessLinkText":"Show Less Refinements...",
    "numRefinements":"10",
    "maxNumRefinements":"200",
    "boostRefinements":[

    ],
    "buryRefinements":[

    ]
  },
```

Adding a Boolean editor

A Boolean editor provides a checkbox for Experience Manager users to specify the value of a Boolean property.

To add a Boolean editor:

1. Insert a `editors/BooleanEditor` element within `editors/DefaultEditorPanel`.
2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The `name` of the Boolean property that this editor is associated with. This property must be declared in the same template as the Boolean editor.
`enabled`	If set to `false`, the checkbox displays in Experience Manager but the value cannot be changed by the user. By default, checkboxes are enabled.

The following example illustrates a checkbox Boolean editor:

```
{
  "ecr:type":"template",
  "@group":"Navigation",
  "@description":"${template.description}",
```

```
  "@thumbnailUrl":"thumbnail.jpg",
  "typeInfo":{
    <!-- additional elements omitted from this example -->
    "showMoreLink":{
      "@propertyType":"Boolean"
    },
    <!-- additional elements omitted from this example -->
  },
  "defaultContentItem":{
    "@name":"Dimension Navigation",
    "dimensionName":"",
    "dimensionId":"",
    "sort":"default",
    "showMoreLink":false,
    "moreLinkText":"Show More Refinements...",
    "lessLinkText":"Show Less Refinements...",
    "numRefinements":"10",
    "maxNumRefinements":"200",
    "boostRefinements":[

    ],
    "buryRefinements":[

    ]
  },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
           <!-- additional elements omitted from this example -->
            {
                "editor":"editors/BooleanEditor",
                "enabled":true,
                "label":"${property.showMoreLink.label}",
                "propertyName":"showMoreLink"
            },
<!-- additional elements omitted from this example -->
}
}
```

Adding an item property

A property can consist of a collection of properties (key-value pairs) of any valid type.

Because item properties can be used for a variety of purposes, Experience Manager does not include any generic editors for working with items. However, editors intended for specific purposes may store their values in item properties.

To add an item property to a template:

1. In `typeInfo` section, insert an `Item` element inside a `@propertyType` element.

```
  "typeInfo": {
           "sortOption": {"@propertyType": "Item"}
      }
```

2. In `defaultContentItemInsert`, add a `sortOption` element with a `sorts` element and specify the `@class` attribute with the fully qualified class name of the configuration model class that corresponding to this item property.

```
 "defaultContentItem": {
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
    }
```

3. Optionally, specify other attributes within the `sortOption` element.

Following is an example of a template that uses an item:

```
{
     "@description": "${template.description}",
     "@group": "MainContent",
     "ecr:createDate": "2016-09-12T17:33:58.542+05:30",
     "@thumbnailUrl": "thumbnail.png",
     "ecr:type": "template",
     "defaultContentItem": {
        "@name": "Results List",
        "relRankStrategy": "",
        "boostStrata": [],
        "buryStrata":[],
        "recordsPerPage": "10",
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
        },
    <!-- additional elements omitted from this example -->
  "typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"},
        "recordsPerPage": {"@propertyType": "String"},
        "relRankStrategy": {"@propertyType": "String"},
        "sortOption": {"@propertyType": "Item"}
    }
```

Adding a list property

A property can consist of an ordered list of strings, Booleans, items, or other lists.

Because lists can be used for a variety of purposes, Oracle Guided Search does not include any generic editors for working with lists. However, editors intended for specific purposes may store their values in list properties.

To add a list property to a template:

1. Insert a `List` element inside a `@propertyType"` element in `typeInfo`.
2. Optionally, specify a default value by inserting either `String`, `Boolean`, `List`, or `Item` elements in `defaultContentItem`.

Following is an example of a template that uses lists both with and without default values:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:58.542+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Results List",
        "relRankStrategy": "",
        "recordsPerPage": "10",
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
    },
<!-- additional elements omitted from this example -->
    "typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"},
        "recordsPerPage": {"@propertyType": "String"},
        "relRankStrategy": {"@propertyType": "String"},
        "sortOption": {"@propertyType": "Item"}
    }
}
```

Adding a group label

In the Experience Manager interface, group labels can serve as a visual cue that several properties are related.

Group labels are only used to provide additional context in the editing interface of Experience Manager and do not affect rendering in the front-end application. Group labels are optional.

One use of group labels is to give the content administrator information about properties that they need to configure the cartridge. For example, if a template defines properties that are required in order to render the

content properly, you can indicate these with a descriptive group label so that the content administrator can easily identify the required fields in Experience Manager.

The editor panel in Experience Manager includes a default heading of "Section settings." This heading includes the required Name field and the read-only group of a template, as well as any properties that are defined before the first group label.

To add a group label to the editor panel:

1. Insert the `GroupLabel` element inside `editors/DefaultEditorPanel` as in the following example:

```
{
    "@description": "${template.description}",
    "@group": "SecondaryContent",
    "ecr:createDate": "2016-09-12T17:33:58.290+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Spotlight Records",
        "maxNumRecords": "10",
        "seeAllLinkText": "",
        "showSeeAllLink": false,
        "title": "Featured Items",
        "recordSelection": {"@class":
 "com.endeca.infront.cartridge.RecordSpotlightSelection"}
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.spotlight.label}"
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "title",
                "label": "${property.title.label}",
                "enabled": true            },
            {
                "editor": "editors/SpotlightSelectionEditor",
                "maxNumRecords": "maxNumRecords",
                "propertyName": "recordSelection",
                "seeAllLinkText": "seeAllLinkText",
                "showSeeAllLink": "showSeeAllLink",
                "label": "${property.recordSelection.label}"
            }
        ]
    },
    "typeInfo": {
        "maxNumRecords": {"@propertyType": "String"},
        "recordSelection": {"@propertyType": "Item"},
        "seeAllLinkText": {"@propertyType": "String"},
        "showSeeAllLink": {"@propertyType": "Boolean"},
        "title": {"@propertyType": "String"}
    }
}
```

`GroupLabel` is an empty tag that allows you to specify the label text with the `label` attribute.

Complex property editors

This section describes editors that are designed for specific aspects of feature configuration.

About the microbrowser

The microbrowser is used in several editors in the core cartridges to enable a content administrator to specify a set of records. It is deprecated in this release; use the Select Records dialog instead.

The microbrowser is a lightweight search and Guided Navigation application that enables a content administrator to browse to a particular location in the data set (which may include search terms, dimension refinements, or a combination of both). The content administrator can then do one of two things:

- Save the current filter state to designate a dynamic set of records.
- Select specific records from that filter state (or other filter states) to designate a set of specific featured records.

An instance of a microbrowser is usually bound to a list property, which contains items that represent either refinements or record IDs.

The microbrowser communicates with the MDEX Engine to retrieve search and navigation results.

Note

In order to enable the microbrowser, ensure that you have enabled communication between Experience Manager and the MDEX Engine. For instructions, see "Communicating with the MDEX Engine" in the *Tools and Frameworks Installation Guide*.

Data service configuration reference

The microbrowser uses a data service to access MDEX Engine information. By default, the service is configured to provide relevant record properties for the Discover Electronics reference application.

The data service is configured in the file `<app dir>config\import\configuration\tools\xmgr\_.json`, as shown below:

```
{
    "name": "dataservice",
    "host": "myhost.mydomain.com",
    "port": "15002",
    "recordSpecName": "common.id",
    "aggregationKey": "product.code",
    "recordFilter": "",
    "wildcardSearchEnabled": false,
    "recordNameField": "",
    "fields": {
        "product.id": "",
        "product.name": "plain",
        "product.price": "currency",
        "product.short_desc": ""
```

```
    }
}
```

It specifies the following:

Key	Value
name	The name of the service, "dataservice".
`host`	The hostname or IP address of your MDEX Engine server. By default, this is populated with the same host as the authoring MDEX Engine when you deploy the Discover Electronics reference application and run the `initialize_services` script.
`port`	The port that the MDEX Engine server listens on. By default, this is populated with the same port as the authoring MDEX Engine.
`recordSpecName`	The dimension used as the record specifier. This must be a unique identifier.
`aggregationKey`	Optional. Enables aggregated records mode in the microbrowser, using the specified property or dimension as the aggregation key when displaying and sorting records. All records with the same value in the selected dimension or property are treated as a single record.
`recordFilter`	Optional. The property used to filter records for record boost and bury.
`wildcardSearchEnabled`	Optional. Wildcard search is enabled by default. If your configuration does not index dimensions by wildcard index, you must explicitly set this property to `false`.
`recordNameField`	Optional. The property that should be used to represent the name of a record.
`fields`	Each key in the array of key/value pairs specifies a property or dimension to display as a column in the microbrowser. Optionally, you may specify a formatting value from among the following: • `plain` — no formatting. Used as the default if no format value is present. • `currency` — adds a dollar ($) symbol before the value. • `integer` — removes the decimal point and any trailing digits, if present. This setting does not round the integer value. • `html` — attempts to handle markup tags within the content returned from the MDEX Engine.

Running `<app dir>\control\set_editors_config` pushes changes to the Discover Electronics reference application.

About the Select Records dialog

The Select Records dialog is used in several editors in the core cartridges to enable a content administrator to specify a set of records.

The Select Records dialog is a lightweight search and Guided Navigation application that enables a content administrator to browse to a particular location in the data set (which may include search terms, dimension refinements, or a combination of both). The content administrator can then do one of two things:

- Save the current filter state to designate a dynamic set of records.
- Select specific records from that filter state (or other filter states) to designate a set of specific featured records.

An instance of a Select Records dialog is usually bound to a `<List>` property in a cartridge template, which contains `<Item>` properties that represent either dimension refinements or record IDs. The dialog communicates with the MDEX Engine to retrieve search and navigation results.

> **Note**
>
> In order to enable the Select Records dialog, ensure that you have enabled communication between Experience Manager and the MDEX Engine. For instructions, see "Communicating with the MDEX Engine" in the *Tools and Frameworks Installation Guide*.

The following editors launch the Select Records dialog:

- Link Builder editor
- Boost-Bury Record editor
- Spotlight Selection editor

Select Records data service configuration reference

The Select Records dialog in Experience Manager communicates with the MDEX Engine through a configurable data service. By default, the service is configured to provide relevant record properties for the Discover Electronics reference application.

The service endecaBrowserService in configured in the file, `<app dir>\config\import\configuration \tools\xmgr\_.json`, as shown below:

```
{
    "name" : "endecaBrowserService"
    "host": "myhost.mydomain.com",
    "port": "15002",
    "recDisplayNameProp" : "product.name",
    "recSpecProp": "common.id",
    "recAggregationKey": "product.code",
    "recFilter": "",
    "recImgUrlProp" : "product.img_url_thumbnail",
    "recDisplayProps": [ "product.name", "product.price", "product.short_desc" ],
    "textSearchKey": "All",
    "textSearchMatchMode" : "ALLPARTIAL"
}
```

It specifies the following:

Key	Value
name	The name of the service, "endecaBrowserService".
`host`	The hostname or IP address of your MDEX Engine server. By default, this is populated with the same host as the authoring MDEX Engine when you deploy the Discover Electronics reference application and run the `initialize_services` script.
`port`	The port that the MDEX Engine server listens on. By default, this is populated with the same port as the authoring MDEX Engine.
`recDisplayNameProp`	The dimension used as the record display name in the editor that launches the dialog.
`recSpecProp`	The dimension used as the record specifier. This must be a unique identifier.
`recAggregationKey`	Optional. Enables aggregated records mode in the Select Records dialog using the specified property or dimension as the aggregation key when displaying and sorting records. All records with the same value in the selected dimension or property are treated as a single record.
`recFilter`	Optional. The property used to filter records for record boost and bury.
`recImgUrlProp`	Optional. The property used to retrieve the URL for the record thumbnail image.
`recDisplayProps`	An array of record properties to display in the dialog.
`textSearchKey`	Optional. Specifies the search key to apply to text searches in the Select Records dialog.
`textSearchMatchMode`	Optional. Specifies the match mode to apply to text searches in the Select Records dialog.

You can modify these values as necessary for your own application. Running `<app dir>\control \set_editors_config` pushes changes to the Discover Electronics reference application.

About the Dynamic Slot editor

The Dynamic Slot editor enables the content administrator to configure a section of an application page at query time by specifying one or more folders from which to return content.

The editor has no associated template configuration, although it launches a configuration dialog in Experience Manager. When the content administrator edits the cartridge in Experience Manager, the editor queries the Endeca Configuration Repository for a list of folders. These results are refined based on the template group or template ID restrictions entered by the content administrator.

Creating a cartridge template with a dynamic slot

You should configure a separate cartridge template for each template group that requires dynamic slot functionality.

To create a cartridge template with a dynamic slot:

1. Insert a `typeInfo` that includes the following properties:

   - `ruleLimit`
   - `contentPaths` — Include a nested `List` element.
   - `templateTypes` — Include a nested `List` element.
   - `templateIds` — Include a nested `List` element.

   For example:

```
"typeInfo": {
        "contentPaths": {"@propertyType": "List"},
        "ruleLimit": {"@propertyType": "String"},
        "templateIds": {"@propertyType": "List"},
        "templateTypes": {"@propertyType": "List"}
    }
```

   These properties are sent in as configuration to a `ContentSlotConfig` object that dynamically populates the page with a suitable content item. For more information, see About ContentInclude and ContentSlotConfig objects (page 9).

2. Add any default values to the `defaultContentItem`.

   For example:

```
"defaultContentItem": {
        "@name": "Secondary Content Slot",
        "ruleLimit": "1",
        "templateTypes": ["SecondaryContent"]
    }
```

3. In the `EditorPanel`, insert an `editors/DynamicSlotEditor` element within a `editors/DefaultEditorPanel`:

```
"editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [{"editor": "editors/DynamicSlotEditor"}]
    }
```

4. Save and close the template.
5. Upload the template to your application:

   a. Navigate to your `<app dir>\control` directory.

For the Discover Electronics reference application, this is `C:\Endeca\apps\Discover\control` on Windows, or `/usr/local/endeca/apps/discover/control` on UNIX.

b. Run the `set_templates` batch or shell script.

Note

You must configure a cartridge handler for your template in order to use it in Experience Manager.

The following shows the sample template in the Discover Electronics application for a dynamic slot cartridge. The slot is restricted to cartridges of the `SecondaryContent` group.

```
{
    "@description": "${template.description}",
    "@group": "SecondaryContent",
    "ecr:createDate": "2016-09-12T17:33:56.623+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Secondary Content Slot",
        "ruleLimit": "1",
        "templateTypes": ["SecondaryContent"]
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [{"editor": "editors/DynamicSlotEditor"}]
    },
    "typeInfo": {
        "contentPaths": {"@propertyType": "List"},
        "ruleLimit": {"@propertyType": "String"},
        "templateIds": {"@propertyType": "List"},
        "templateTypes": {"@propertyType": "List"}
    }
}
```

You must specify a cartridge handler for each cartridge template that you configure as a dynamic slot.

Specifying a cartridge handler for a dynamic slot template

All dynamic slot cartridges can share the same cartridge handler, but each unique cartridge must be explicitly configured to do so.

As soon as you have created a cartridge template that uses a dynamic slot, you must register a cartridge handler for that template. This cartridge handler should inherit the `CartridgeHandler_ContentSlot` handler.

To specify a cartridge handler for a dynamic slot template:

1. Open the configuration file for your application framework.

   In the Discover Electronics reference application, this is the Spring context configuration file located in `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring\WEB-INF\assembler-context.xml`.

2. Configure a cartridge handler for your template that inherits or extends the `ContentSlotHandler`.

   In the Spring implementation of the Assembler, this consists of adding a new `CartridgeHandler` bean for your dynamic slot cartridge:

a. Set the `id` attribute to `CartridgeHandler_<template_id>`.

b. Set the `parent` attribute to the `CartridgeHandler_ContentSlot` handler.

c. Set the `scope` attribute to `prototype` to instantiate a new handler each time one is required.

This results in configuration similar to the following:

```
<bean id="CartridgeHandler_MyPageSlot" parent="CartridgeHandler_ContentSlot"
      scope="prototype"/>
```

3. Repeate as necessary for any other dynamic slot templates in your application.

4. Save and close the file.

Adding a Link Builder

The Link Builder editor allows the content administrator to specify a link to a static page, a single selected record, or a navigation state.

The Link Builder uses the Select Records dialog to enable the content administrator to browse to a single record or a particular navigation state in the data set (which may include search terms, dimension refinements, or a combination of both). Alternately, the Link Builder also supports entering an absolute URL to a static resource.

To add a Link Builder to a template:

1. In `typeInfo`, insert an `Item` property named `link` as in the following example:

```
"typeInfo": {
        "link": {"@propertyType": "Item"}
    }
```

2. In defaultContentItem, insert a link class of `com.endeca.infront.cartridge.model.LinkBuilder`, as in the following example:

```
"defaultContentItem": {
        "@name": "Media Banner",
        "imageAlt": "",
        "link": {"@class": "com.endeca.infront.cartridge.model.LinkBuilder"},
        "media": {"@class": "com.endeca.infront.cartridge.model.MediaObject"}
    },
```

3. Insert a corresponding `editors/LinkBuilderEditor` element within `editors/DefaultEditorPanel`.

4. Specify the `propertyName` attribute and any additional label attributes for the editor:

```
{
                "editor": "editors/LinkBuilderEditor",
                "propertyName": "link",
                "label": "${property.link.label}",
```

```
                "enabled": true
            }
```

The following shows an example of a template that includes a link builder editor:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:57.427+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Media Banner",
        "imageAlt": "",
        "link": {"@class": "com.endeca.infront.cartridge.model.LinkBuilder"},
        "media": {"@class": "com.endeca.infront.cartridge.model.MediaObject"}
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
<!-- additional elements omitted from this example -->
            {
                "editor": "editors/LinkBuilderEditor",
                "propertyName": "link",
                "label": "${property.link.label}",
                "enabled": true
            }
        ]
    },
    "typeInfo": {
        "imageAlt": {"@propertyType": "String"},
        "link": {"@propertyType": "Item"},
        "media": {"@propertyType": "Item"}
    }
}
```

About configuring the Link Builder

The Link Builder must be configured with a path to a data service.

Below is the configuration for the Link Builder in the editor JSON file for the Discover Electronics reference application, located at `<app dir>\config\import\editors\LinkBuilderEditor\_.json`:

```
{
   "ecr:type":"editor",
   "config":{
      "resourcePath":"/configuration/tools/xmgr/services/endecaBrowserService.json"
   }
}
```

Related links

- Select Records data service configuration reference (page 228)
- About the Select Records dialog (page 228)

About the Media editor

The Media editor allows the content administrator to select and link to media assets stored in a content repository.

The media editor consists of an Experience Manager editor and a lightweight Web application that enables the content administrator to browse and navigate across a set of media assets in order to more easily find specific files.

The default Discover Electronics reference application stores media directly in the Endeca Configuration Repository and uses a built-in asset browser to present these assets to the content administrator. You may also initialize an MDEX Engine to index media asset metadata and URIs as records, making them available for Guided Navigation in an enhanced Media Browser.

Note

The configuration repository provides an acceptable store for media files when used for preview purposes in an authoring environment, but Oracle recommends serving media assets from a media or content delivery server for production environments.

About the Media Browser

The default asset browser for the Media editor can only be configured to browse media assets in the Endeca Configuration Repository. If you are using another system for managing media assets, you must stand up a corresponding media MDEX Engine and enable the Media Browser in the editor configuration file.

Adding a Media editor

A Media editor allows a content administrator to link media into a cartridge. It can be combined with the Link Builder in order to create images that link to destinations in your application, such as those used in site banners.

To add a Media editor to a template:

1. In `typeInfo`, insert an `Item` property named `media` as in the following example:

```
"typeInfo": {
        "media": {"@propertyType": "Item"}
    }
```

2. In `defaultContentItem`, insert a media element with a of class `com.endeca.infront.cartridge.model.MediaObject`, as in the following example:

```
"defaultContentItem": {
        "media": {"@class": "com.endeca.infront.cartridge.model.MediaObject"}
    },
```

3. Insert a corresponding `editors/MediaEditor` element within `editors/DefaultEditorPanel`.

4. Specify the `propertyName` attribute and any additional label attributes for the editor:

```
{
```

```
                "editor": "editors/MediaEditor",
                "propertyName": "media",
                "label": "${property.media.label}",
                "enabled": true
            },
```

The following shows an example of a template that includes a media editor as part of a media banner cartridge:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:57.427+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Media Banner",
        "imageAlt": "",
        "link": {"@class": "com.endeca.infront.cartridge.model.LinkBuilder"},
        "media": {"@class": "com.endeca.infront.cartridge.model.MediaObject"}
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.media.label}"
            },
            {
                "editor": "editors/MediaEditor",
                "propertyName": "media",
                "label": "${property.media.label}",
                "enabled": true
            },
           <!-- additional elements omitted from this example -->
        ]
    },
    "typeInfo": {
        "imageAlt": {"@propertyType": "String"},
        "link": {"@propertyType": "Item"},
        "media": {"@propertyType": "Item"}
    }
}
```

In order to use the Media editor, if you are using the Endeca Configuration Repository as your media store, you must upload any media files to the repository. If you are using an external digital asset management system with a corresponding MDEX Engine, the matching application must be configured and running and the Media Browser must be enabled.

Related links

- Uploading media to the Endeca Configuration Repository (page 237)

About Media editor configuration

You can specify allowable media formats in the editor JSON file. You can also enable or disable the Media Browser, and specify the MDEX Engine that it should query for media records.

The Discover Electronics reference application uses the Endeca Configuration Repository to store media and accesses these resources through a default asset browser, rather than relying on the Media Browser and an accompanying media MDEX Engine.

Below is the configuration for the Media editor in the editor JSON file, located at `<app dir>\config\import\editors\MediaEditor\_.json`:

```
{
   "ecr:type":"editor",
   "config":{
      "useMediaBrowser":"false",
      "mediaRoots":{
         "default":"http://EXAMPLE:8006/ifcr/sites/Discover/media/",
         "MediaSource":"http://EXAMPLE:8006/discover-authoring/images/"
      },
      "mdexPort":"17000",
      "mdexHost":"EXAMPLE",
      "videoFormats":"mp4|ogg|ogv|webm",
      "imageFormats":"jpg|png|gif",
      "mediaURI":"/ifcr/sites/Discover/media/"
   }
}
```

This sets the following properties across all instances of the media editor in the application:

Property	Description
`useMediaBrowser`	This property enables or disables the media browser. By default, it is set to `false`.
`mediaRoots`	This property specifies the absolute URLs to available media repositories. It includes a nested `default` property that points to the Endeca Configuration Repository, and an additional property for each repository indexed by the media MDEX Engine. For more information, see "About resolving media paths in content items."
`default`	The absolute URL to the Endeca Configuration Repository, used by the default asset browser. The specified host and port should match those used by Workbench.
content source (`mediaSource`)	An absolute URL to a media content source. In the reference data application, records are assigned a `media.repository_id` property with a value of `mediaSource`. Your own data ingest process may assign different values for media served from varying locations. In this case, each `media.repository_id` value should have a corresponding element in the editor JSON file that identifies the URL for that content source.
`mdexPort`	For applications using the Media Browser, this is the hostname or IP address of the media MDEX Engine server.
`mdexHost`	For applications using the Media Browser, this is the port on which the specified media MDEX Engine server listens.

Property	Description
`videoFormats`	A pipe-delineated list of valid video formats. Videos do not display in either the default asset browser or Media Browser. The Discover Electronics reference application is configured to use mp4, ogg, ogv, and webm, formats. Note that Internet Explorer only supports the mp4 format. If you attempt to view an ogg, ogv, or webm format video in the Media Editor using Internet Explorer, an invalid source error message appears.
`imageFormats`	A pipe-delineated list of valid image formats. Any images not matching a listed format do not display in either the default asset browser or Media Browser. The Discover Electronics reference application is configured to use jpg, png, and gif formats.
`mediaURI`	The location of the media node within the Endeca Configuration Repository. This is only used by the default asset browser.

Note the following:

- If you store media in the Endeca Configuration Repository, you can only preview media in the jpg, png, or gif formats.
- The Media MDEX browser can only display thumbnails that have the jpg, png, or gif formats.

Working with videos and images

The JSON examples of video and image formats in this guide include only those that are supported by the renderers for the Discover Electronics reference application. If you wish to extend this list for your own application, ensure that your cartridge renderers can handle additional formats, and that your application includes logic for displaying them. If you want to display a video on your site, the renderer needs to let the browser determine the size.

The ability to store media in the Endeca Configuration Repository has been deprecated in Oracle Commerce 11.3. The Endeca Configuration Repository is not intended for serving any media in a production environment. If you store media in the Endeca Configuration Repository, the Media browser might have problems displaying videos. For example the video might display without controls or as a static image. The appearance of videos is affected by many factors, such as the browser that you are using and the operating system.

The Media MDEX browser only lists media with formats listed in the `<app dir>\config\cas\media-crawl.xml` file for the Media MDEX. You can add other formats to this file.

```
<regex>^(?i:.*\.(?:gif|avi|flv|mpe?g?|fla|flr|sol|m4v|mov|wmv|ogg|ogv|webm|bmp|ico|
tiff?|png|jpg|jpe?g|bmp|mp4|mp3))$</regex>
```

Use the **cas-cmd createCrawls** command to set that crawl. For more information on creating crawls, see the *Oracle Commerce Content Acquisition System Developer's Guide*.

Uploading media to the Endeca Configuration Repository

The ability to store media in the Endeca Configuration Repository has been deprecated in Oracle Commerce 11.3. Oracle Commerce Strongly recommends against storing any media in the Endeca Configuration Repository.

If you wish to use the Endeca Configuration Repository as your media content source, you can upload assets directly to Experience Manager. It might be useful in a development environment, where a separate media server may not be worth the effort of maintaining.

All applications created using the Deployment Template include a `set_media` script in the `<app dir>\control` directory. This script uploads media content from the `<app dir>\config\media` directory to the Endeca Configuration Repository. After uploading content, it becomes available for use in Experience Manager.

In general, you can store moderate amounts of media content in the Endeca Configuration Repository. Very roughly speaking, a moderate amount of media content is approximately thousands of media files but not tens of thousands of media files. This storage mechanism is intended as a convenience when you build an application in a development environment.

If you have larger amounts of media content, Oracle recommends employing a digital asset management system rather than uploading the media content into the Endeca Configuration Repository.

Here are a few specific guidelines to keep in mind before you upload media content to the Endeca Configuration Repository:

- Do not upload more than approximately 1 GB of media content per transaction. In this context, a transaction is one run of `set_media`.
- Do not upload more than approximately 5000 files in one transaction. This guideline essentially means you should not have more than approximately 5000 files stored in `<app dir>\config\media` and its subdirectories.
- If you have more than approximately 1000 files to upload, create subdirectories under `<app dir>\config\media` and distribute the media files among the subdirectories. (One run of `set_media` uploads all content in all subdirectories.)

To upload media content for use in Experience Manager:

1. Ensure any new media content is stored locally in `<app dir>\config\media`.

   This may include image files, video files, and so on.

2. In a command prompt, navigate to the `<app dir>\control` directory of your deployed application.

   This is located under your application directory. For example: `C:\Endeca\apps\Discover\control`.

3. Run the `set_media` script.

4. To verify that your media assets are available:

   a. Log in to Workbench.

   b. Open Experience Manager.

   c. Select a cartridge that includes the Media editor.

   d. Click the Select button to launch the Media editor and confirm that your media assets display.

About resolving media paths in content items

Links to media assets are resolved in the Media editor by combining configuration in the editor configuration file with the `media.path` property on the selected record. At runtime, these links are resolved against the media sources specified in the Assembler context file.

About media root elements

You identify authoring content sources as nested elements within the <mediaRoots> element in the editor configuration file. The name of each such element corresponds to the value of the media.repository_id property assigned to each record in your media MDEX Engine. The value of each element identifies the root location of the authoring content source.

When a content administrator opens the Media Browser in Experience Manager, media assets are retrieved for preview by appending the value of the media.path property on the record to the corresponding content source element within <mediaRoots>. The media.path is then saved to the content item when the content administrator saves the cartridge configuration.

By keeping the relative location of your media assets consistent across environments, you can maintain separate content sources for authoring and live environments without requiring content administrators to reconfigure content items.

For example, assume the following element within <mediaRoots> in the editor configuration file:

```
<myMediaSource>http://myhost.mydomain.com:8006/myCMS/Discover/media/</myMediaSource>
```

A media record with a media.repository_id value of "myMediaSource" and a media.path value of "images/foo.jpg" would resolve to:

```
http://myhost.mydomain.com:8006/myCMS/Discover/media/images/foo.jpg
```

At runtime, the value of the media.path property is instead appended to the appropriate media source configured in assembler-context.xml:

```
<!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ Media Sources
-->

<bean id="authoringMediaSources" class="java.util.ArrayList" lazy-init="true">
    <constructor-arg>
        <list>
            <bean class="com.endeca.infront.cartridge.model.MediaSourceConfig">
                <property name="sourceName" value="MyMediaSource" />
                <property name="sourceValue" value="http://${workbench.host}:
${workbench.port}/ifcr/sites/
${workbench.app.name}/media/" />
            </bean>
            <bean class="com.endeca.infront.cartridge.model.MediaSourceConfig">
                <property name="sourceName" value="default" />
                <property name="sourceValue" value="http://${workbench.host}:
${workbench.port}/ifcr/sites/
${workbench.app.name}/media/" />
            </bean>
        </list>
    </constructor-arg>
</bean>
```

```
<bean id="liveMediaSources" class="java.util.ArrayList" lazy-init="true">
    <constructor-arg>
        <list>
            <bean class="com.endeca.infront.cartridge.model.MediaSourceConfig">
                <property name="sourceName" value="MyMediaSource" />
                <property name="sourceValue" value="/images/" />
            </bean>
            <bean class="com.endeca.infront.cartridge.model.MediaSourceConfig">
                <property name="sourceName" value="default" />
                <property name="sourceValue" value="/images/" />
            </bean>
        </list>
    </constructor-arg>
</bean>
```

In a live environment, the aforementioned media record would resolve to:

```
http://myhost.mydomain.com:8006/myBiggerFasterCMS/Discover/media/assets/images/foo.jpg
```

Note

While the tooling, authoring, and live content sources can all differ, Oracle recommends configuring the Media Browser to use the authoring content source.

Enabling the Media Browser

The default browser for the Media editor can only be configured to browse media assets in the Endeca Configuration Repository. If you are serving media assets from an external content source, you must enable the Media Browser and configure it to use your media MDEX Engine.

Follow these steps to configure the Media Browser in the MediaEditor JSON file for the Discover Electronics reference application, located at `<app dir>\config\import\editors\MediaEditor\_.json`:

1. Change the value of the `useMediaBrowser` to true.

```
 "useMediaBrowser": "true",
```

2. Include a content source element under `mediaRoots`that points to your media host.

   The element name is a unique key that identifies a media host. Each record has a corresponding `media.repository_id` property that identifies its content source. The relevant content source property maps that source to a URL.

   For example, in the CAS crawl configuration for the reference data application, each record is assigned a `media.repository_id` property with a value of `mediaSource`. The `mediaSource` property in the editor configuration file specifies the URL:

```
"mediaRoots": {
        "default": "http://Example-LAP:8006/ifcr/sites/Discover/media/",
        "mediaSource": "http://Example-LAP:8006/ifcr/sites/Discover/media/"
      },
```

Note

The `default` value is only used by the default asset browser. For more information, see "About Media editor configuration" and "Media MDEX Engine schema definition."

3. Modify the `mdexPort` and `mdexHost` elements to point to the host and port of the MDEX Engine backing your media host.

```
"mdexPort": "17000",
      "mdexHost": "Example-LAP",
```

4. Save and close the file.

```
{
    "ecr:type":"editor",
    "config": {
      "useMediaBrowser": "true",
      "mediaRoots": {
        "default": "http://Example-LAP:8006/ifcr/sites/Discover/media/",
        "mediaSource": "http://Example-LAP:8006/ifcr/sites/Discover/media/"
      },
      "mdexPort": "17000",
      "mdexHost": "Example-LAP",
      "videoFormats":"mp4|ogg|ogv|webm",
      "imageFormats": "jpg|png|gif",
      "mediaURI": "/ifcr/sites/Discover/media/"
     }
}
```

5. Navigate to the `<app dir>\control` directory.

6. Run the `set_editors_config` script to publish your changes to the Endeca Configuration Repository.

Related links

- Using an MDEX Engine to index media assets (page 241)

Using an MDEX Engine to index media assets

If you are storing media resources in an independent content store, you can set up an MDEX Engine where records represent media assets and include asset metadata and URIs. Storing this information as records allows content administrators to navigate assets based on image size, modification date, or other attributes when selecting media assets for a content item.

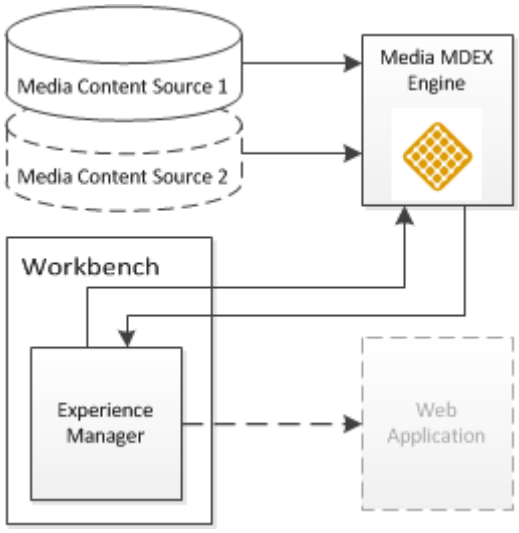
Tools and Frameworks includes a reference media MDEX application, including a CAS pipeline and Deployment Template configuration.

Interaction between Experience Manager and the media MDEX Engine

The interactions between a media MDEX Engine, Experience Manager, and an Assembler application are summarized below.

Interaction between a media MDEX Engine and Experience Manager

Experience Manager retrieves media asset information as follows:



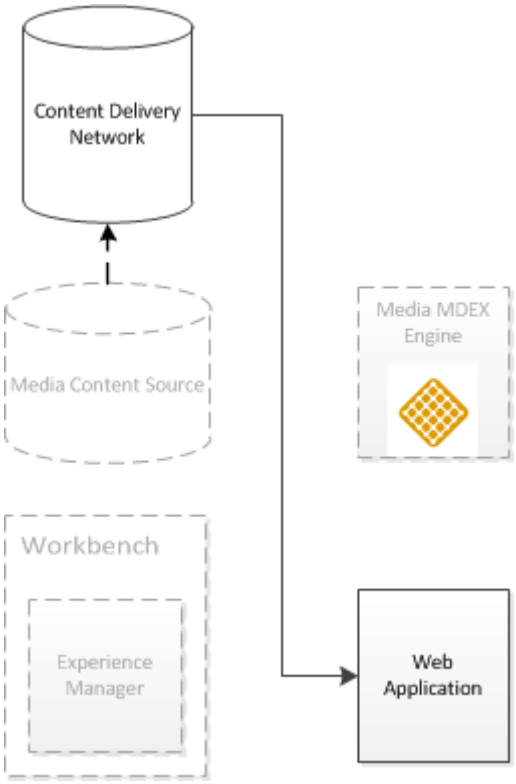
Assuming that an MDEX Engine exists with media records that adhere to the required data schema:

1. In Experience Manager, the Media Browser queries the media MDEX Engine for media records. This allows the content administrator to select media assets by navigating across them with Guided Navigation.

2. The content administrator's configuration changes are published to the application each time a content item is saved.

Interaction between the media content source and an Assembler application

In a production environment, the Assembler application can be configured to retrieve media assets from a content delivery network or another media delivery server:



1. Media assets are uploaded from the media content source to the runtime content delivery network.
2. The application retrieves media from this content delivery network.

Note

The server hosting media assets can differ between authoring and live environments, as long as the media path relative to the media root is consistent. In the case of the reference pipeline, the authoring Discover Electronics Web application serves as the content source. For more information on configuring content sources, see About Media editor configuration (page 235).

Overview of the reference data application

The Tools and Frameworks package includes a reference implementation of a media MDEX Engine that includes a CAS crawl and Forge pipeline for crawling resources on the file system and indexing the corresponding metadata and URIs. The Experience Manager can then query the MDEX Engine for record information.

The reference media application illustrates the schema requirements and configuration that you should use when building your own media pipeline.

Software requirements

In addition to the hardware and software required for Oracle Tools and Frameworks, the data ingest process for the reference data application requires the Oracle Content Acquisition System. You must have CAS installed on the machine on which you are running the ITL process for the data application.

Reference CAS crawl

The crawl uses the following manipulators:

1. **Directory Filter**: Filters out directory records, so that only media files are output to the MDEX Engine.

2. **Image Property Generator**: Analyzes image binaries to determine their width and height. It adds corresponding `image.width` and `image.height` properties to each record.

3. **Application Property Generator**: Assigns a `media.application` property based on the application specified when running the Deployment Template. This allows the Media Browser to display only those media assets that are relevant to the application that the content administrator is currently modifying in Workbench.

4. **Path Manipulator**: Creates a `media.path` property that contains the path to given asset with respect to the media root.

Media MDEX Forge pipeline

The Forge pipeline for the reference data application reads data from the Record Store populated by the CAS crawl and runs manipulators against the data to generate the required MDEX Engine schema.

Deploying the reference data application for Discover Electronics

The reference media MDEX Engine data application assumes an environment where all required Oracle components are running on the same machine.

You must have the Oracle Content Acquisition System and Oracle Tools and Frameworks installed on the machine onto which you wish to deploy the media MDEX Engine.

The reference data application runs an MDEX Engine with indexed media resources, and integrates with the Discover Electronics reference application to expose the media records to a business user in the Media editor in Experience Manager. The records include properties for metadata, such as image dimensions, making it easier to narrow down a large quantity of media assets to those that fit the requirements for a cartridge in the front-end application.

To deploy the reference data application:

1. Include the CAS manipulators for the reference data application as server plugins:

   a. Navigate to the `%CAS_ROOT%\lib\cas-server-plugins`.

   b. Create a directory named `mediaMDEX`.

   c. Navigate to the `%ENDECA_TOOLS_ROOT%\reference\media-mdex-cas\cas\media-mdex-manipulators` directory.

   d. Copy the following JAR files to the `%CAS_ROOT%\lib\cas-server-plugins\mediaMDEX` directory you created in step 1b:

      - `media-mdex-manipulators-<version>.jar`
      - `guava-14.0.jar`

   e. Navigate to the `%ENDECA_TOOLS_ROOT%\reference\media-mdex-cas\lib` directory.

   f. Copy the `commons-io-1.4.jar`file to the `%CAS_ROOT%\lib\cas-server-plugins\mediaMDEX` directory you created in step 1b.

2. Restart the CAS Service.

3. Deploy the reference data application:

   a. Open a command prompt or command shell.

Note

If you are running the Tools and Frameworks from the included batch files, you must run `ToolsAndFrameworks/<version>/server/bin/setenv.bat` to set the environment variables for the current command window.

b. Navigate to the `ToolsAndFrameworks\<version>\deployment_template\bin` directory.

c. Run `deploy.bat` or `deploy.sh` with the following options:

```
deploy --app <Endeca Directory>/ToolsAndFrameworks/<version>/reference/media-mdex-cas/deploy.xml
```

d. Confirm the Platform Services installation directory.

e. Enter **n** to skip installation of a base application.

f. Specify **`media`** as the application name.

g. Specify the application directory.

Typically, this is `C:\Endeca\apps` on Windows, or `/usr/local/endeca/apps` on UNIX.

h. Specify the EAC port previously used for the Discover Electronics reference application.

By default, this is port 8888.

i. Specify the port that Workbench runs on.

By default, this is port 8006.

j. Specify a Dgraph port.

Note

This must be a different port from any other Dgraphs used for other applications.

By default, this is port 17000. If you change this value, you must also update the configuration for the `MediaEditor` in the `\config\import\editors\MediaEditor\_.json` file after deploying the application.

k. Specify the CAS installation directory.

l. Specify the CAS version.

m. Enter the port that CAS runs on.

By default, this is port 8500.

n. Enter the name of the application in which you wish to enable media browsing.

For the Discover Electronics reference application this should be `Discover`.

o. Enter the absolute path to the location on disk where media assets are stored.

This is the file path that the Content Acquisition System crawls to index the files. In a default Discover Electronics deployment it is `C:\Endeca\apps\Discover\config\media` on Windows, or `usr/local/endeca/apps/Discover/config/media` on Unix.

4. Provision the application with the EAC and run a baseline update:

   a. Navigate to the `control` directory of the deployed media application.

   b. Run the `initialize_services` script to provision the application in the EAC.

   c. Run the `baseline_update` script to crawl the directory specified in Step 4 and index the assets in the MDEX Engine.

Pipeline configuration for a media crawl

In order for the Media Browser in Experience Manager to have sufficient information for forming content XML, any pipeline that you configure for a media MDEX Engine must define specific properties and dimensions.

Required properties

The following properties are required for the Media Browser to function correctly:

Field	Description
`record.id`	A unique identifier for each of the media items.
`media.name`	The filename of the media asset.
`media.path`	The file path, relative to the root of the content source.
`media.repository_id`	The logical host of the content source. The value of this property is mapped to configuration elements for the Media editor in the editor configuration file, which in turn contain the path to the content source. For additional information, see "About Media editor configuration."
`media.application`	The EAC application that the specified media asset is associated with. The Media editor in Experience Manager filters entries in the Media Browser based on which application the content administrator is currently editing.
`media.size`	The binary size of the media asset, in bytes.
`image.height`	The height of the media asset, if it is an image. The renderer for the Media editor uses this information to scale images appropriately.
`image.width`	The width of the media asset, if it is an image. The renderer for the Media editor uses this information to scale images appropriately.

Properties and dimensions provided in the reference data application

Optionally, additional properties and dimensions can be displayed in the Media Browser. The reference implementation of a media MDEX Engine includes the following such fields:

Field	Type	Description
`media.file_type`	Property	The MIME type of the media asset. This enables filtering by media type and file extension in the Media Browser.
`media.last_modification_date`	Property	The date and time that the file was last modified prior to being crawled by the Content Acquisition System.
`fileType`	Dimension	A searchable dimension based on `media.file_type` values.
`height`, `width`	Dimension	Searchable dimensions based on `image.height` and `image.width` values.
`application`	Dimension	A searchable dimension based on `media.application` values.

If you configure your own media MDEX Engine that includes properties or dimensions not listed above, they become available for Guided Navigation in the Media Browser. However, any such properties are not saved to content XML once a media asset has been selected.

Search interface requirements

The Media Browser requires a defined search interface named "All" that includes all searchable properties and dimensions in the data set. Additionally, the Media Browser in the reference application uses the "MatchAllPartial" search mode.

Adding a Boost-Bury Record editor

The Boost-Bury Record editor enables a content administrator to specify certain records to display either at the top or bottom of the list of results for a page.

The Boost-Bury Record editor uses the Select Records dialog to enable the content administrator to specify either an ordered list of record IDs or a set of refinements that define the set of records to be boosted or buried.

> **Note**
>
> The Boost-Bury Record editor communicates with the MDEX Engine. In order to enable the editor, ensure that you have enabled communication between Experience Manager and the MDEX Engine.

To add a Boost-Bury Record editor:

1. In `typeInfo`, insert `boostStrata` and `buryStrata` with property types of list.

```
"typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"}
```

```
    }
```

2. Insert an `editors/BoostBuryRecordEditor` element within `editors/DefaultEditorPanel`.

3. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The name of the item property that represents the records to be boosted to the top of the results. This property must be declared in the same template as the Record Stratification editor.
`buryProperty`	Required. The name of the list property that represents the records to be buried at the bottom of the results. This property must be declared in the same template as the Record Stratification editor.

```
{
              "editor": "editors/BoostBuryRecordEditor",
              "buryProperty": "buryStrata",
              "propertyName": "boostStrata",
              "label": "${property.boostBury.label}"
          },
```

The following shows an example of a template that includes a Boost-Bury Record editor:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:58.542+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Results List",
        "relRankStrategy": "",
        "recordsPerPage": "10",
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
```

```
                "editor": "editors/BoostBuryRecordEditor",
                "buryProperty": "buryStrata",
                "propertyName": "boostStrata",
                "label": "${property.boostBury.label}"
            },
<!-- additional elements omitted from this example -->
                ]
    },
    "typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"},
        "recordsPerPage": {"@propertyType": "String"},
        "relRankStrategy": {"@propertyType": "String"},
        "sortOption": {"@propertyType": "Item"}
    }
}
```

Related links

- Select Records data service configuration reference (page 228)
- About the Select Records dialog (page 228)

About configuring the boost-bury record editor

The boost-bury record editor must be configured with a path to a data service.

Below is the configuration for the boost-bury editor in the editor JSON file for the Discover Electronics reference application, located at `<app dir>\config\import\editors\BoostBuryRecordEditor\_.json`:

```
{
    "ecr:type":"editor",
    "config" : {
       "resourcePath": "/configuration/tools/xmgr/services/endecaBrowserService.json"
     }
}
```

Adding a Guided Navigation editor

The Guided Navigation editor enables a content administrator to quickly create a navigation menu through the use of the Generate Guided Navigation wizard.

Note

The Guided Navigation editor communicates with the MDEX Engine. In order to enable the editor, ensure that you have enabled communication between Experience Manager and the MDEX Engine.

A content administrator can use the Generate Guided Navigation button to trigger the Generate Guided Navigation wizard. The wizard allows them to select and order a set of dimensions to add as Refinement Menu cartridges. Alternately, they can choose to add, order, and configure the cartridges manually.

To add a Guided Navigation editor:

1. Insert an `editors/GuidedNavigationEditor` element within `editors/DefaultEditorPanel`.
2. Set a `propertyName` attribute on the `editors/GuidedNavigationEditor` element.

   This must be set to the `name` of the `ContentItemList` property that represents the list of Refinement Menu content items. The property must be declared in the same template.
3. Insert an `contentItemMapping` element within the editor.
4. Map the content item name to the dimension property that should populate it.

   This determines the name of the Refinement Menu content items created by the Generate Guided Navigation wizard.

   a. Include a `names` element within `contentItemMapping`:

```
"contentItemMapping": {
                "names": [],
            }
```

   b. Specify the dimension property to use for the content item name in a `dimensionProperty` attribute, and specify the dimension name as a fallback value.

   The Generate Guided Navigation wizard uses the first non-`null` value when naming a newly-created content item.

```
"contentItemMapping": {
                "names": [
                    {"dimensionProperty": "display_name"},
                    {"dimensionProperty": "endeca:name"}
                ],
            )
```

5. Map the `dimensionName` and `dimensionID` properties to the dimension properties that populate them:

```
"propertys": [
                    {
                       "dimensionProperty": "endeca:name",
                        "name": "dimensionName"
                    },
                    {
                       "dimensionProperty": "endeca:identifier",
                        "name": "dimensionId"
                    }
                ]
```

The following shows an example of a template that includes a guided navigation editor:

```
{
    "@description": "${template.description}",
    "@group": "SecondaryContent",
    "ecr:createDate": "2016-09-12T17:33:57.044+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
```

```
    "defaultContentItem": {"@name": "Navigation Container"},
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [{
            "editor": "editors/GuidedNavigationEditor",
            "propertyName": "navigation",
            "contentItemMapping": {
                "names": [
                    {"dimensionProperty": "display_name"},
                    {"dimensionProperty": "endeca:name"}
                ],
                "propertys": [
                    {
                       "dimensionProperty": "endeca:name",
                        "name": "dimensionName"
                    },
                    {
                       "dimensionProperty": "endeca:identifier",
                        "name": "dimensionId"
                    }
                ]
            }
        }]
    },
    "typeInfo": {"navigation": {
        "@propertyType": "ContentItemList",
        "@group": "Navigation"
    }}
}
```

Adding a Dimension Selector

A Dimension Selector enables a content administrator to specify a dimension by name.

Note

The Dimension Selector communicates with the MDEX Engine. In order to enable the Dimension Selector, ensure that you have enabled communication between Experience Manager and the MDEX Engine.

To add a Dimension Selector:

1. Insert an `editors/DimensionSelectorEditor` element within `editors/DefaultEditorPanel`.
2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The name of the string property that represents the dimension name. This property must be declared in the same template as the Dimension Selector.

Attribute	Description
`idProperty`	Required. The name of the string property that represents the dimension id. This property must be declared in the same template as the Dimension Selector.
`enabled`	If set to `false`, this attribute makes the property read-only so that the value of the property displays in the Content Details Panel in Experience Manager, but cannot be edited. Use this option only if you specify a default value in the definition of the dimension name and dimension ID properties. Editors are enabled by default.

The following shows an example of a template that includes a dimension selector:

```
{
    "@description": "${template.description}",
    "@group": "Navigation",
    "ecr:createDate": "2016-09-12T17:33:58.404+05:30",
    "@thumbnailUrl": "thumbnail.jpg",
    "ecr:type": "template",
    "defaultContentItem": {
        "lessLinkText": "Show Less Refinements...",
        "numRefinements": "10",
        "@name": "Dimension Navigation",
        "dimensionId": "",
        "moreLinkText": "Show More Refinements...",
        "maxNumRefinements": "200",
        "sort": "default",
        "showMoreLink": false,
        "dimensionName": ""    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "editors/DimensionSelectorEditor",
                "idProperty": "dimensionId",
                "propertyName": "dimensionName",
                "label": "${property.dimensionName.label}",
                "enabled": true
            },
            <!-- additional elements omitted from this example -->
        ]
    },
    "typeInfo": {
        "boostRefinements": {"@propertyType": "List"},
        "buryRefinements": {"@propertyType": "List"},
        "dimensionId": {"@propertyType": "String"},
        "dimensionName": {"@propertyType": "String"},
        "lessLinkText": {"@propertyType": "String"},
        "maxNumRefinements": {"@propertyType": "String"},
        "moreLinkText": {"@propertyType": "String"},
        "numRefinements": {"@propertyType": "String"},
        "showMoreLink": {"@propertyType": "Boolean"},
        "sort": {"@propertyType": "String"}
```

```
    }
}
```

Related links

- Select Records data service configuration reference (page 228)
- About the Select Records dialog (page 228)

Adding a Dimension List editor

The Dimension List editor enables a content administrator to select a list of dimensions from the application data set. The templates included with the reference application use this editor to specify which dimensions should be available for display in a dimension search auto-suggest panel or a dimension search results panel.

Note

The Dimension List editor communicates with the MDEX Engine. In order to enable the editor, ensure that you have enabled communication between Experience Manager and the MDEX Engine.

To add a Dimension List editor:

1. Insert an `editors/DimensionListEditor` element within `editors/DefaultEditorPanel`.
2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The name of the List property that represents the selected dimension values. The property must be declared in the same template.

The following shows an example of a template that includes a dimension list editor:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:56.937+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Dimension Search Results",
        "maxResultsPerDimension": "3",
        "showCountsEnabled": false,
        "displayImage": true,
        "maxResults": "8",
        "title": "We also found the following Categories:"
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
           <!-- additional elements omitted from this example -->
            {
```

```
                "editor": "editors/DimensionListEditor",
                "propertyName": "dimensionList",
                "label": "${property.dimensionList.label}",
                "enabled": true
            },
            {
                "editor": "editors/NumericStepperEditor",
                "minValue": 1,
                "propertyName": "maxResultsPerDimension",
                "label": "${property.maxResultsPerDimension.label}"
            }
        ]
    },
    "typeInfo": {
        "dimensionList": {"@propertyType": "List"},
        "displayImage": {"@propertyType": "Boolean"},
        "maxResults": {"@propertyType": "String"},
        "maxResultsPerDimension": {"@propertyType": "String"},
        "showCountsEnabled": {"@propertyType": "Boolean"},
        "title": {"@propertyType": "String"}
    }
}
```

Adding a Dimension Value Boost-Bury editor

The boost-bury editor enables a content administrator to specify certain dimension values to display either at the top or bottom of the list of refinements for a particular dimension.

In order to enable a Dimension Value Boost-Bury editor, the cartridge template must include a `dimensionId` property with an associated editor or a default value. This specifies the dimension to which the boost-bury editor applies.

Note

The Dimension Value Boost-Bury editor makes use of an auto-suggest dimension search component to enable the content administrator to quickly find the relevant dimension values. In order for this component to display partial matches as the user types in the search box, ensure that wildcard search is enabled for dimension searches in your MDEX Engine configuration.

To add a Dimension Value Boost-Bury editor:

1. Insert an `editors/BoostBuryEditor` element within `editors/DefaultEditorPanel`.
2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The name of the list property that represents the list of dimension values to be boosted to the top of the list of refinements. This property must be declared in the same template as the boost-bury editor.

Attribute	Description
`dimensionId`	Required. The ID of the dimension that contains the dimension refinements to boost or bury.
`boostProperty`	Required. The name of the list property that represents the list of dimension values to be boosted to the top of the refinement list. This property must be declared in the same template as the boost-bury editor.
`buryProperty`	Required. The name of the list property that represents the list of dimension values to be buried at the bottom of the list of refinements. This property must be declared in the same template as the boost-bury editor.
`enabled`	If set to false, this attribute makes the property read-only so that the value of the property displays in the Content Details Panel in Experience Manager, but cannot be edited. Use this option only if you specify a default value for the `boostList` and `buryList` properties. Editors are enabled by default.

The following shows an example of a template that includes a dimension value boost-bury editor:

```
{
    "@description": "${template.description}",
    "@group": "Navigation",
    "ecr:createDate": "2016-09-12T17:33:58.404+05:30",
    "@thumbnailUrl": "thumbnail.jpg",
    "ecr:type": "template",
    "defaultContentItem": {
        "lessLinkText": "Show Less Refinements...",
        "numRefinements": "10",
        "@name": "Dimension Navigation",
        "dimensionId": "",
        "moreLinkText": "Show More Refinements...",
        "maxNumRefinements": "200",
        "sort": "default",
        "showMoreLink": false,
        "dimensionName": ""    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
<!-- additional elements omitted from this example -->
            {
                "editor": "editors/BoostBuryEditor",
                "buryProperty": "buryRefinements",
                "propertyName": "boostRefinements",
                "label": "${property.boostBury.label}",
                "dimensionIdProperty": "dimensionId",
                "enabled": true
            },
<!-- additional elements omitted from this example -->
```

```
            ]
        },
    "typeInfo": {
        "boostRefinements": {"@propertyType": "List"},
        "buryRefinements": {"@propertyType": "List"},
        "dimensionId": {"@propertyType": "String"},
        "dimensionName": {"@propertyType": "String"},
        "lessLinkText": {"@propertyType": "String"},
        "maxNumRefinements": {"@propertyType": "String"},
        "moreLinkText": {"@propertyType": "String"},
        "numRefinements": {"@propertyType": "String"},
        "showMoreLink": {"@propertyType": "Boolean"},
        "sort": {"@propertyType": "String"}
    }
}
```

Adding a Dimension Value List editor

The Dimension Value List editor enables a content administrator to select a list of dimension values from the application data set.

Note

The Dimension Value List editor communicates with the MDEX Engine. In order to enable the editor, ensure that you have enabled communication between Experience Manager and the MDEX Engine.

To add a Dimension Value List editor:

1. Insert an `editors/DimvalListEditor` element within `editors/DefaultEditorPanel`.
2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The name of the List property that represents the selected dimension values. The property must be declared in the same template.
`dimensionId`	Required. The ID of the dimension that the editor applies to.

The following shows an example of a Refinement Menu template that uses two Dimension Value List editors to specify boosted and buried refinements, instead of a Dimension Value Boost-Bury editor:

```
{
    "@description": "${template.description}",
    "@group": "Navigation",
    "ecr:createDate": "2016-09-12T17:33:58.404+05:30",
    "@thumbnailUrl": "thumbnail.jpg",
```

```
    "ecr:type": "template",
    "defaultContentItem": {
        "lessLinkText": "Show Less Refinements...",
        "numRefinements": "10",
        "@name": "Dimension Navigation",
        "dimensionId": "",
        "moreLinkText": "Show More Refinements...",
        "maxNumRefinements": "200",
        "sort": "default",
        "showMoreLink": false,
        "dimensionName": ""    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "Boost and Bury Dimension Refinements"
            },
            {
                "editor": "editors/DimvalListEditor",
                "dimensionIdProperty": "dimensionId",
                "label": "Boost Records",
                "propertyName": "boostRefinements"
            },
            {
                "editor": "editors/DimvalListEditor",
                "dimensionIdProperty": "dimensionId",
                "label": "Bury Records",
                "propertyName": "buryRefinements"
            },
<!-- additional elements omitted from this example -->
        ]
    },
    "typeInfo": {
        "boostRefinements": {"@propertyType": "List"},
        "buryRefinements": {"@propertyType": "List"},
        "dimensionId": {"@propertyType": "String"},
        "dimensionName": {"@propertyType": "String"},
        "lessLinkText": {"@propertyType": "String"},
        "maxNumRefinements": {"@propertyType": "String"},
        "moreLinkText": {"@propertyType": "String"},
        "numRefinements": {"@propertyType": "String"},
        "showMoreLink": {"@propertyType": "Boolean"},
        "sort": {"@propertyType": "String"}
    }
}
```

Adding an Image Preview

An image preview retrieves an image from a URL and displays it in the Experience Manager interface.

You can construct an image preview URL from a hard-coded value, or from any number of String properties. Image preview supports JPEG, GIF, and PNG image formats.

To add an image preview to a template:

1. Insert an `editors/ImagePreview` element within `editors/DefaultEditorPanel`.

2. Specify attributes for the image preview:

Attribute	Description
`urlExpression`	Required. The source of the image URL. You can construct `urlExpression` from any number of string properties, or you can enter a static value.
`maxHeight`	The height in pixels of the image preview presented in the Experience Manager interface. The default value is `100`.
`maxWidth`	The width in pixels of the image preview presented in the Experience Manager interface. The default value is `300`.
`displayUrl`	A Boolean indicating whether to display the resolved URL. The default value is `true`.

If you are using more than one string property to compose the URL, you may want to use a `GroupLabel` to indicate to Experience Manager users that these properties are related.

The following examples show options for constructing an image preview.

```
    "ecr:createDate": "2016-09-12T17:33:57.256+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Image Banner",
        "url": "category_cameras_gen_1004x225.jpg",
        "serverURL":"http://TRAGHAVA-LAP:8006/ifcr/sites/Discover/media"
```

```
{
    "@description": "${template.description}",
    "@group": "ImageBanner",
    "ecr:createDate": "2016-09-12T17:33:57.256+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Image Banner",
        "url": "category_cameras_gen_1004x225.jpg",
        "serverURL":"http://TRAGHAVA-LAP:8006/ifcr/sites/Discover/media"
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
```

```
{
    "@description": "${template.description}",
    "@group": "ImageBanner",
    "ecr:createDate": "2016-09-12T17:33:57.256+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Image Banner",
        "url": "category_cameras_gen_1004x225.jpg",
        "serverURL":"http://TRAGHAVA-LAP:8006/ifcr/sites/Discover/media"
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
              "editor": "GroupLabel",
              "label": "${group.image.label}"
            },
            {
              "editor": "editors/StringEditor",
              "enabled": true,
              "label": "${property.serverURL.label}",
              "propertyName": "serverURL"
            },
            {
              "editor": "editors/StringEditor",
              "label": "${property.url.label}",
              "propertyName": "url"
            },
            {
              "editor": "editors/ImagePreview",
              "enabled": true,
              "maxWidth": 700,
              "maxHeight": 100,
              "urlExpression": "{serverURL}/{url}",
              "label": "${property.imageBanner.label}"
            }
        ]
    },
    "typeInfo": {"url": {"@propertyType": "String"},
    "serverURL": {"@propertyType": "String"}}
}
```

Adding a Rich Text editor

The Rich Text editor provides a text field and formatting toolbar that allows a content administrator to include formatted text and hyperlinks in a content item.

To add a Rich Text editor to a template:

1. In `typeInfo`, insert a `String` element inside a `@propertyType` element.
2. In `defaultContentItem`, specify the default value for the property as the content of the `String` element.
3. Insert a corresponding `editors/RichTextEditor` element within `editors/DefaultEditorPanel`.
4. Specify the `propertyName` attribute and any additional label attributes for the editor:

```
{
                "editor": "editors/RichTextEditor",
                "propertyName": "content",
                "label": "${property.content.label}",
                "enabled": true,
                "height": 200
            }
```

5. Specify the toolbar configuration for the editor:

```
"toolbar": "[
                { name: 'document',    items : [ 'Source'] },
                { name: 'clipboard',   items :
 [ 'PasteText','PasteFromWord','-','Undo','Redo' ] },
                { name: 'insert',      items :
 [ 'Table','HorizontalRule','SpecialChar'] },
                { name: 'paragraph',   items :
 [ 'NumberedList','BulletedList','-','Outdent','Indent','-','JustifyLeft','JustifyCenter','JustifyRight'
                { name: 'links',       items : [ 'Link','Unlink','Anchor' ] },
                '/',
                { name: 'basicstyles', items :
 [ 'Bold','Italic','Underline','Strike','Subscript','Superscript' ] },
                { name: 'styles',      items :
 [ 'Styles','Format','Font','FontSize' ] },
                { name: 'colors',      items : [ 'TextColor'] }         ]",
```

Note

The Rich Text editor is an implementation of the open source CKEditor WYSIWYG Rich Text editor. For more information about toolbar buttons and their functionality, see the documentation for version 4.x of the CKEditor at http://docs.ckeditor.com/#%21/guide/dev_toolbar.

The following shows an example of a template that includes a rich text editor:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
```

```
    "ecr:createDate": "2016-09-12T17:33:58.681+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Rich Text",
        "content": ""    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.contents.label}"
            },
            {
                "toolbar": "[
                 { name: 'document',    items : [ 'Source'] },
                 { name: 'clipboard',   items :
 [ 'PasteText','PasteFromWord','-','Undo','Redo' ] },
                 { name: 'insert',      items :
 [ 'Table','HorizontalRule','SpecialChar'] },
                 { name: 'paragraph',   items :
 [ 'NumberedList','BulletedList','-','Outdent','Indent','-','JustifyLeft','JustifyCenter','JustifyR
                 { name: 'links',       items : [ 'Link','Unlink','Anchor' ] },
                 '/',
                 { name: 'basicstyles', items :
 [ 'Bold','Italic','Underline','Strike','Subscript','Superscript' ] },
                 { name: 'styles',      items :
 [ 'Styles','Format','Font','FontSize' ] },
                 { name: 'colors',      items : [ 'TextColor'] }         ]",
                "editor": "editors/RichTextEditor",
                "propertyName": "content",
                "label": "${property.content.label}",
                "enabled": true,
                "height": 200
            }
        ]
    },
    "typeInfo": {"content": {"@propertyType": "String"}}
}
```

Adding a Sort editor

A Sort editor enables the content administrator to choose a sort order (sort key and direction) to apply to a list of records.

Within the results list cartridge, this sort order (along with any boost/bury that is configured for the page) is applied to the results list by default when the end user first arrives at a page. If additional sort options are specified for this cartridge, the end user can select an alternate sort order and later return to the default ord ering as specified by the content administrator.

To add a Sort editor:

1. Insert an `editors/SortEditor` element within `editors/DefaultEditorPanel`.

2. Specify additional attributes for the editor:

Attribute	Description
`propertyName`	Required. The name of the item property that represents the default sort option. This property must be declared in the same template as the Sort editor.

3. Specify one or more items of class `com.endeca.infront.navigation.model.SortOption` from which the content administrator can select.

The following shows an example of a template that includes a sort editor:

```
{
    "@description": "${template.description}",
    "@group": "MainContent",
    "ecr:createDate": "2016-09-12T17:33:58.542+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Results List",
        "relRankStrategy": "",
        "recordsPerPage": "10",
        "sortOption": {
            "@class": "com.endeca.infront.navigation.model.SortOption",
            "label": "Most Sales",
            "sorts": [{
                "@class": "com.endeca.infront.navigation.model.SortSpec",
                "key": "product.analytics.total_sales",
                "descending": false
            }]
        }
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
<!-- additional elements omitted from this example -->
            {
                "editor": "editors/SortEditor",
                "propertyName": "sortOption",
                "label": "${property.sortOption.label}",
                "sortOptions": [
                    {
                        "@class": "com.endeca.infront.navigation.model.SortOption",
                        "label": "${property.sortOption.default.label}"
                    },
                    {
                        "@class": "com.endeca.infront.navigation.model.SortOption",
                        "label": "${property.sortOption.sales.label}",
                        "sorts": [{
                            "@class": "com.endeca.infront.navigation.model.SortSpec",
                            "key": "product.analytics.total_sales",
                            "descending": true
                        }]
                    },
                    {
```

```
                        "@class": "com.endeca.infront.navigation.model.SortOption",
                        "label": "${property.sortOption.conversionRate.label}",
                        "sorts": [{
                            "@class": "com.endeca.infront.navigation.model.SortSpec",
                            "key": "product.analytics.conversion_rate",
                            "descending": true
                        }]
                    },
                    {
                        "@class": "com.endeca.infront.navigation.model.SortOption",
                        "label": "${property.sortOption.priceAscending.label}",
                        "sorts": [{
                            "@class": "com.endeca.infront.navigation.model.SortSpec",
                            "key": "product.price",
                            "descending": false
                        }]
                    },
                    {
                        "@class": "com.endeca.infront.navigation.model.SortOption",
                        "label": "${property.sortOption.priceDescending.label}",
                        "sorts": [{
                            "@class": "com.endeca.infront.navigation.model.SortSpec",
                            "key": "product.price",
                            "descending": true
                        }]
                    }
                ]
            }
        ]
    },
    "typeInfo": {
        "boostStrata": {"@propertyType": "List"},
        "buryStrata": {"@propertyType": "List"},
        "recordsPerPage": {"@propertyType": "String"},
        "relRankStrategy": {"@propertyType": "String"},
        "sortOption": {"@propertyType": "Item"}
    }
}
```

Adding a Spotlight Selection editor

The Spotlight Selection editor uses the Select Records dialog to enable a content administrator to designate specific records to spotlight in a section, or to specify a query to return a dynamic list of records.

Note

The Spotlight Selection editor communicates with the MDEX Engine. In order to enable the editor, ensure that you have enabled communication between Experience Manager and the MDEX Engine.

A Spotlight Selection editor is bound to a `recordSelection` property, which can contain either a list of record IDs (for featured records) or a set of dimension refinements (for dynamic records).

To add a Spotlight Selection editor to a template:

1. Insert an `Item` property of class `com.endeca.infront.cartridge.RecordSpotlightSelection`.

In the following `typeInfo` example, this is the `recordSelection` property:

```
"typeInfo": {
        "recordSelection": {"@propertyType": "Item"}
    }
```

In the corresponding `defaultContentItem` example, this is the class:

```
"defaultContentItem": {
        "recordSelection": {"@class":
 "com.endeca.infront.cartridge.RecordSpotlightSelection"}
    },
```

2. Insert a `String` property that stores the maximum number of records to display in the spotlight.

   In the following `typeInfo`example, this is the `maxNumRecords` property:

```
"typeInfo": {
        "maxNumRecords": {"@propertyType": "String"},
        "recordSelection": {"@propertyType": "Item"},
        }
```

   In the corresponding `defaultContentItem` example, this is the maxNumRecords:

```
"defaultContentItem": {
        "maxNumRecords": "10",
        "recordSelection": {"@class":
 "com.endeca.infront.cartridge.RecordSpotlightSelection"}
    },
```

3. Insert a `Boolean` property that controls the display of the "See All" link.

   In the following `typeInfo`example, this is the `showSeeAllLink` property:

```
"typeInfo": {
        "maxNumRecords": {"@propertyType": "String"},
        "recordSelection": {"@propertyType": "Item"},
        "showSeeAllLink": {"@propertyType": "Boolean"},
        }
```

   In the corresponding `defaultContentItem` example, this is the showSeeAllLink:

```
"defaultContentItem": {
        "maxNumRecords": "10",
        "showSeeAllLink": false,
        "recordSelection": {"@class":
 "com.endeca.infront.cartridge.RecordSpotlightSelection"}
    },
```

4. Insert a `String` property to contain the text for the "See All" link.

   In the following `typeInfo` example, this is the `seeAllLinkText` property:

```
"typeInfo": {
        "maxNumRecords": {"@propertyType": "String"},
        "recordSelection": {"@propertyType": "Item"},
        "seeAllLinkText": {"@propertyType": "String"},
        "showSeeAllLink": {"@propertyType": "Boolean"}
     }
```

   In the corresponding `defaultContentItem` example, this is the seeAllLinkText:

```
"defaultContentItem": {
        "@name": "Spotlight Records",
        "maxNumRecords": "10",
        "seeAllLinkText": "",
        "showSeeAllLink": false,
        "recordSelection": {"@class":
 "com.endeca.infront.cartridge.RecordSpotlightSelection"}
    },
```

5. Insert an `editors/SpotlightSelectionEditor` element within `editors/DefaultEditorPanel`.

6. Specify label attributes and map the editor to the associated properties:

Attribute	Description
`propertyName`	Required. The name of the record selection property that represents the selected records or navigation state. This property must be declared in the same template as the record selection editor.
`maxNumRecords`	Required. Specifies the maximum number of records to display in the spotlight.
`showSeeAllLink`	Required. Controls the display of the "See All" link.
`seeAllLinkText`	Required. Specifies the text for the "See All" link.

The following shows an example of a template that includes a spotlight selection editor:

```
{
    "@description": "${template.description}",
    "@group": "SecondaryContent",
    "ecr:createDate": "2016-09-12T17:33:58.290+05:30",
    "@thumbnailUrl": "thumbnail.png",
    "ecr:type": "template",
    "defaultContentItem": {
        "@name": "Spotlight Records",
        "maxNumRecords": "10",
        "seeAllLinkText": "",
```

```
        "showSeeAllLink": false,
        "title": "Featured Items",
        "recordSelection": {"@class":
 "com.endeca.infront.cartridge.RecordSpotlightSelection"}
    },
    "editorPanel": {
        "editor": "editors/DefaultEditorPanel",
        "children": [
            {
                "editor": "GroupLabel",
                "label": "${group.spotlight.label}"
            },
            {
                "editor": "editors/StringEditor",
                "propertyName": "title",
                "label": "${property.title.label}",
                "enabled": true            },
            {
                "editor": "editors/SpotlightSelectionEditor",
                "maxNumRecords": "maxNumRecords",
                "propertyName": "recordSelection",
                "seeAllLinkText": "seeAllLinkText",
                "showSeeAllLink": "showSeeAllLink",
                "label": "${property.recordSelection.label}"
            }
        ]
    },
    "typeInfo": {
        "maxNumRecords": {"@propertyType": "String"},
        "recordSelection": {"@propertyType": "Item"},
        "seeAllLinkText": {"@propertyType": "String"},
        "showSeeAllLink": {"@propertyType": "Boolean"},
        "title": {"@propertyType": "String"}
    }
}
```

Related links

- Select Records data service configuration reference (page 228)
- About the Select Records dialog (page 228)

About configuring the spotlight selection editor

The spotlight selection editor must be configured with a path to a data service in order to display the Select Records dialog.

Below is the configuration for the spotlight selection editor in the editor JSON file for the Discover Electronics reference application, located at `<app dir>\config\import\editors\SpotlightSelectionEditor\_.json`:

```
{
    "ecr:type":"editor",
    "config" : {
       "resourcePath": "/configuration/tools/xmgr/services/endecaBrowserService.json"
     }
}
```

Application feature property reference

This is an overview of the mappings between features in a front-end application and their associated configuration properties.

Query configuration mappings

Global configuration for the features below is typically set in the Assembler context file on the class and property specified in the table.

Feature	URL Parameter	Global Configuration `<class>.<property>`	`Cartridge Handler(s)`
Navigation query	`N`	`FilterState.navigationFilters`	`UrlNavigationStateBuilder`
Refinement display in menu	`Nrmc`	`RefinementMenuConfig.refinementsShown`	`RefinementMenu`
Enable "Show More Refinements" link		`RefinementMenuConfig.showMore`	`RefinementMenu`
"Show More" dimension IDs		`NavigationContainer.showMoreIds`	`NavigationContainer`
Record details	`R`	`DefaultResultsListConfig`	`UrlNavigationStateBuilder`
Record offset	`No`	`ResultsListConfig.offset`	`ResultsList`
Records to show per aggregate record	`- -`	`ResultsListConfig.subRecordsPerAggregateRecord`	`ResultsList`
Record filter	`Nr`	`FilterState.recordFilters`	`UrlNavigationStateBuilder`
Records per page	`Nrpp`	`ResultsListConfig.recordsPerPage`	`ResultsList`
Record search key	`Ntk`	`FilterState.SearchFilters.key`	`ResultsList`, `DimensionSearchResult`

Feature	URL Parameter	Global Configuration <class>.<property>	Cartridge Handler(s)
Aggregate record selection	A	- -	UrlNavigationStateBuilder
Aggregate record offset	Nao	ResultsListConfig.offset	ResultsList
Aggregate record rollup key	- -	FilterState.rollupKey	UrlNavigationStateBuilder
Why Rank	whyrank	ResultsListConfig.whyRankEnabled	ResultsList
Why Match	whymatch	ResultsListConfig.whyMatchEnabled	ResultsList
Why Precedence Rule Fired	whyprecedencerulefired	RefinementMenu.whyPrecedenceRuleFired, NavigationContainer.whyPrecedenceRuleFired	RefinementMenu, NavigationContainer
Range filter	Nf	FilterState.rangeFilters	UrlNavigationStateBuilder
Geocode range filter	Nfg	FilterState.rangeFilters	UrlNavigationStateBuilder
Set preview time	Endeca_Time	UserState.date	- -
Relevance ranking Match Mode	Nrm	FilterState.SearchFilters.MatchMode	ResultsList
Relevance ranking strategy	- -	ResultsListConfig.relRankStrategy, DimensionSearchResultsConfig.relRankStrategy	ResultsList
Relevance ranking search terms	Nrt	- -	- -
Relevance ranking search key	Nrk	- -	ResultsList
EQL filter	Nrs	FilterState.eqlFilter	UrlNavigationStateBuilder
Sort key	Ns	ResultsListConfig.sortOption, RefinementMenu.sort	ResultsList, RefinementMenu
Sort order			
Compute phrasings	Ntp	SearchAdjustmentsConfig.phraseSuggestionsEnabled	UrlNavigationStateBuilder

Feature	URL Parameter	Global Configuration <class>.<property>	Cartridge Handler(s)
Rewrite query with alternate phrasing			
Search terms	Ntt	FilterState.SearchFilters.terms	UrlNavigationStateBuilder
Search mode	Ntx	FilterState.SearchFilters.matchMode	UrlNavigationStateBuilder
"Did You Mean"	Nty	SearchAdjustmentsConfig.spellSuggestionEnabled	UrlNavigationStateBuilder
Signal dimension search	Dy		
Dimension search term	Ntt with Dy=1	See Ntt	See Ntt
Dimension search range filter	Nf with Dy=1	See Nf	See Nf
Enable dimension search relevance ranking	- -	DimensionSearchResultConfig.relRank	DimensionSearchResultHandler
Dimension search scope	N with Dy=1	See N	See N
Dimension search result offset	- -	- -	- -
Dimension search dimVal count	- -	DimensionSearchResultConfig.maxResultsPerDimension	DimensionSearchResultHandler
Dimension search record filter	Nr with Dy=1	See Nr	See Nr

Feature	URL Parameter	Global Configuration `<class>.<property>`	`Cartridge Handler(s)`
Dimension search refinement configuration	`- -`	`DimensionSearchResultConfig.showCountsDisabled`	`DimensionSearchResultHandler`
Dimension search EQL filter	`Nrs` with `Dy=1`	See `Nrs`	See `Nrs`
Dimension search options	`- -`	`- -`	`- -`

7 Navigation Cartridge Configuration Reference

This appendix provides an overview of the configuration models for the included navigation cartridges. You should review this information if you use these cartridges in your Assembler application to communicate with an MDEX Engine.

Related links

- Navigation cartridge URL parameter reference (page 271)
- About the navigation cartridge configuration models (page 291)
- Request Event Attributes (page 327)

Navigation cartridge URL parameter reference

This section provides a reference to URL parameters in the navigation cartridges. The documented parameter names are configured in the Assembler, and your application can include additional parameters if you choose to extend the `RequestParamMarshaller` class or its cartridge-specific subclasses.

About this section

The tables in this section describe the Endeca navigation cartridge query parameters. They include the following information:

URL parameter description format

Parameter	The query parameter, which is case-sensitive.
Name	The common name for the query parameter.
Type and format	The valid value type for the query parameter, as well as the format for listing multiple parameters, if applicable.

Object	The associated object in the Assembler API.
Description	A description of the parameter's functionality.
Dependencies	Additional query parameters that are required to give this parameter context.

Core URL query parameters

The URL query parameters that define the search and navigation objects passed into the MDEX Engine Navigation API are configured on the `UrlNavigationStateBuilder` object. By default, the Assembler is configured to use the following parameters:

URL Parameter	Feature
`N`	Navigation filter
`Ntt`	Record search terms
`Ntk`	Record search key
`Ntx`	Record search match mode
`Nf`	Range filter
`Nfg`	Geocode filter
`Nr`	Record filter
`Nrs`	EQL filter
`Rsel`	Featured Records selector
`R`	Record
`A`	Aggregate record
`Ntp`	Auto-phrasing
`Ntl`	Language ID

Note

To execute an aggregate record query using the `A` parameter, you must specify an aggregated record rollup key. Oracle recommends setting this key in your global application configuration; for example, in the Discover Electronics reference application, it is configured in the Assembler context file.

These parameters are described in detail in the following sections. The examples provided are for the Discover Electronics reference application.

N (Navigation)

The `N` parameter sets the navigation field for a query.

Parameter	`N`
Name	Navigation
Type and format	`<dimension value id>+<dimension value id>+<dimension value id>...`
Object	`FilterState`
Description	A unique combination of dimension value IDs. A value of zero indicates the root navigation object.
Dependencies	(none)

Example 7.1. Examples

The following example is for an all-inclusive search, as it does not refine the results by any dimension value:

```
N=0
```

The following example returns products with an average review rating of 5:

```
N=100021
```

Note

The Discover Electronics reference application has Search Engine Optimization enabled by default, which encodes the above URL value to `N-256d`. For more information about creating optimized URLs, see Building optimized URLs (page 116)and the *Sitemap Generator Developer's Guide*.

Ntt (Record Search Terms)

The `Ntt` parameter sets the actual terms of a record search for a navigation query.

Parameter	`Ntt`
Name	Record Search Terms
Type and format	`<string>+<string> | <string> | <string>+<string>+<string>...`
Object	`FilterState.SearchFilter`

Description	Sets the terms of the record search for a navigation query. Each term is delimited by a plus sign (+). Each set of terms is delimited by a pipe (|). **Note** There is no explicit text search descriptor API object, so your application logic must extract search terms from the query if you wish to display them in Breadcrumbs or a similar search tracker.
Dependencies	`N`,`Ntk`; `Ntt` should have the same number of terms as `Ntk` has keys.

Example 7.2. Examples

The following example returns records with a match for the term "zoom":

```
N=0&Ntt=zoom
```

The following example returns records with a match for the terms "cameras" and "silver" in the `product.description` record property. Note that the combined terms count as a single "search term" for the purposes of query syntax:

```
N=0&Ntk=product.description&Ntt=cameras+silver
```

Note

The Discover Electronics reference application is configured to use a default search key of "All" in the Spring context definition file for the Assembler, so it will accept a Record Search terms URL parameter (`Ntt`) without an accompanying Record Search key (`Ntk`) parameter.

Ntk (Record Search Key)

The `Ntk` parameter sets which dimension, property, or search interface is evaluated for a record search query.

Parameter	`Ntk`
Name	Record Search Key
Type and format	`<search key> | <search key>...`
Object	`FilterState.SearchFilter`
Description	Sets the keys of the record search for the navigation query. Multiple keys are delimited by a vertical pipe (`|`). A search key can be a search interface defined in the MDEX Engine, a valid dimension name, or the name of a property enabled for record search in the data set.

Dependencies	N, Ntt; Ntk should have the same number of keys as Ntt has terms.

Example 7.3. Examples

The following example returns records with a match for the terms "cameras" and "silver" in the product.description record property. Note that the combined terms count as a single "search term" for the purposes of query syntax:

```
N=0&Ntk=product.description&Ntt=cameras+silver
```

The following example returns records with a match for the term "cameras" in the product.description record property OR a match for the term "silver" in the camera.color record property. Note that these are evaluated as separate terms, and that each search term is associated with the search key that appears in the same location in the sequence:

```
N=0&Ntk=product.description|camera.color&Ntt=cameras|silver
```

Note

The Discover Electronics reference application is configured to use a default search key of "All" in the Spring context definition file for the Assembler, so it will accept a Record Search terms URL parameter (Ntt) without an accompanying Record Search key (Ntk) parameter.

Ntx (Record Search Match Mode)

The Ntx parameter sets the options for record search in the navigation query.

Parameter	Ntx
Name	Record Search Mode
Type and format	<string> | <string>...
Object	FilterState.SearchFilter
Description	Sets the options for record search in the navigation query. Multiple values are separated with a vertical pipe (|) character.
Dependencies	N, Ntt, Ntk

Example 7.4. Examples

The following example returns records with a match for the terms "cameras" and "silver" in the `product.description` record property. It overrides the default match mode with "MatchAllAny":

```
N=0&Ntk=product.description&Ntt=cameras+silver&Ntx=matchallany
```

Nf (Range Filter)

The `Nf` parameter sets the range filters for the navigation query.

Parameter	`Nf`
Name	Range Filter

Type and format	`<search key>|[LT|LTEQ|GT|GTEQ]+<numeric value> ||[Another range filter]…`
	
	`<search key>|BTWN+<numeric value>+<numeric value>…`

Object	`FilterState.RangeFilter`
Description	Sets the range filters for the navigation query on properties or on dimensions. Multiple range filters are separated with a double vertical pipe (||) delimiter. Accepts property and dimension values of Numeric type (Integer, Floating point, DateTime). For values of type Floating point, you can specify values using both decimal (0.00...68), and scientific notation (6.8e-10).
Dependencies	`N`

Example 7.5. Examples

The following example returns products with a price below $25:

```
N=0?Nf=product.price|LT+25
```

The following example returns products with a price between $50 and $100 (inclusive):

```
N=0?Nf=product.price|BTWN+50+100
```

It is equivalent to specifying a "greater than or equal to" filter in combination with a "less than or equal to" filter:

```
N=0?Nf=product.price|GTEQ+50||product.price|LTEQ+100
```

Nfg (Geocode Filter)

The `Nfg` parameter sets a geocode filter for the navigation query, with radius in kilometers.

Parameter	`Nfg`
Name	Geocode Filter
Type and format	`<key>|<latitude>|<longitude>|<radius>`
Object	`FilterState.GeoFilter`
Description	Filters records by evaluating the geocode location contained in the `key` property to see if it falls within the circular area defined by a central point at `latitude`, `longitude` with the specified `radius` in kilometers. Positive `latitude` values are interpreted as °N of the equator, and positive `longitude` values are interpreted as °E of the Prime Meridian.
Dependencies	`N`

Examples

The following example checks store geocodes within 10 km of the Statue of Liberty in NYC, NY:

```
N=0&Nfg=store.geocode|40.6893|-74.0446|10
```

Nr (Record Filter)

The `Nr` parameter sets a record filter on a navigation query.

Parameter	`Nr`
Name	Record Filter
Type and format	`<string>`
Object	`FilterState.RangeFilter`
Description	This parameter can be used to specify a record filter expression that restricts the results of a navigation query. Record filter syntax is described in the *MDEX Engine Development Guide*.
Dependencies	`N`

Example 7.6. Examples

A general syntax example is given below:

```
N=0&Nr=AND(132831,propertyA:valueX,OR(propertyB:valueY,propertyC:valueZ))
```

The following example only includes records that are tagged as products, and it excludes any products that are not in stock:

```
N=0&Nr=AND((common.record_type:product),NOT(product.inventory.count:0))
```

Nrs (Endeca Query Language Filter)

The `Nrs` parameter sets an EQL record filter on a navigation query. Using EQL enables you to specify multiple filters (such as a geocode range filter, a dimension value filter, and a record search filter) as part of the same query parameter.

Parameter	`Nrs`
Name	Endeca Query Language Filter
Type and format	`<string>`
Object	`FilterState.RangeFilter`

Description	Sets the Endeca Query Language expression for the navigation query. The expression acts as a filter to restrict the results of the query. Endeca Query Language syntax is documented in the *MDEX Engine Development Guide*. **Note** The `Nrs` parameter must be URL-encoded. For clarity's sake, however, the examples below are not URL-encoded.
Dependencies	`N`

Example 7.7. Examples

Consider the sample Geocode Filter discussed earlier, which matches records at stores within 10 km of the Statue of Liberty in NYC, NY:

```
N=0&Nfg=store.geocode|40.6893|-74.0446|10
```

Combining the above with a record filter that excludes out-of-stock records results in the following:

```
N=0&Nfg=store.geocode|40.6893|-74.0446|10&Nr=NOT(product.inventory.count:0))
```

The above functionality can be duplicated with a single EQL query parameter by using the following expression:

```
N=0&Nrs=collection()/record[product.inventory.count!=0 and
 endeca:distance(store.geocode,endeca:geocode(40.6893,-74.0446))<10]
```

R (Record)

The `R` parameter specifies a single Endeca record to return from the MDEX Engine.

Parameter	`R`
Name	Record
Type and format	`<record id>`
Object	`RecordState`
Description	Query to obtain a single record from the MDEX Engine.
Dependencies	(none)

Examples

The following example specifies the IXUS 85 IS camera in the Discover Electronics data set; however, because the application is configured with a global aggregate record rollup key, all records are treated as aggregated records, so the R URL query parameter has no effect:

```
R=1469273
```

Rsel (Featured Records Selector)

The Rsel parameter restricts the search results list to a set of records specified by record ID.

Parameter	Rsel
Name	Featured Records Selector
Type and format	<record ID>,<record ID>,<record ID>...
Object	FilterState
Description	A comma-delineated list of record IDs. Search results are restricted to only those records specified as values for this query parameter.
Dependencies	R

Examples

The following example restricts the results list to the Z980 and Digital IXUS 85 IS cameras:

```
R=0?Rsel=1469273,1980692
```

A (Aggregated Record)

The A parameter specifies a single aggregated record to return from the MDEX Engine.

Parameter	A
Name	Aggregated Record
Type and format	<aggregated record id>
Object	RecordState
Description	Query to obtain a single aggregated record from the MDEX Engine.

Dependencies	(none)

Example 7.8. Example

The following example specifies the IXUS 85 IS camera in the Discover Electronics data set; however, because the application serves record detail pages using the `/detail` servlet with a record-specific path, it has no effect:

```
A=1469273
```

Ntp (Auto-Phrasing)

The `Ntp` parameter sets whether the MDEX Engine applies computed alternative phrasings for the current query.

Parameter	`Ntp`
Name	Auto-Phrasing
Type and format	`[0|1]`
Object	`FilterState`
Description	Set to 1 to enable auto-phrasing, or 0 to disable it. If enabled, the MDEX Engine both computes and applies alternate query phrasings. If disabled, the MDEX Engine does not apply alternate query phrasings, but may compute them if `SearchSuggestionMdexQuery.phraseSuggestionEnabled=true`.
Dependencies	`N`,`Ntt`,`Ntk`

Examples

The following example searches the product description field for "auto focus" as a phrase, rather than searching the terms "auto" and "focus":

```
N=0?Ntk=product.description&Ntt=auto+focus&Ntp=1
```

Ntl (Language ID)

The `Ntl` parameter sets the language ID to pass in to the MDEX Engine.

Parameter	`Ntl`
Name	Language ID

Type and format	`<ISO-639 language code>`
Object	`FilterState`
Description	Specifies a language to cause the MDEX Engine to perform language-specific operations, such as invoking the correct stemming and phrasing dictionaries. For a list of supported languages, see the *MDEX Engine Development Guide*.
Dependencies	`N`

Examples

The following example specifies British English:

```
Ntl=en-GB
```

Cartridge-specific URL query parameters

For some cartridges, it is appropriate for aspects of their configuration to be overridden at query time. Typically, request-based configuration is specified as URL query parameters. This section covers the URL query parameters for the core cartridges included with Tools and Frameworks.

By default, the Assembler is configured to use the following parameters:

URL Parameter	Cartridge	Feature
`Dy`	Dimension Search Results	Enables or disables the display of returned dimension refinements.
`Ntp`	Search Adjustments	Specifies whether to display automatic phrasing; core parameter, see Ntp (Auto-Phrasing) (page 281).
`Nty`	Search Adjustments	Specifies whether to display automatic spelling correction / "Did You Mean"
`Nrmc`	Refinement Menu, Navigation Container	The `Nrmc` parameter takes multiple arguments allow you to configure dimension refinement behavior in a cartridge.
`Nrpp`	Results List	Records per page
`Ns`	Results List	Sort key and sort order
`No`	Results List	Record offset (used for paging)
`Nrt`	Results List	Relevance Ranking search terms
`Nrk`	Results List	Relevance Ranking search key

URL Parameter	Cartridge	Feature
`Nrm`	Results List	Relevance Ranking strategy
`whymatch`	Results List	Includes record matching information if query debugging is enabled
`whyrank`	Results List	Includes record ranking information if query debugging is enabled

These parameters are described in detail in the following sections. For additional information about the URL query parameters for the core cartridges, refer to the *Assembler API Reference (Javadoc)* for the relevant `RequestParamMarshaller` subclass. These classes define the URL parameters that each cartridge accepts, and their mappings to properties on the cartridge configuration model.

Dy (Dimension Search)

The `Dy` parameter controls the display of the Dimension Search Results cartridge.

Parameter	`Dy`
Name	Dimension Search
Type and format	`[0|1]`
Object	`DimensionSearchResultsConfig`
Description	Set to 1 to enable cartridge display, or 0 to disable it.
Dependencies	`N`,`Ntt`

Example 7.9. Examples

The following example returns records with a match for the term "Silver," with the Dimension Search Results cartridge enabled:

```
N=0&Ntt=Silver&Dy=1
```

Nty (Auto-Correct / DYM)

The `Nty` parameter controls the display of auto-correct and "Did You Mean" results in the Search Adjustments cartridge.

Parameter	`Nty`
Name	Auto-Correct / "Did You Mean"
Type and format	`[0|1]`
Object	`SearchAdjustmentsConfig`
Description	Set to 1 to enable display, or 0 to disable it.
Dependencies	`N`,`Ntt`

Example 7.10. Examples

The following example returns records with a match for the term "Sliver," with auto-correct enabled to correct the query to "silver":

```
N=0&Ntt=Sliver&Nty=1
```

Nrmc (Refinement Menu Config)

The `Nrmc` parameter takes multiple arguments that configure dimension refinement behavior in the Refinement Menu cartridge.

Because the Navigation Container cartridge returns a list of `RefinementMenu` objects, it takes the same `Nrmc` URL parameter as the Refinement Menu cartridge.

Parameter	`Nrmc`
Name	Refinement Menu Config
Type and format	`<Dimension ID>+show:[all|some|none] | <dimension ID>+show:[all|some|none]...`
Object	`RefinementMenuConfig`
Description	The `Nrmc` parameter takes the following values: • `<Dimension ID>` — Required. The ID of the dimension you wish to configure. • `+show:[all|some|none]` — Required; the value is passed to the `refinementsShown` property on the `RefinementMenuConfig` object, and controls how many dimension refinements to display. Configuration for multiple dimensions is separated with a vertical pipe (|) character.
Dependencies	`N`

Example 7.11. Examples

The following modifies the Refinement Menu to display all of the dimension refinements for the "Features" dimension, and hides all refinements for the "Color" dimension:

```
N=0?Nrmc=100031+show:all||101908+show:none
```

Results List cartridge URL query parameters

The following URL query parameters determine the display of search results in the Results List cartridge. They are typically set in the front-end application by the end user.

Nrpp (Records Per Page)

The `Nrpp` parameter limits the records returned from the MDEX Engine.

Parameter	`Nrpp`
Name	Records Per Page
Type and format	`<integer>`
Object	`ResultsListConfig`
Description	Sets the maximum number of records to include in the `ResultsList` object.
Dependencies	`N`

Example 7.12. Examples

The following example shows ten records per page:

```
N=0&Nrpp=10
```

Ns (Sort Key and Sort Order)

The `Ns` parameter controls sorting options for the current query. It enables the end user to override default sorting behavior on a per-query basis.

Parameter	`Ns`
Name	Sort Key and Sort Order
Type and format	`<sort key>[<geocode reference>]|[0|1] || <sort key>|[0|1]...`
Object	`ResultsListConfig`

Description	The <sort key> specifies the property or dimension to sort by. Optionally, each key can be followed by a suffix of "|1" to indicate descending sort order, or "|0" to indicate ascending order (the default). Multiple entries are separated with a double vertical pipe (||), and each entry after the first applies its sorting within the strata created by preceding entries. To sort records by a geocode property, add the optional geocode argument to the parameter (the <sort key> must be a geocode property). Records are sorted by the distance from the geocode reference point to the geocode point indicated by the <sort key> value.
Dependencies	N

Example 7.13. Examples

The following settings sort query results by product rating in descending order (higher rated products first). For each rating, it then sorts by price in ascending order (cheaper products first):

```
N=0&Ns=product.rating|0||product.price
```

The following example sorts records with a store.geocode property based on proximity to the Statue of Liberty in NYC, NY:

```
N=0&Ns=store.geocode|40.6893,-74.0446
```

No (Record Offset)

The No parameter sets the record offset in the query results list.

Parameter	No
Name	Record Offset
Type and format	<integer>
Object	ResultsListConfig
Description	Offsets the results set by the number of records specified. The offset is applied to a zero-based index; If No=20, the list of records starts at record 21. If an offset is greater than the number of items in a navigation object's record list, an empty record list is returned.
Dependencies	N,Nrpp

Example 7.14. Examples

The following example displays the second page of a results set (since the results list is configured to display 36 records per page, and is offset by that amount to start at the 37th record):

```
N=0&Nrpp=36&No=36
```

Nrt (Relevance Ranking Search Terms)

The `Nrt` parameter optionally sets search terms for a Relevance Ranking enabled record search query.

You can apply Relevance Ranking to a subset of your MDEX Engine query by specifying the desired terms in the `Nrt` parameter.

Note

If you specify a Relevance Ranking strategy on the cartridge without specifying Relevance Ranking search terms and a search key (`Nrt` and `Nrk`), the MDEX Engine evaluates the query using the Record Search Terms and Record Search Key (`Ntt` and `Ntk`) parameters. For additional information about relevance ranking strategies, see the *MDEX Engine Development Guide*.

Parameter	`Nrt`
Name	Relevance Ranking Search Terms
Type and format	`<string>+<string>+<string>...`
Object	`ResultsListConfig`
Description	Sets the terms of the record search for a navigation query with relevance ranking. Each term is delimited by a plus sign (+). **Note** Unlike the `Ntt` parameter, `Nrt` does not support using multiple sets of terms.
Dependencies	`N`, `Nrk`. Additionally, you must set the `relRankStrategy` on the cartridge.

Example 7.15. Examples

Because the Discover Electronics application uses Spring as a configuration mechanism, the cartridge-wide default values for Relevance Ranking in the Results List cartridge are specified in the `reference\discover-electronics-authoring\WEB-INF\assembler-context.xml` file:

```
<bean id="CartridgeHandler_ResultsList"
 class="com.endeca.infront.cartridge.ResultsListHandler"
          parent="NavigationCartridgeHandler" scope="prototype">
    <property name="contentItemInitializer">
        <bean class="com.endeca.infront.cartridge.ConfigInitializer" scope="request">
            <property name="defaults">
                <bean class="com.endeca.infront.cartridge.ResultsListConfig"
 scope="singleton">
                    <!-- additional configuration omitted from this example -->
                           <property name="relRankKey" value="All" />
                    <property name="relRankMatchMode" value="ALLPARTIAL" />
                    <property name="relRankStrategy"
 value="nterms,maxfield,exact,static(product.analytics.conversion_rate,descending)" />
                           <!-- additional configuration omitted from this example -->
                </bean>
            </property>
            <!-- additional configuration omitted from this example -->
        </bean>
    </property>
    <!-- additional configuration omitted from this example -->
</bean>
```

The following example returns records with a match for the terms "cameras" and "silver" in the `product.description` record property, and applies the Relevance Ranking strategy specified at the cartridge level:

```
N=0&Ntk=product.description&Ntt=cameras+silver
```

Nrk (Relevance Ranking Search Key)

The `Nrk` parameter sets which dimension, property, or search interface is evaluated for a Relevance Ranking enabled record search query.

Note

If you specify a Relevance Ranking strategy on the cartridge without specifying Relevance Ranking search terms and a search key (`Nrt` and `Nrk`), the MDEX Engine evaluates the query using the Record Search Terms and Record Search Key (`Ntt` and `Ntk`) parameters. For additional information about relevance ranking strategies, see the *MDEX Engine Development Guide*.

Parameter	`Nrk`
Name	Relevance Ranking Search Key

Type and format	`<search key>`
Object	`ResultsListConfig`
Description	Sets the search key for the record search query. This must be a navigable dimension, property name, or search interface defined in the MDEX Engine.
Dependencies	`N`, `Nrt`. Additionally, you must set the `relRankStrategy` on the cartridge.

Example 7.16. Examples

The following example returns records with a match for the terms "cameras" and "silver" in the `product.description` record property, and applies the Relevance Ranking strategy specified at the cartridge level:

```
N=0&Ntk=product.description&Ntt=cameras+silver
```

Nrm (Relevance Ranking Match Mode)

The `Nrm` parameter sets the relevance ranking strategy for ranking the results of the record search.

You can override the default Relevance Ranking strategy on a per-query basis by using the `Nrm` parameter. For additional information about match modes, see the *MDEX Engine Basic Development Guide*.

Parameter	`Nrm`
Name	Relevance Ranking Strategy
Type and format	`<string>`
Object	`ResultsListConfig`
Description	Sets the options for record search in a relevance ranking enabled query. **Note** Unlike the `Ntx` parameter, `Nrm` does not support using multiple match modes.
Dependencies	`N`, both `Nrt` and `Nrk`, OR both `Ntt` and `Ntk`. Additionally, you must set the `relRankStrategy` on the cartridge.

Example 7.17. Examples

The following example returns records with a match for the terms "cameras" and "silver" in the `product.description` record property, and applies the Relevance Ranking strategy specified at the cartridge level. It overrides the default "MatchAllPartial" match mode with "MatchAllAny":

```
N=0&Ntk=product.description&Ntt=cameras+silver&Nrm=matchallany
```

whymatch (Record Match Info)

The `whymatch` parameter controls the logging of record match information about a per-query basis.

This property enables you to include record matching information about a per-query basis, rather than at the cartridge handler level.

Parameter	`whymatch`
Name	Record match debugging information
Type and format	`[0|1]`
Object	`ResultsListConfig`
Description	Set to 1 to include record matching information, or 0 to disable it.
Dependencies	`N`, as well as either `Ntt` and `Ntk` or `Nrt` and `Nrk`. Additionally, you must have query debugging enabled in your application.

Example 7.18. Examples

The following example returns record matching information for a search against "silver cameras.":

```
N=0&Ntk=product.description&Ntt=silver+cameras&whymatch=1
```

A portion of the response (serialized to JSON) is shown below. The `DGraph.WhyDidItMatch` key contains the relevant debugging information:

```
"DGraph.WhyDidItMatch": [
    "product.long_desc: <b>The high-quality 10.0 Megapixel Digital IXUS 870 IS -
 finished in gold or silver -
     commands attention.
     ...
     Advanced compression technologies reduce file size, to free up valuable extra
 memory. (Stemming)"
],
```

whyrank (Record Rank Info)

The `whyrank` parameter controls the logging of relevance ranking information about a per-query basis.

This property enables you to include record relevance ranking information about a per-query basis, rather than at the cartridge handler level.

Parameter	`whyrank`
Name	Record ranking debugging information
Type and format	`[0|1]`
Object	`ResultsListConfig`
Description	Set to 1 to include record ranking information, or 0 to disable it.
Dependencies	`N`, as well as either `Ntt` and `Ntk` or `Nrt` and `Nrk`. Additionally, you must have query debugging enabled in your application.

Example 7.19. Examples

The following example returns record ranking information for a search against "silver cameras.":

```
N=0&Ntk=product.description&Ntt=silver+cameras&whyrank=1
```

A portion of the response (serialized to JSON) is shown below. The `DGraph.WhyRank` key contains the relevant debugging information:

```
"DGraph.WhyRank": [
    "stratify": [
      evaluationTime: "0.00048828125"
      stratumRank: "3"
      stratumDesc: "no match"
   ]
],
```

About the navigation cartridge configuration models

This section describes the configuration models for the navigation cartridges.

You can use these models as a reference when developing your own cartridges and cartridge handlers. Generally, Oracle recommends adhering to a similar approach and dividing configuration inputs to a cartridge across the following categories (ordered from lowest to highest priority):

- **Application-wide default configuration** — For the navigation cartridges, these values are configured in the Spring context file.
- **Template-specific default configuration** — These values are included in the cartridge template XML.

- **Instance configuration** — These values are configured by the business user in Experience Manager.
- **End user inputs** — For the navigation cartridges, these values are passed in as URL parameters.

Overview of the navigation cartridge configuration models

The behavior of the navigation cartridges depends on multiple sources of configuration. The data from these source is combined into a configuration model within the `initialize()` method of each associated cartridge handler in the Assembler.

Navigation cartridge configuration falls into the following categories, in ascending order of priority:

- **Default cartridge configuration**, which is specified in the Spring context file for the Assembler application
- **Cartridge instance configuration**, which is specified by the content administrator in Experience Manage
- **Request-based configuration**, which is specified by the end user in the client application

Additionally, while it is not represented in the cartridge configuration model, configuration in the MDEX Engine impacts the behavior of the navigation cartridges.

Request-based configuration overrides the cartridge instance configuration, which overrides the cartridge-level defaults, which override default feature behavior configured in the MDEX Engine.

The core cartridges typically consist of a strongly typed configuration model, a response model, and a cartridge handler that processes the configuration model into the response model. By convention, they are named as follows:

Class name	Description
`<CartridgeName>Config`	The configuration model for the cartridge. For the core cartridges, the properties of this class represent all the configuration parameters that the cartridge handler needs to do its processing. It does not include configuration that can only be specified in the MDEX Engine or pass-through properties that are used by the reference application renderers without any modification by the cartridge handler.
`<CartridgeName>Handler`	The handler that processes a cartridge. The core cartridge handlers are responsible for layering the default configuration, instance configuration, and request-based configuration during processing.
`<CartridgeName>`	The response model produced by the cartridge handler. Cartridge response models may include objects that are reused among cartridges. For example, the `ResultsList` and `RecordSpotlight` both contain `Record` objects.

For details about the implementations of these classes for specific cartridges, refer to the *Assembler API Reference (Javadoc)*.

Default cartridge configuration

You can specify default configuration settings for the navigation cartridges in the reference implementation by adding values to the cartridge handler configuration in the Spring context file.

Cartridge handler configuration (including default configuration values) is specified as part of the Spring context file for the Assembler. In the Discover Electronics application, this is defined in `WEB-INF/assembler-context.xml`.

You specify the cartridge handler for a specific cartridge by defining a bean whose ID follows the format `CartridgeHandler_<CartridgeType>`, where the `<CartridgeType>` is the `id` of the corresponding cartridge template. For example, the cartridge handler for the Breadcrumbs cartridge is defined in the `CartridgeHandler_Breadcrumbs` bean. You can map more than one cartridge to the same cartridge handler.

Typically, you specify the default configuration for a cartridge by defining a `contentItemInitializer` property within the cartridge handler. The value of this property is a bean whose class implements the `ContentItemInitializer` interface. The core cartridges use the `ConfigInitializer` class, which provides a default implementation for merging the default, instance, and request-based configuration for a cartridge. Within the `contentItemInitializer` bean, the `defaults` property (if defined) must be a bean whose class is a `ContentItem` representing the cartridge configuration model to use as a default.

For information about the properties available in the configuration model for the core cartridges, refer to the *Assembler API Reference* (Javadoc) for the relevant configuration model class.

The following shows an example of default configuration for a Record Spotlight cartridge. The `defaults` property of the `ConfigInitializer` bean is an instance of `RecordSpotlightConfig` that has been initialized with a set of default values for the `fieldNames` property.

```
<bean id="CartridgeHandler_RecordSpotlight"
    class="com.endeca.infront.cartridge.RecordSpotlightHandler"
    parent="NavigationCartridgeHandler"
    scope="prototype">
    <property name="contentItemInitializer">
        <bean class="com.endeca.infront.cartridge.ConfigInitializer" scope="request">
            <property name="defaults">
                <bean class="com.endeca.infront.cartridge.RecordSpotlightConfig"
 scope="singleton">
                    <property name="fieldNames">
                        <list>
                            <value>product.name</value>
                            <value>product.brand.name</value>
                            <value>product.price</value>
                            <value>product.min_price</value>
                            <value>product.max_price</value>
                            <value>product.img_url_thumbnail</value>
                            <value>product.review.avg_rating</value>
                        </list>
                    </property>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

Feature configuration in the MDEX Engine

There are two subcategories of MDEX Engine–level feature configuration: dynamic configuration that can be updated in a running MDEX Engine without re-indexing, and static configuration that must be specified at index time.

Dynamic configuration includes search interfaces, thesaurus, and automatic phrasing. Static configuration includes features such as such as stop words or precedence rules. Updating static configuration requires that

you re-run the data ingest process before the changes can take effect. For detailed information about feature configuration in the MDEX Engine, refer to the *MDEX Engine Basic Development Guide* and the *MDEX Engine Development Guide*.

In addition, some features depend on certain Dgraph and Dgidx flags to enable or configure their functionality. For information about Dgraph and Dgidx flags, refer to the *Oracle Commerce Administrator's Guide*.

Cartridge instance configuration

The content administrator can configure each instance of a cartridge using Experience Manager in Workbench. The cartridge instance configuration is passed in as the argument to the `initialize()` method of the cartridge handler.

You define which aspects of a cartridge are configurable in Workbench via the cartridge template. Typically this is a subset of the properties in the configuration model. The sample templates provided as part of the Discover Electronics application are intended to cover the majority of use cases.

Cartridge templates for the reference application are included in the `reference\discover-data \cartridge_templates` directory, or `<app dir>\config\cartridge_templates` directory for a deployed application.

You can customize the templates for the core cartridges by adding properties to a template in addition to those required by the configuration model. These additional properties can either be processed by a custom cartridge handler implementation or passed through directly to the response model. Some of the templates in the Discover Electronics application define pass-through properties; these are described in the sections on the specific cartridges.

For details about configuring properties and editors in a cartridge template, refer to the "Template Property and Editor Reference" appendix in this guide.

Related links

- *Template Property and Editor Reference* (page 209)

Request-based configuration

For some cartridges, it is appropriate for aspects of their configuration to be overridden at query time. Typically, request-based configuration is specified as URL query parameters.

To enable per-request configuration based on URL parameters, the `contentItemInitializer` bean of the cartridge handler can specify a `requestParamMarshaller` bean whose class is `RequestParamMarshaller` or a subclass. `RequestParamMarshaller` is a helper class that parses request parameters into properties of the cartridge configuration model.

For information about the URL query parameters that apply to the core cartridges, refer to the *Assembler API Reference (Javadoc)* for the relevant `RequestParamMarshaller` subclass. These classes define the URL parameters that the cartridge accepts and their mappings to properties on the configuration model.

Search cartridges

The Discover Electronics application includes reference implementations of several commonly-used search features. The configuration models for these features are described in the following section.

Search box

The Search Box cartridge enables the site visitor to enter search terms and view record results. If dimension search is enabled, dimension search results may also be displayed. A content administrator can configure Search Box behavior such as whether to apply search adjustments or display auto-suggest search results.

The response model for this cartridge is `SearchBox`.

The Search Box cartridge does not make use of a configuration model or a cartridge handler; properties specified in the cartridge template and in the end user's search request are passed through to the renderer.

MDEX Engine configuration for the Search Box cartridge

Because the Search Box enables keyword search for records and dimension values, most search configuration affects the behavior of this cartridge. This section focuses on record search configuration.

Dynamic configuration

The main aspects of search-related configuration that can be updated without re-indexing are the search interfaces for an application. Search interfaces specify a collection of properties and dimensions against which text searches are performed, and may also specify a default relevance ranking strategy.

The properties and dimensions within a search interface must be enabled for record search as part of the data ingest process.

Search results are also affected by thesaurus configuration that a content administrator can specify in Workbench.

For information about creating search interfaces, and enabling properties and dimensions for search, refer to the *Oracle Commerce Workbench User's Guide*.

Static configuration

Aspects of search behavior that must be specified at index time include stop words, stemming, and search characters.

- *stop words* are commonly occurring words (like "the") that are ignored for keyword search.
- *stemming* broadens search results to include root words and variants of root words.
- *search characters* configuration enables you to designate certain non-alphanumeric characters as significant for search.

For information about configuring these features, refer to the *Oracle Commerce MDEX Engine Developer's Guide*.

Template configuration for the Search Box cartridge

The Search Box cartridge does not include a configuration model or a cartridge handler; instead, template configuration is passed through to the cartridge renderer.

The Search Box cartridge template calls the typeahead service, which is configured separately.

The Search Box cartridge template includes the following configurable pass-through property:

Property name	Description
`minAutoSuggestInputLength`	This property specifies how many characters a user must type before the typeahead service is started.

Note

If you do not want to provide the option of enabling auto-suggest search results in Experience Manager, remove the properties and editors from the template, and remove the JavaScript module from the component.

Related links

- Auto-suggest search results (page 296)

Auto-suggest search results

Auto-suggest search results display as the site visitor types in the search box, rather than displaying after the visitor has completed the search. In the Discover Electronics reference application, the Search Box cartridge calls the typeahead service to display auto-suggest search results. The typeahead service must be configured by a content administrator.

A cartridge configuration class "ApplicationFilterStateConfig" represents a dynamic application filter state. This dynamic application filter state should be specified in the service or page definition as a property named "@appFilterState".

Table 7.1. Configuration Options for the @appFilterState Property

Name	Description
`languageId`	The language id to be used by the request while querying MDEX
`autoPhraseEnabled`	Auto phrase flag used to tell MDEX whether alternate phrasing should be applied.
`typeAhead`	Specifies whether typeahead should be turned on in the engine for record and dimension searches. If true, dgraph will treat this query as a typeahead use case, and rewrites the query based on the language id of the query.
`rollupKey`	The key to be used by MDEX for aggregate records
`securityFilter`	Sets a security filter, which is specified using an MDEX record filter string.
`recordFilters`	The List of record filters to be applied by MDEX
`geoFilter`	The geo filter to be applied by MDEX
`eqlFilter`	The EQL expression to be passed to MDEX.

Sample content.xml

```
<ContentItem type="Page" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns="http://endeca.com/schema/content/2008"
             xmlns:xavia="http://endeca.com/schema/xavia/2010">
    <TemplateId>TypeaheadService</TemplateId>
    <Name>Typeahead Service</Name>
    <Property name="@appFilterState">
        <ContentItem type="ApplicationFilterStateConfig">
            <Property name="typeAhead">
                <Boolean>true</Boolean>
```

```
            </Property>
            <Property name="recordFilters">
                <xavia:List>
                    <String>product.active:1</String>
                </xavia:List>
            </Property>
        </ContentItem>
    </Property>
    <Property name="resultsList">
        <ContentItem type="ResultsList">
            <TemplateId>ResultsList</TemplateId>
            <Property name="recordsPerPage">
                <String>3</String>
            </Property>
        </ContentItem>
    </Property>
</ContentItem>
```

Dimension search results

The Dimension Search Results cartridge displays refinement links based on the names of dimension values that match the search keywords entered by the site visitor.

The dimension search results display in a panel after the site visitor performs the search. These results provide suggestions for additional navigation refinements based on the search terms.

The response model for this cartridge is `DimensionSearchResults`. It contains a list of `DimensionSearchGroup` objects that in turn contain `dimensionSearchValues` that provide refinement links.

Configuration model for the Dimension Search Results cartridge

The Dimension Search Results cartridge configuration model controls the number, ranking, and display of returned results.

The configuration model for this cartridge is `DimensionSearchResultsConfig`. It includes the following properties:

Property name	Description
`enabled`	Enables or disables the display of returned dimension refinements. By default, this property is `false`. It is enabled via URL request by setting the `Dy` URL parameter to `1`.
`maxResults`	Specifies the maximum number of dimension value results across all dimensions to display.
`maxResultsPerDimension`	Specifies the maximum number of dimension values to display per dimension.
`dimensionList`	Specifies the dimensions on which to perform dimension search. The results display based on the order in which the dimensions are specified, up to the maximum number of suggestions.
`showCountsEnabled`	Specifies whether to display refinement counts in dimension search results.

Property name	Description
relRank	Optional. Specifies a relevance ranking string to use for dimension search, such as "first,static(nbins,desc)". If you do not set this property, dimension value relevance ranking is set to the default (alpha, numeric, or manual) defined in the application's evaluator.properties file.

MDEX Engine configuration for dimension search results

Different aspects of dimension search can be configured on a global or per-dimension basis.

Dynamic configuration

You can specify global dimension search behavior in the Dimension Search Configuration editor in the Oracle Commerce Workbench. Oracle recommends enabling wildcard search for dimensions, especially if you are using the Auto-Suggest Dimension Search cartridge or the Dimension Value Boost-Bury editor. Wildcard search enables partial matches to be returned for searches in addition to full word matches (for example, a search for "pink" would also return "gray/pink") which is useful for displaying suggestions while the user is typing search terms.

Additional options include whether to return only the highest ancestor dimension value, and whether to return inert dimension values in dimension search results. For more information about global dimension configuration, refer to the *Oracle Commerce Workbench User's Guide*.

Static configuration

You can configure dimension-specific search behavior in the Dimension editor in Oracle Commerce Workbench. This includes whether to search across the entire dimension hierarchy rather than only individual dimension values and also enables you to specify dimension value synonyms to be used for search. For more information about per-dimension configuration, refer to the *Oracle Commerce Workbench User's Guide*.

Cartridge handler configuration for Dimension Search Results

The Dimension Search Results cartridge handler extends the NavigationCartridgeHandler.

The cartridge handler uses the DimensionSearchResultsConfigInitializer to merge the layered configuration. The included requestParamMarshaller bean enables URL request-based configuration for the cartridge, which is required for dynamically enabling the feature.

Template configuration for the Dimension Search Results cartridge

The Dimension Search Results cartridge template allows a content administrator to configure how many results should be displayed to the end user, and how they should display. The cartridge template also includes two pass-through properties that are passed directly to the cartridge renderer.

The Dimension Search Results cartridge template allows a content administrator to configure the following properties on the configuration model:

- maxResults
- dimensionList
- maxResultsPerDimension

- `showCountsEnabled`

In addition, the cartridge template includes the following pass-through properties:

Property name	Description
`title`	Optional. A header that displays above the dimension search results.
`displayImage`	If set to true, a thumbnail image displays next to each dimension value. The URL of the image must be the value of a dimension value property named `img_thumbnail_url`. **Note** If there is no such property on dimension values in the data set, remove this option and its associated editor from the template to disable this feature.

URL request parameters for the Dimension Search Results cartridge

The display of the Dimension Search Results cartridge on a page is controlled by setting the value of the `enabled` property on the cartridge configuration model at runtime via the `Dy` URL parameter.

The cartridge renderer in the reference implementation sets the `Dy` parameter to `1` in all cases. While this is equivalent to setting the property to `true` in the cartridge handler configuration, or as a non-editable property in the cartridge template, the intent is to demonstrate where the logic belongs in the application.

Property name	URL Parameter	Description
`enabled`	`Dy`	Enables or disables the display of returned dimension refinements. Setting `Dy=1` sets the property to `true`.

Search adjustments

Search adjustments include automatic spelling correction, automatic phrasing, and Did You Mean functionality.

The response model for this cartridge is `SearchAdjustments`.

The behavior of the spelling correction and Did You Mean features are configured at the MDEX Engine level. The Search Adjustments cartridge enables content administrators to specify whether or not search adjustments messaging displays on a page; it does not have any configuration options in Experience Manager.

Configuration model for the Search Adjustments cartridge

The Search Adjustments cartridge configuration model enables you to enable or disable automatic phrasing and automatic spelling correction. If query debugging features are enabled in your application, you can also enable or disable debugging information about Word Interpretation.

The configuration model for this cartridge is `SearchAdjustemntsConfig`. It includes the following properties:

Property name	Description
`phraseSuggestionEnabled`	Specifies whether to enable automatic phrasing. Defaults to `true`. Set via URL request by setting the `Ntp` URL parameter to `1`.
`spellSuggestionEnabled`	Specifies whether to enable automatic spelling correction. Defaults to `false`. Set via URL request by setting the `Nty` URL parameter to `1`.
`showWordInterp`	If query debugging features are enabled, this property enables debugging information about word or phrase subsitutions as a map that can be accessed via `SearchAdjustments.getInterpretedTerms()`. For additional information, see "About query debugging results in the reference application."

MDEX Engine configuration for the Search Adjustments cartridge

Search adjustments features are configured at indexing and at Dgraph startup.

Dynamic configuration

You can specify a list of phrases to be automatically applied to text search queries in the Oracle Commerce Workbench. For more information about configuring automatic phrasing, refer to the *MDEX Engine Development Guide*.

Static configuration

You can configure the constraints on the spelling dictionaries for record search and dimension search in the Spelling editor in Oracle Commerce Workbench. These settings determine the size of the spelling dictionary that is generated at indexing time. Larger spelling dictionaries lead to slower performance of spelling correction at query time; setting more restrictive constraints on the contents of the spelling dictionary can lead to improved query performance. For more information about tuning the size of the spelling dictionary, refer to the *Performance Tuning Guide*.

Dgidx flags

You specify the spelling mode as a flag to Dgidx. Generally, applications that only need to correct normal English words can enable just the default Aspell module. Applications that need to correct international words, or other non-English/non-word terms (such as part numbers) should enable the Espell module. For more information about spelling modes and the associated Dgidx flags, refer to the *MDEX Engine Development Guide*.

The Deployment Template application configuration for the Discover Electronics reference application has spelling correction and Did You Mean enabled as in the following example:

```
<!--
  ########################################################################
  # Dgidx
  #
-->
<dgidx id="Dgidx" host-id="ITLHost">
  <properties>
```

```
    <property name="numLogBackups" value="10" />
    <property name="numIndexBackups" value="3" />
  </properties>
  <args>
    <arg>-v</arg>
    <arg>--compoundDimSearch</arg>
  </args>
  <log-dir>./logs/dgidxs/Dgidx</log-dir>
  <input-dir>./data/forge_output</input-dir>
  <output-dir>./data/dgidx_output</output-dir>
  <temp-dir>./data/temp</temp-dir>
  <run-aspell>true</run-aspell>
</dgidx>
```

Dgraph flags

You enable spelling correction and Did You Mean through Dgraph flags. Additional Dgraph flags provide advanced tuning options for the spelling adjustment features that affect performance and behavioral characteristics, such as the threshold for the number of hits at or above which spelling corrections or Did You Mean suggestions are not generated. For more information on Dgraph flags for search adjustment tuning, refer to the *MDEX Engine Development Guide*.

Note

Auto-correct should be relatively conservative. You only want the engine to complete the correction when there is a high degree of confidence. For more aggressive suggestions, it is best to use Did You Mean.

The Deployment Template application configuration for the Discover Electronics reference application has spelling correction and Did You Mean enabled as in the following example:

```
<!--
  ###############################################################################
  # Global Dgraph Settings - inherited by all dgraph components.
  #
-->
<dgraph-defaults>
  <properties>
    <!-- additional elements removed from this example -->
  </properties>
  <directories>
    <!-- additional elements removed from this example -->
  </directories>
  <args>
    <arg>--threads</arg>
    <arg>2</arg>
    <arg>--whymatch</arg>
    <arg>--spl</arg>
    <arg>--dym</arg>
    <arg>--dym_hthresh</arg>
    <arg>5</arg>
    <arg>--dym_nsug</arg>
    <arg>3</arg>
    <arg>--stat-abins</arg>
  </args>
  <startup-timeout>120</startup-timeout>
</dgraph-defaults>
```

Cartridge handler configuration for Search Adjustments

The Search Adjustments cartridge handler extends the `NavigationCartridgeHandler`. The application-wide default configuration in the Assembler context file allows you to enable or disable the word interpretation debugging feature.

The cartridge handler uses a `contentItemInitializer` to merge the layered configuration. The included `requestParamMarshaller` bean enables URL request-based configuration for the cartridge, which is required for dynamically disabling or enabling automatic phrase suggestions and spelling correction.

Related links

- About implementing automatic phrasing (page 302)

Template configuration for the Search Adjustments cartridge

The cartridge template for the Search Adjustments cartridge does not include any configurable properties. A content administrator can add the cartridge to a page in order to enable the display of Search Adjustments, but cannot otherwise configure cartridge behavior.

URL request parameters for the Search Adjustments cartridge

Automatic phrasing and spelling correction are controlled by setting the value of their respective properties on the cartridge configuration model at runtime via the `Ntp` and `Nty` URL parameters.

The cartridge renderer in the reference implementation sets both parameters to `1` in all cases. While this is equivalent to setting the properties in the cartridge handler configuration, or in the cartridge template, the intent is to demonstrate where the logic belongs in the application.

Property name	URL Parameter	Description
`phraseSuggestionEnabled`	`Ntp`	Specifies whether to enable automatic phrasing.
`spellSuggestionEnabled`	`Nty`	Specifies whether to enable automatic spelling correction.

About implementing automatic phrasing

You can configure the MDEX Engine to consider certain combinations of words in a text search as a phrase search and specify whether to apply phrasing automatically to a site visitor's text search queries.

The high level steps for enabling automatic phrasing are:

- Enabling the MDEX Engine to compute phrases
- Configuring the default behavior of the Assembler application as to whether or not to automatically apply computed phrases
- Adding application logic to enable Did You Mean suggestions or override the default automatic phrasing behavior in certain situations

You enable the MDEX Engine to compute phrases that can be applied to a site visitor's text search by creating a phrase dictionary. For information about creating a phrase dictionary, refer to the section on Automatic Phrasing in the *MDEX Engine Developer's Guide*.

You can configure the default behavior of the Assembler application as to whether to automatically rewrite a text search as a phrase search or keep it as a search for individual keywords using the following property on the Filter State object:

Property	Description
`autoPhraseEnabled`	If set to `true`, instructs the MDEX Engine to compute phrases that can be applied to a text search and automatically rewrite the query using the phrased version. Automatic phrasing is enabled by default.

The `autoPhraseEnabled` setting on the default Filter State can be overridden at query time using the URL parameter `autophrase`. If the value of `autophrase` is `1`, then computed phrases are automatically applied to the query. If the value is `0` then phrases may still be computed, but are not automatically applied to the query.

The Filter State configuration in the Assembler context file for the Discover Electronics reference application is shown below:

```
<bean id="navigationStateBuilder" scope="request"
    class="com.endeca.infront.navigation.url.UrlNavigationStateBuilder">
    <!-- additional elements removed from this example -->
                        <property name="defaultFilterState">
        <bean scope="singleton"
 class="com.endeca.infront.navigation.model.FilterState">
            <property name="rollupKey" value="product.code" />
            <property name="autoPhraseEnabled" value="true" />
            <!-- <property name="securityFilter" value="" /> -->
            <!-- <property name="languageId" value="en" /> -->
        </bean>
    </property>
    <!-- additional elements removed from this example -->
</bean>
```

For Oracle Commerce Experience Manager, if your application contains multiple sites, Oracle recommends using a `filterState.xml` file instead of the Filter State configuration in the Assembler context file. For example, a `filterState.xml` file in /pages/DiscoverElectronics/ might contain the following `autophrase` property:

```
<Item class="com.endeca.infront.navigation.model.FilterState" xmlns="http://endeca.com/
schema/xavia/2010">
 <Property name="autoPhraseEnabled">
     <Boolean>true</Boolean >
 </Property>
</Item>
```

Interaction with the Did You Mean feature

Whether automatic phrasing is applied or not, you can specify whether to return a "Did You Mean" link for the alternate version using the `Nty` URL parameter. For example, if phrasing was automatically applied, the Did You Mean suggestion would provide a link to the unphrased version of the query, and vice versa. If the value of `Nty`

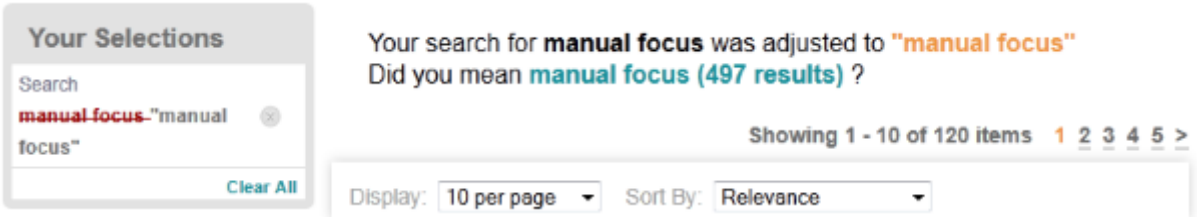
is 1, then the Assembler returns suggestions for the alternate form of the query. If the value is 0, no suggestions are returned.

Note

The Nty parameter controls Did You Mean suggestions for regular text search as well as for automatic phrasing.

Phrase search scenario: Automatically applying phrases

In the Discover Electronics application, the default behavior is to automatically apply phrases to text search queries and to return the unphrased version as a search suggestion.



In this scenario, autoPhraseEnabled is set to true on the default Filter State object, and the Search Box cartridge sets Nty=1 on the text search query. The user has two choices:

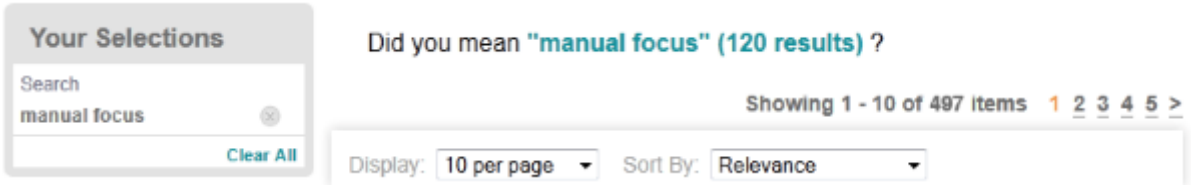
- Select the Did You Mean suggestion to search for the keywords separately, rather than as a phrase. This link sends the same query with the URL parameter Ntp=0 to override the Filter State configuration, and also sets Nty=0 since we do not need to suggest the phrased version of the query after the user has decided to use the unphrased version.
- Make another selection on the page, such as clicking on a refinement or advancing to the next page of results. This signifies acceptance of the automatically applied phrase, so we keep autoPhraseEnabled=true from the Default Filter State and suppress further suggestions by setting Nty=0.

These outcomes are summarized in the following table:

User action	Autophrase setting (Ntp)	Did You Mean setting (Nty)	Result
Initial search	true	Nty=1	Phrase is automatically applied to the text search. A Did You Mean suggestion is offered for the unphrased version.
Select Did You Mean suggestion	Ntp=0	Nty=0	Phrase is not applied to the search. No suggestion is offered.
Make another follow-on selection	true	Nty=0	Phrase continues to be automatically applied. Suggestions are no longer offered.

Phrase search scenario: Phrases as a search suggestion

You can configure the application not to apply phrases by default, but to return phrases as a search suggestion.



In this scenario, `autoPhraseEnabled` is set to `false` on the default Filter State object, and the Search Box cartridge sets `Nty=1` on the text search query. The user has two choices:

- Select the Did You Mean suggestion to consider the text search as a phrase. This link sends the same query with the URL parameter `Ntp=1` to override the default Filter State configuration, and also sets `Nty=0` since we do not need to suggest the unphrased version of the query after the user has decided to use the phrased version.
- Make another selection on the page, such as clicking on a refinement or advancing to the next page of results. This signifies acceptance of the unphrased query, so we keep `autoPhraseEnabled` set to `false` and suppress further suggestions by setting `Nty=0`.

These outcomes are summarized in the following table:

User action	Autophrase setting (`Ntp`)	Did You Mean setting (`Nty`)	Result
Initial search	`false`	`Nty=1`	Phrase is not applied to the text search. A Did You Mean suggestion is offered for the phrased version.
Select Did You Mean suggestion	`Ntp=1`	`Nty=0`	Phrase is automatically applied to the search. No suggestion is offered.
Make another follow-on selection	`false`	`Nty=0`	Text search continues to be treated as individual keywords instead of as a phrase. Suggestions are no longer offered.

Keyword redirects

Content administrators can configure keyword redirects that redirect a front-end user to a new page if the user's search terms match the set keyword.

When an end user enters a search term that matches a keyword redirect, the Assembler returns the redirect URI with the response model. The Assembler response can be limited to the redirect URI, or it can also return the results for the user's search term.

The content administrator specifies a search term, match mode, and redirect URI on the Keyword Redirects page in Workbench.

Cartridge handler configuration for keyword redirects

The Assembler API includes a `RedirectAwareContentIncludeHandler` that implements keyword redirect functionality.

The cartridge handler takes the following two properties:

- `defaultFullAssembleOnRedirect` — A Boolean that specifies whether to return search results in addition to the redirect URI when making an `assemble()` call. Defaults to `false`. If you do not necessarily wish to execute a redirect (for cases where the redirect URI is displayed as a link, or may be skipped entirely if the user is not on a specific device), you must set this property to `true`.
- `defaultRedirectCollection` — A string that contains the name of the keyword redirect collection in the Endeca Configuration Repository. Setting a null or empty value for this property disables keyword redirect functionality.

The cartridge handler configuration in the Assembler context file for Discover Electronics is shown below:

```
<!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ BEAN: CartridgeHandler_ContentInclude
    ~ Used by the assembler service when keyword redirects are not enabled
-->
<bean id="CartridgeHandler_ContentInclude"
  class="com.endeca.infront.content.ContentIncludeHandler"
  scope="prototype">
    <property name="contentSource" ref="contentSource" />
    <property name="siteState" ref="siteState"/>
    <property name="userState" ref="${user.state.ref}"/>
</bean>

<!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ BEAN: CartridgeHandler_RedirectAwareContentInclude
    ~ For root calls to the assembler when keyword redirects are desired
-->
<bean id="CartridgeHandler_RedirectAwareContentInclude"
  class="com.endeca.infront.cartridge.RedirectAwareContentIncludeHandler"
  scope="prototype">
    <property name="contentSource" ref="contentSource" />
    <property name="contentBroker" ref="contentRequestBroker" />
    <property name="navigationState" ref="navigationState" />
    <property name="defaultFullAssembleOnRedirect" value="false"/>
    <property name="siteState" ref="siteState"/>
    <property name="userState" ref="${user.state.ref}"/>
</bean>
```

Note

The redirect-aware version of the cartridge is included in the Navigation JAR rather than the core Assembler JAR because it relies on keyword redirects, which are interpreted by the MDEX Engine. The standard Content Include cartridge and classes do not have this dependency, and are packaged with the core JAR file.

Content XML for keyword redirects

You can override the default settings for the `fullAssembleOnRedirect` or `redirectCollection` properties by setting new values in the content XML that is retrieved by the `RedirectAwareContentIncludeHandler`.

The primary use case for setting these properties on content XML is for deployments running the Assembler service. Keyword redirects are programatically enabled in the service, so by default the feature is explicitly disabled for services where it does not apply (Dimension Search and Record Details) by including an element in the content XML that sets `redirectCollection` to a null value.

Note

If you are creating your Assembler application in Java, you can disable keyword redirects by using the `ContentInclude` class instead of `RedirectAwareContentInclude` for those services where you wish to disable the feature.

About using keyword redirects with the Assembler service

The Assembler service in the Discover Electronics application implements the `com.endeca.infront.assembler.servlet.AbstractAssemblerServlet` abstract class. Keyword redirect configuration is configured in the application's `web.xml` file.

The JSON and XML servlets in the Discover Electronics reference application are configured in `reference\discover-service\WEB-INF\web.xml`:

```
<servlet>
    <servlet-name>JsonAssemblerServiceServlet</servlet-name>
    <servlet-class>com.endeca.infront.assembler.servlet.spring.SpringAssemblerServlet</
servlet-class>
    <init-param>
         <param-name>assemblerFactoryID</param-name>
         <param-value>assemblerFactory</param-value>
    </init-param>
    <init-param>
         <param-name>responseWriterID</param-name>
         <param-value>jsonResponseWriter</param-value>
    </init-param>
    <init-param>
        <param-name>enableKeywordRedirects</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>
```

When the application queries the Assembler service, the redirect URI is returned as part of the response.

About handling keyword redirects in an application

In order to execute a redirect, an application must include logic for handling the URI components returned from the Assembler. You must use the `RedirectAwareContentInclude` class for any content items that require keyword redirect functionality.

The `assemble.jsp` service uses the `RedirectAwareContentInclude` class to enable keyword redirects, as shown below:

```
                        <%@page
 import="com.endeca.infront.cartridge.RedirectAwareContentInclude"%>

...
```

```
AssemblerFactory assemblerFactory =
 (AssemblerFactory)webappCtx.getBean("assemblerFactory");
Assembler assembler = assemblerFactory.createAssembler();

//Retrieve the content for the given content uri
ContentItem contentItem = new RedirectAwareContentInclude("/browse" + contentUri);

// Assemble the content
ContentItem responseContentItem = assembler.assemble(contentItem);
```

The Assembler response

When an end user enters a search term that matches a keyword redirect configured in Workbench, the Assembler response includes a `ContentItem` with the necessary information for creating a destination URI.

The following example shows a JSON response in an Experience Manager implementation from the Guided Search service when `fullAssembleOnRedirect` is `false`:

```
{
    endeca:siteRootPath: "/pages",
    endeca:contentPath: "/services/guidedsearch",
    endeca:assemblerRequestInformation:
    {
        @type: "AssemblerRequestEvent",
        endeca:assemblyStartTimestamp: 1341943119538,
        endeca:assemblyFinishTimestamp: 1341943119546,
        endeca:contentPath: "/guidedsearch",
        endeca:requestId":"140252272098164091",
        endeca:sessionId: "FF9D21355A3CBB9DFF75614DD7D2948D",
        endeca:siteRootPath: "/services"
    },
    endeca:redirect:
    {
        @type: "Redirect",
        link: {
            @class: "com.endeca.infront.cartridge.model.UrlAction",
            url: "/browse/cameras/_/N-25y6"
        }
    }
}
```

The keyword redirect information is included in the `ContentItem` with the key `endeca:redirect`. The value specifies an `Action` object with the destination URI, which may be either relative or absolute.

In an Oracle Commerce Guided Search implementation (without Experience Manager), the site root path and content path in the JSON response would be the following:

```
endeca:siteRootPath: "/services",
    endeca:contentPath: "/guidedsearch",
```

Using the Assembler response

You must retrieve and use the information from the Assembler response in your application to execute a keyword redirect. In the Discover Electronics reference application, this is accomplished in the `assemble.jsp` service:

```
                        <%@ taglib prefix="util" uri="/WEB-INF/tlds/functions.tld"
 %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

...

// Assemble the content
ContentItem responseContentItem = assembler.assemble(contentItem);

request.setAttribute("component", responseContentItem);
request.setAttribute("rootComponent", responseContentItem);

Map map = (Map) request.getAttribute("component");
if (map.containsKey("endeca:redirect")) {
    request.setAttribute("action", ((ContentItem)
 map.get("endeca:redirect")).get("link"));
    %>
    <c:redirect url="${util:getUrlForAction(action)}"/>
    <%
}
...
```

For more information about `Action` objects in an Assembler application, see "Working with Application URLs," or consult the *Assembler API Reference (Javadoc)*.

Guided Navigation cartridges

The following sections provide an overview of the configuration models for Guided Navigation features included with Tools and Frameworks and implemented in Discover Electronics.

Refinement menu

The Refinement Menu cartridge displays dimension values within a single dimension for Guided Navigation. It supports dimension value boost and bury.

The response model for this cartridge is `RefinementMenu`, which contains a list of `Refinement` objects.

Dimension value boost and bury

Dimension value boost and bury is a feature that enables re-ordering of dimension values within a particular dimension for Guided Navigation. With dimension value boost, you can assign specific dimension values to ranked strata, with those in the highest stratum being shown first, those in the second-ranked stratum shown next, and so on. With dimension value bury, you can assign specific dimension values to strata that are ranked much lower relative to others. This boost/bury mechanism therefore lets you manipulate ranking of returned dimension values in order to promote or push certain refinements to the top or bottom of the navigation menu.

The Refinement Menu cartridge enables the content administrator to specify an ordered list of dimension values to boost and an ordered list of dimension values to bury. Each dimension value is translated into its own stratum in the query that returns refinements so as to preserve the exact order of refinements specified by the content administrator.

For more information about dimension value boost and bury, refer to the *MDEX Engine Basic Development Guide*.

Configuration model for the Refinement Menu cartridge

The Refinement Menu cartridge configuration model allows you to configure sorting, "Show More..." link behavior, and boosted and buried refinements. Additionally, it includes a `whyPrecedenceRuleFired` property that can be used for debugging precedence rule behavior in your application.

The configuration model for this cartridge is `RefinementMenuConfig`. It includes the following properties:

Property name	Description
`dimensionId`	A string representing the id of the dimension being configured.
`boostRefinements`	An ordered list of dimension value refinements to display at the top of the list.
`buryRefinements`	An ordered list of dimension value refinements to display at the bottom of the list.
`sort`	The base sort order of dimension values within this dimension. This property should have one of the following values: • `default` — Sort dimension values according to the application configuration for this dimension. • `static` — Sort dimension values in alphabetic or numeric order, depending on the dimension configuration. • `dynRank` — Sort dimension values so that the refinements with the highest number of records display first.
`showMoreLink`	A Boolean indicating whether to enable a link to show more refinements than are displayed by default.
`moreLinkText`	A string representing the text to use for the "show more refinements" link.
`lessLinkText`	A string representing the text to use for the "show fewer refinements" link.
`numRefinements`	A string representing the number of refinements to display by default, or when a user clicks the "show fewer refinements" link.
`maxNumRefinements`	A string representing the maximum number of refinements to display when a user clicks the "show more refinements" link.
`refinementsShown`	A string that sets the amount of refinements to return, from the following values: • `none` — returns no refinements. • `some` — returns `numRefinements` refinements. • `all` — returns `maxNumRefinements` refinements.

Property name	Description
showMore	(Deprecated) A Boolean indicating whether to display the maxNumRefinements number of menu items. When this value is false, the number of menu items generated is limited by numRefinements, and a "show more refinements" link is generated. This value should be set using showMoreIds URL parameter when the "show more refinements" link is selected.
useShowMoreIdsParam	(Deprecated) A Boolean that sets whether to use the showMoreIds URL parameter when determining how many refinements to display. If false, the showMore property on the RefinementMenuConfig object is used instead. If this property is set to true, refinements cannot be collapsed. Defaults to true.
whyPrecedenceRuleFired	If query debugging features are enabled, this property enables debugging information about why precedence rules fired on a query in a DGraph.WhyPrecedenceRuleFired property for each root dimension value. For additional information, see "About query debugging results in the reference application."

Notes on sorting

The static sort option is described as "Alphanumeric" sorting in the Experience Manager user interface for the default Refinement Menu cartridge. Dimension values are ordered alphanumerically within a dimension by default, however it is possible to manually specify a dimension order (for example, using the Dimension Values editor in the Oracle Commerce Workbench). This custom dimension value order is used when static sorting is specified. To ensure alphanumeric sorting of dimension values, do not specify a custom dimension value order.

Dynamic refinement ranking is incompatible with displaying disabled refinements for a dimension. In the default Refinement Menu cartridge, the option to show disabled refinements is not available unless the content administrator has explicitly selected static sorting.

MDEX Engine configuration for Guided Navigation

No special configuration is necessary to enable Guided Navigation, however, there is some static configuration that affects the display of refinements.

Static configuration

In the Dimension editor in Developer Studio, you can configure dimensions to be:

- *multiselect* — A multiselect dimension enables a user to select more than one refinement at the same time. You can specify whether the navigation results when multiple refinements are selected are treated as a Boolean AND or Boolean OR on a per-dimension basis.
- *hidden* — A hidden dimension does not display in Guided Navigation; however, users can still search for records based on their dimension values in a hidden dimension.

You can also configure the following refinement behavior on a per-dimension basis:

- *dynamic refinement ranking* — Dynamic ranking returns refinements based on their popularity (number of associated record results for each refinement). This is a default setting that can be overridden by the content administrator in Experience Manager.
- *refinement statistics* — Enabling refinement statistics returns the number records (or aggregated records) are associated with each refinement so that this information can be displayed in the application.

Additionally, you can designate specific dimension values as inert. For more information about these configuration options, refer to the *MDEX Engine Basic Development Guide*.

Cartridge handler configuration for the Refinement Menu cartridge

The Refinement Menu cartridge handler extends the `NavigationCartridgeHandler`. The application-wide default configuration in the Assembler context file determines the behavior of collapsed dimensions and "show more" and "show less" links, and can be set to enable or disable the precedence rule debugging feature if query debugging features are enabled.

The cartridge handler uses a `contentItemInitializer` to merge the layered configuration. The included `requestParamMarshaller` bean enables URL request-based configuration for the cartridge, which is required for disabling or enabling the full list of refinement results returned when the end user clicks the "show more refinements" link.

Template configuration for the Refinement Menu cartridge

The Refinement Menu cartridge template allows a content administrator to configure which dimension to query for the cartridge and how many results should display. It also allows control over boosted and buried dimension refinements, in order to modify the order in which dimensions display to the end user.

The Refinement Menu cartridge template allows a content administrator to configure the following properties on the configuration model:

- `dimensionId`
- `sort`
- `showMoreLink`
- `moreLinkText`
- `lessLinkText`
- `numRefinements`
- `maxNumRefinements`
- `boostRefinements`
- `buryRefinements`

In addition, the cartridge template includes the following pass-through property:

Property name	Description
`dimensionName`	The `name` of the string property that represents the dimension name. This is required by the Dimension Selector editor to enable a content administrator to select a dimension by name, rather than by ID.

URL request parameters for the Refinement Menu cartridge

You can configure the Refinement Menu cartridge at runtime by setting the value of the `DYNAMIC_REFINEMENT_MENU_CONFIG` property on the `RefinementMenuRequestParamMarshaller` via the `Nrmc` URL parameter.

The sample cartridge renderer includes logic for displaying the `maxNumRefinements` number of results when a user clicks on the "show more refinements" link.

Property name	URL parameter	Description
`DYNAMIC_REFINEMENT_MENU_CONFIG`	`Nrmc`	The `Nrmc` parameter takes multiple arguments allow you to configure dimension refinement behavior in the cartridge.
`showMore`	`ShowMoreIds`	(Deprecated) A Boolean indicating whether to display the `maxNumRefinements` number of menu items. Use the `refinementsShown` property if you are refactoring your code or developing a new application.

About `Nrmc` URL parameter syntax

The `Nrmc` parameter takes the following values:

- Dimension ID — Required. The ID of the dimension you wish to configure.
- `+show:<value>` — Required; `<value>` is the value to pass to the `refinementsShown` property on the configuration object.

The configuration for each dimension is separated by a vertical pipe, as in the example below:

```
20001+show:all|20002+show:some
```

Note

You can also use the notation used with the Presentation API, for example: `Nrc=id+10074+expand+true+more+true`. For more information about this notation, see the *MDEX Engine Basic Development Guide*.

Navigation Container

The Navigation Container is provided as an alternative the refinement menu cartridge for implementations using Oracle Guided Search with the packaged services. It enables you to retrieve the full list of available dimension refinements for a dimension query.

The response model for the Navigation Container includes a list of `RefinementMenu` objects that each include the records within a dimension refinement. The `NavigationContainerHandler` handles the "show more refinements" link and associated link Action for each of these refinements, and also controls whether to display debugging information.

Configuration model for the Navigation Container

The Navigation Container configuration model includes the `List<String>` property of dimension IDs that are returned with the response model. Since it is a dimension navigation feature, it includes a `whyPrecedenceRuleFired` property that can be used for debugging precedence rule behavior in your application.

The configuration model for this cartridge is `NavigationContainerConfig`. It includes the following properties:

Property name	Description
`showMoreIds`	A List of dimension IDs to return as expanded lists of available refinements. Any dimension refinements not included in this List are returned in the default, shorter form output by the MDEX Engine.
`moreLinkText`	A string representing the text to use for the "show more refinements" link. The same string is used for each of the included dimension refinements.
`lessLinkText`	A string representing the text to use for the "show fewer refinements" link. The same string is used for each of the included dimension refinements.
`refinementsShownByDefault`	A Boolean indicating whether the refinement menus should be fully expanded. Defaults to `true`. When using a dataset that includes dimensions with a large number of refinements, you should set this to `false`.
`refinementsShown`	A string that sets the amount of refinements to return on each refinement menu, from the following values: • `none` — returns no refinements. • `some` — returns `numRefinements` refinements.
`useShowMoreIdsParam`	(Deprecated) A Boolean that sets whether to use the `showMoreIds` URL parameter when determining how many refinements to display. If `false`, the `showMore` property on the `RefinementMenuConfig` object is used instead. If this property is set to `true`, refinements cannot be collapsed. Defaults to `true`.
`whyPrecedenceRuleFired`	If query debugging features are enabled, this property enables debugging information about why precedence rules fired on a query in a `DGraph.WhyPrecedenceRuleFired` property for each root dimension value. For additional information, see "About query debugging results in the reference application."

Cartridge handler configuration for the Navigation Container

The Navigation Container handler extends the `NavigationCartridgeHandler`. The application-wide default configuration in the Assembler context file determines the behavior of collapsed dimensions and "show more" and "show less" links, and can be set to enable or disable the precedence rule debugging feature if query debugging features are enabled.

The cartridge handler uses a `contentItemInitializer` to merge the layered configuration. The included `requestParamMarshaller` bean enables URL request-based configuration for the cartridge, which is required for modifying the properties on the response model through URL parameters.

URL request parameters for the Navigation Container

Because the Navigation Container returns a list of `RefinementMenu` objects, it takes the same `Nrmc` URL parameter as the Refinement Menu cartridge.

Property name	URL parameter	Description
`DYNAMIC_REFINEMENT_MENU_CONFIG`	`Nrmc`	The `Nrmc` parameter takes multiple arguments allow you to configure dimension refinement behavior in the cartridge.
`whyPrecedenceRuleFired`	`whyPrecedenceRuleFired`	If query debugging is enabled for the reference application, this property allows you to include debugging information about why precedence rules fired on a query in a `DGraph.WhyPrecedenceRuleFired` property for each dimension value.

For details on configuring the `Nrmc` parameter, see "URL request parameters for the Refinement Menu cartridge."

Breadcrumbs

The Breadcrumbs cartridge displays the parameters defining the search or navigation state for the current set of search results.

The response model for this cartridge is `Breadcrumbs`, which may contain `SearchBreadcrumb`, `RefinementBreadcrumb`,`RangeFilterBreadcrumb`, and `GeoFilterBreadcrumb` objects as appropriate. Each breadcrumb contains information about search or navigation selections that the end user has made, and provides links to remove that selection from the filter state.

The Breadcrumbs cartridge does not have any associated Experience Manager configuration options or MDEX Engine configuration.

Cartridge handler configuration for Breadcrumbs

The Breadcrumbs cartridge handler extends the `NavigationCartridgeHandler`, but otherwise does not require any additional configuration.

Results cartridges

The following sections provide an overview of the configuration models for features that display search results in the reference implementation.

Results list

The Results List cartridge displays search and navigation results in a list view.

The response model for this cartridge is `ResultsList`, which contains a list of `Record` objects and `SortOptionLabel` objects that enable the end user to select from a set of pre-defined sort orders.

About the order of records in the record list

The order of records returned by the MDEX Engine is determined by a sort key or relevance ranking strategy depending on the type of query that returns the results.

Relevance ranking is applied when the query includes a text search. Record *sorting* is applied to all other queries including navigation queries. The sort options that are available to the end user in the application represent static sort orders that are not based on relevance to any search terms.

Record boost and bury

Record boost and bury is a feature that enables fine-grained re-ordering of records within search or navigation results. With record boost, you can assign records to ranked strata, with those in the highest stratum being shown first, those in the second-ranked stratum shown next, and so on. With record bury, you can assign records to strata that are ranked much lower relative to others. This boost/bury mechanism therefore lets you manipulate ranking of returned record results in order to promote or push certain records to the top or bottom of the results list. The records in each stratum are defined as a set of specific records or a navigation state that the records must satisfy. A record is assigned to the highest stratum whose definition it matches, so boosting takes precedence over burying. Record boost and bury apply regardless of whether the records returned are the results of a search or navigation query.

The core Results List cartridge enables the content administrator to specify one set of records to boost and one set of records to bury. Boost and bury are applied to the result list before any additional sorting or relevance ranking modules. For more information about record boost and bury, refer to the *MDEX Engine Developer's Guide*.

Configuration model for the Results List cartridge

The Results List configuration model allows you to configure the number and sorting of records returned by a search or navigation query. Additionally, it includes `whyMatchEnabled` and `whyRankEnabled` properties that can be used for debugging the set of records returned for a query.

The configuration model for this cartridge is `ResultsListConfig`. It includes the following properties:

Property name	Description
`recordsPerPage`	An integer that controls the number of results to display per page. This value can be set using `Nrpp` URL parameter.
`recordDisplayFieldName`	A String that specifies the field that stores the record's logical name.

Property name	Description
sortOption	An enumerated list of sort options on the results list available to the site visitor. Each item in this list is a SortOptionConfig with the following properties: • label — A descriptive label that displays to the site visitor in the client application • value — A sort order specified in the format <key>|<direction>, where key is the name of the property or dimension on which to sort, and the direction is 0 for ascending and 1 for descending. An empty string represents the default sort order specified by the content administrator in Experience Manager. You can set this value via the Ns URL parameter.
sortRequestParameter	A String that specifies the selected Sort.
includePrecomputedSorts	A Boolean that specifies whether to return precomputed sorts. Defaults to false. If you do not set this to true, any calls to the getPrecomputedSorts() method return an empty list.
relRankStrategy	(Optional) The Relevance Ranking Strategy. If you specify a Relevance Ranking Strategy without setting relRankTerms, relRankKey, or relRankMatchMode, your Relevance Ranking strategy will apply to the results from the current search filter. This setting is ignored if an end user explicitly selects a sort.
relRankKey	(Optional) The Relevance Ranking key to use with the selected Relevance Ranking strategy. This can be a search interface, dimension, or property set in the MDEX Engine. You must set a relRankStrategy and relRankTerms if you specify a value for this property.
relRankTerms	(Optional) Relevance Ranking terms, delimited by a + sign. These can be different from the terms in the search filter. You must set a relRankStrategy and relRankKey if you specify a value for this property.
relRankMatchMode	(Optional) The match mode that determines the subset of results to apply Relevance Ranking to. You must set a relRankStrategy if you specify a value for this property.
boostStrata	An ordered list of CollectionFilters that enable items to be boosted to the top of the results list. This setting is ignored if an end user explicitly selects a sort.
buryStrata	An ordered list of CollectionFilters that enable items to be buried at the bottom of the results list. This setting is ignored if an end user explicitly selects a sort.

Property name	Description
subRecordsPerAggregateRecord	The number of sub-records to return for any aggregated records in the results list. This property should have one of the following values: • ZERO — Sub-records are not returned. • ONE — A single representative record is returned. • ALL — All sub-records are returned. The default value is ONE. For best performance, Oracle recommends that you use ZERO or ONE.
offset	An integer record offset for the result list. This property defaults to 0 and is used for paging. This value can be set using No URL parameter.
includeDerivedProperties	For aggregated records, returns all derived properties to the end user. Defaults to true.
includeSnippetedProperties	For aggregated records, returns all snippeted properties to the end user. Defaults to true.
includeGeoDistanceProperties	For aggregated records, returns all geodistance properties to the end user. Defaults to true. Geocode properties are not calculated by default by MDEX . You must specify one or more geo location filters(Nf) or geo distance sorts(Ns) for these properties to be calculated. For information about geocode properties, see the *MDEX Engine Developer's Guide*.
fieldNames	A list of record fields to pass through from each record to the Record output model of the ResultsListHandler.
subRecordFieldNames	For aggregated records, a list of sub-record fields to pass through from each sub-record to the Record output model of the ResultsListHandler.
whyMatchEnabled	If query debugging features are enabled, this property enables debugging information about why each record matched the search and navigation state. For additional information, see "About query debugging results in the reference application."
whyRankEnabled	If query debugging features are enabled, this property enables debugging information about why each record was ranked in the given order. For additional information, see "About query debugging results in the reference application."

Note

You only need to set the relRankKey, relRankTerms and relRankMatchMode properties if you wish to apply relevance ranking to values other than those specified in the search filter, or to the results of an EQL expression.

MDEX Engine configuration for the Results List cartridge

Your MDEX Engine configuration for your application allows you to configure which properties and dimensions should display in the results list view, optimize certain properties to use for sorting records, and specify a default sort order.

Dynamic configuration

In the Property and Dimension editors in Developer Studio, you can specify which properties and dimensions are returned for the record with the record list. This configuration can be overridden in the cartridge handler configuration. For more information about configuring the display of properties and dimensions for the record list, refer to the *Developer Studio Help*.

Static configuration

Although you can sort on any property or dimension at query time, it is also possible to optimize a property or dimension for sorting in Developer Studio. This controls the generation of a precomputed sort, which you can retrieve on the `ResultsListConfig` object by using the `getPrecomputedSorts()` method. For more information about precomputed sorts, refer to the *MDEX Engine Developer's Guide*.

Dgidx flags

You can specify the default sort order for records as a flag in Dgidx. For more information about Dgidx flags and sorting, refer to the *MDEX Engine Developer's Guide*.

The Deployment Template configuration for the Discover Electronics reference application does not specify a default sort key.

Cartridge handler configuration for the Results List cartridge

The Results List cartridge handler extends the `NavigationCartridgeHandler`. The application-wide default configuration in the Assembler context file specifies default sort options, relevance ranking strategy, and record and sub-record properties to pass through to the cartridge handler response model. It also allows you to enable or disable debugging features if query debugging features are enabled.

The cartridge handler uses a `contentItemInitializer` to merge the layered configuration. The included `requestParamMarshaller` bean enables URL request-based configuration for the cartridge.

Template configuration for the Results List cartridge

The Results List template allows a content administrator to configure the main results of a search or navigation query based on the site visitor's filter state. Configuration options include sort order, boost/bury, and number of records to display per page.

The Results List cartridge template allows a content administrator to configure the following properties on the configuration model:

- `recordsPerPage`
- `sortOption`
- `relRank`
- `boostStrata`

- buryStrata

URL request parameters for the Results List cartridge

End user configuration is passed to the configuration model as URL parameters. This allows application end users to specify how records should be displayed and sorted in order to customize their navigation experience.

For most of the properties on the configuration model, the cartridge renderer in the reference implementation respects the values set at the cartridge handler or template level. The `offset` value is used to control paging display.

Property	URL Parameter	Description
`recordsPerPage`	`Nrpp`	The cartridge renderer uses this property to enable an application end user to set their own limit on records to display per page.
`sortOption`	`Ns`	This parameter enables you to override sort options on a per-query basis.
`offset`	`No`	This parameter enables you to control record display when paging.
`relRankKey`	`Nrk`	(Optional) The Relevance Ranking key. You must set a `relRankStrategy` on the cartridge to use this parameter. You must also specify `relRankTerms`.
`relRankTerms`	`Nrt`	(Optional) Relevance Ranking terms, delimited by a + sign. You must set a `relRankStrategy` on the cartridge to use this parameter. You must also specify a `relRankKey`.
`relRankMatchMode`	`Nrm`	(Optional) The match mode that determines the subset of results to apply Relevance Ranking to. You must set a `relRankStrategy`, `relRankKey`, and `relRankTerms` if you specify a value for this property.
`whyMatchEnabled`	`whymatch`	If query debugging is enabled for the reference application, this property enables you to include record matching information about a per-query basis, rather than at the cartridge handler level.
`whyRankEnabled`	`whyrank`	If query debugging is enabled for the reference application, this property enables you to include record ranking information about a per-query basis, rather than at the cartridge handler level.

Note

The `Nrk`, `Nrt`, and `Nrm` parameters take precedence over any relevance ranking declaration in the `Ntk`, `Ntt`, and `Ntx` parameters.

Enabling snippeting in record results

The Assembler can return snippets (an excerpt from a record property that contains the user's search terms and the surrounding context) for display in results lists.

Snippeting is configured as part of a search interface. You can enable snippeting on one or more properties in a search interface, typically properties that contain multiple lines of text.

To enable snippeting in record results:

1. Enable snippeting on one or more properties in the relevant search interface.

   For more information about configuring snippeting, see the *MDEX Engine Developer's Guide*.

2. In the Results List response model configuration, specify the property `includeSnippetedProperties`, as in the following example:

```
 <bean id="resultsListDefaultConfig" scope="prototype"
class="com.endeca.infront.cartridge.ResultsListConfig">
   <property name="includeSnippetedProperties" value="true" />
                          <property name="includeDerivedProperties" value="true" />
   <property name="fieldNames">
   <list>
    <value>product.id</value>
    <value>product.code</value>
    <value>product.name</value>
    <value>product.brand.name</value>
    <value>product.short_desc</value>
    <value>product.price</value>
    <value>product.img_url_thumbnail</value>
    <value>product.review.avg_rating</value>
    <value>product.review.count</value>
   </list>
  </property>
  <!-- additional elements omitted from this example -->
 </bean>
```

The snippet is returned as a string property on the response model for each record for display by the renderer.

Record details cartridges

The following section provides an overview of the configuration model for record detail features in the reference implementation.

Record details page

The Record Details page displays detailed information about a specific record.

The response model for this cartridge is `RecordDetails`, which contains a single `Record`.

The rendering logic for a record details page is expected to be highly customized for each site, in order to display the relevant record information and provide additional functionality such as bookmarking or initiating a purchase transaction.

Configuration model for the Record Details cartridge

The Record Details configuration model allows you to configure which properties on the record should be passed through to the output model of the cartridge handler, so that the renderer can display them.

The configuration model for this cartridge is `RecordDetailsConfig`. It includes the following properties:

Property name	Description
`fieldNames`	A list of record fields to pass through from the record to the `Record` output model of the `RecordDetailsHandler`.
`subRecordFieldNames`	For aggregated records, a list of sub-record fields to pass through from each sub-record to the `Record` output model of the `RecordDetailsHandler`.

MDEX Engine configuration for the Record Details page

No special configuration is required the display of record details, but you can specify what information you want to display on the record page.

Dynamic configuration

You can specify which properties and dimensions are returned with the record for a record details page in Developer Studio. For more information about configuring the display of properties and dimensions for record details, refer to the *Developer Studio Help*.

Cartridge handler configuration for the Record Details cartridge

The Record Details cartridge handler extends the `NavigationCartridgeHandler`, but otherwise does not require any additional configuration.

Template configuration for the Record Details cartridge

The Record Details cartridge in the Discover Electronics application does not require any configuration in Experience Manager. The cartridge can be placed on a Record Details page to display detailed information about a record.

Content and spotlighting cartridges

The following sections provide an overview of the configuration models for features that enable content spotlighting in the reference implementation.

Record Spotlight

The Record Spotlight cartridge can promote either specific featured records or a set of dynamic records based on a navigation state.

The response model for this cartridge is `RecordSpotlight`, which includes a list of `Record` objects and an optional action to show all records (in the case of a dynamic record spotlight).

Configuration model for the Record Spotlight cartridge

The Record Spotlight configuration model allows you to configure the selected records and "See All" link within a record spotlight, as well as the record fields to pass through to the cartridge response model.

The configuration model for this cartridge is `RecordSpotlightConfig`. It includes the following properties:

Property name	Description
`maxNumRecords`	A string representing the maximum number of records that this spotlight can contain. If the content administrator designates specific records in the Experience Manager, the number of records cannot exceed the value of `maxNumRecords`. If the content administrator specifies a query, the Assembler returns no more than this number of records.
`recordSelection`	A `RecordSpotlightSelection` object that represents the records selected for spotlighting. This includes the specified filter state, sort options, and result limit.
`showSeeAllLink`	A Boolean that determines whether to display the "See All" link. The link requires a value for `seeAllLinkText` in order to display.
`seeAllLinkText`	A string representing the display text for a link that represents the navigation state of a dynamic record spotlight. If this string is not configured, no link is generated for the client application.
`includeDerivedProperties`	For aggregated records, returns all derived properties to the end user. Defaults to `true`.
`includeSnippetedProperties`	For aggregated records, returns all snippeted properties to the end user. Defaults to `true`.
`includeGeoDistanceProperties`	For aggregated records, returns all geodistance properties to the end user. Defaults to `true`. Geocode properties are not calculated by default by MDEX . You must specify one or more geo location filters(Nf) or geo distance sorts(Ns) for these properties to be calculated. For information about geocode properties, see the *MDEX Engine Developer's Guide*.
`fieldNames`	A list of record fields to pass through from the record to the `Record` output model of the `RecordSpotlightHandler`.
`subRecordFieldNames`	For aggregated records, a list of sub-record fields to pass through from each sub-record to the `Record` output model of the `RecordSpotlightHandler`.

MDEX Engine configuration for a spotlight

You can configure which properties and dimensions can be displayed in a spotlight.

Dynamic configuration

Although the content administrator can designate the records for a spotlight either by specifying a search and navigation query or by specifying individual record IDs, the Assembler query that fetches the spotlighted

records is always a navigation query (using records in the specific record case). Therefore, the configuration that determines which properties and dimensions are returned with the record for spotlighting is "show with record list." This configuration can be overridden in the cartridge handler configuration. For more information about configuring the display of properties and dimensions for the record list, refer to the *Developer Studio Help*.

Related links

- MDEX Engine configuration for the Results List cartridge (page 319)

Cartridge handler configuration for the Record Spotlight cartridge

The Record Spotlight cartridge handler extends the `NavigationCartridgeHandler`. The application-wide default configuration in the Assembler context file specifies record properties to pass through to the cartridge handler response model.

Template configuration for a record spotlight

A Record Spotlight cartridge enables a content administrator to specify a set of contextually relevant records to spotlight on a particular page.

The Record Spotlight cartridge template allows a content administrator to configure the following properties on the configuration model:

- `maxNumRecords`
- `recordSelection`
- `showSeeAllLink`
- `seeAllLinkText`

These properties are configured using the Spotlight Selection editor.

In addition, the cartridge template includes the following pass-through property:

Property name	Description
`title`	A title that the content administrator can specify to display for this cartridge in the front-end application.

Media Banner

The Media Banner cartridge displays video or images to the site user and can be configured to link to a static page, a single record, or a specified navigation state.

The response model for this cartridge is `MediaBanner`, which includes a `MediaObject` and an `ActionLabel` that contains a destination link.

Configuration model for the Media Banner cartridge

The configuration model for the Media Banner cartridge includes a media object and an associated link.

The configuration model for this cartridge is `MediaBannerConfig`. It includes the following properties:

Property name	Description
`media`	The `MediaObject` representing the image or video asset to display in the application.
`link`	The `LinkBuilder` object used to construct a link to a navigation state or a static page within the application.

MDEX Engine configuration for a media banner

No special configuration is required for the media banner, but your MDEX Engine configuration will affect the display of records in the link selector when setting a navigation state or choosing a specified record.

Dynamic configuration

You can specify how records are sorted and which properties and dimensions are returned with a record in Developer Studio. For more information about configuring record sorting and display, refer to the *Developer Studio Help*.

Cartridge handler configuration for the Media Banner cartridge

The Media Banner cartridge handler extends the `NavigationCartridgeHandler`, but otherwise does not require any additional configuration.

Template configuration for the Media Banner cartridge

The Media Banner enables the content administrator to use the media selector and link editor to create a media banner that links to a specified page, selected record, or dynamic navigation state.

The Media Banner cartridge template allows a content administrator to configure the following properties on the configuration model:

- `media`
- `link`

In addition, the cartridge template includes the following pass-through property:

Property name	Description
`imageAlt`	(Optional) The alt-text to display when the end user hovers over the media asset in the application.

For detailed information about the properties within the `media` and `link` properties, consult the Javadoc for the `MediaObject` and `LinkBuilder` classes.

Dynamic triggering cartridges

The following sections contain information about features related to triggering content items based on the user's context.

The dynamic slot feature is typically used to trigger a cartridge independently from the page that contains it, although the Discover Electronics application uses the same mechanism to trigger entire pages by programmatically creating a content slot configuration and passing it to the Assembler.

About dynamic slots

A dynamic slot is a generic mechanism that enables content administrators to manage the content for specific sections of an Experience Manager-driven page independently from the overall page.

There two main scenarios for using dynamic slots:

- **To share content across different pages.** In this case, the triggers on the content items that populate the slot are more general or orthogonal to the trigger criteria for the page. For example, a header cartridge may be shared across an entire site if it is referenced from every page and has an "Applies at all locations" trigger. A promotion may be configured with a user segment trigger and display when a site visitor who belongs to the specified user segment browses to any of the pages that references the collection that contains the promotion.

- **To create variants of a page.** In this case, the triggers on the content items that populate the slot are more specific than the trigger criteria for the page. (Typically, they would "inherit" the parent content item's triggers and add additional criteria for the variable content.)

Following are some specific use cases for dynamic slots:

- A brand manager needs to control the banner images that display throughout the site. This is a different person from the merchandiser who typically manages pages in Experience Manager.

- A brand manager needs to be able to specify the images that display at a particular navigation state (for example, Digital Cameras > Samsung) even if there is no specific landing page for that navigation state.

- A merchandiser wishes to display promotions in the menu area based on more specific trigger criteria than those that apply to the page as a whole. For example, one could create a page to use as a base for all "Digital Cameras" pages, and populate the menu sections with more specific content based on the brand, price range, or other dimensions. This model enables content reuse for most of the content within a page with page-specific overrides for subsections as needed. It removes the need to create many individual pages for each specific combination of triggers.

- A merchandiser wishes to display promotions in the menu area based on trigger criteria that are simply different from those on the page as a whole. For example, there might be a "Back to School" special for a particular time frame that applies to all pages within a category or even the entire site. This model enables content reuse for individual sections across a variety of pages. The reusable sections are managed in a central location so that updates immediately take effect across all the pages that include the reused content, rather than having to edit each one manually.

Dynamic slot prerequisites

The dynamic slot feature enables content administrators to populate a section of a content item with content from a different collection in Experience Manager. As a prerequisite, your application must include a collection with the appropriate content type for populating an administrator's dynamic slot cartridge.

Note

If a content administrator attempts to populate a dynamic slot in a given collection with a content item from the same collection and creates a circular reference, the Assembler detects the conflict during preprocessing and returns the content item with an `@error` property.

Request Event Attributes

The `RequestEvent` and `NavigationEventWrapper` classes support getting and setting common search and navigation information about a request event. This Appendix provides a reference table of out-of-the-box attributes that you can retrieve or set on a `RequestEvent` object.

Base request event attributes

The following describes the base schema for an Assembler request event.

The `RequestEvent` class includes getter and setter methods for each of these attributes.

Attribute	Type	Description
`endeca:requestId`	`String`	The unique identifier for a request. To retrieve this information, you must register an implementation of `RequestIdProvider` in the request event adapter constructor.
`endeca:sessionId`	`String`	The unique identifier for a browser session. To retrieve this information, you must register an implementation of `SessionIdProvider` in the request event adapter constructor.
`endeca:assemblyStartTimestamp`	`long`	The time (in milliseconds from POSIX Epoch) that the `assemble()` method started
`endeca:assemblyFinishTimestamp`	`long`	The time (in milliseconds from POSIX Epoch) that the `assemble()` method finished

Navigation cartridge request event attributes

The following describes the schema for an Assembler navigation cartridge request event. These fields are in addition to those described for the base request.

The `NavigationEventWrapper` class includes getter and setter methods for each of these attributes.

Attribute	Type	Description
endeca:autocorrectTo	String	The suggested auto-correct term, if it triggers for the request.
endeca:contentPath	String	The content path of the page corresponding to the request.
endeca:didYouMeanTo	List <String>	The suggested "Did You Mean" term, if it triggers for the request.
endeca:dimensions	List <String>	The dimension names selected for navigation.
endeca:dimensionValues	List <String>	The dimension value names selected for navigation.
endeca:eneTime	Long	The time, in milliseconds, that it takes the MDEX Engine to run the query.
endeca:numRecords	Long	The number of records returned for the request.
endeca:numRefinements	Integer	The number of selected refinements.
endeca:recordNames	List <String>	The names of the records returned by the request. To populate this attribute, the recordDisplayFieldName property on the ResultsListConfig object must be set to the name of the field that contains record names.
endeca:recordSpec	String	The record specifier for a selected record.
endeca:requestType	RequestType	The type of request. Possible values are: • T - Root navigation • N - Navigation only • S - Search only • SN - Search, then navigation • R - Record detail • UNKNOWN - Unknown
endeca:searchKey	String	The search key for the current navigation state.

Attribute	Type	Description
`endeca:searchMode`	`String`	The search mode for the request.
`endeca:searchTerms`	`String`	The search terms for the request.
`endeca:siteRootPath`	`String`	The site root path of the page corresponding to the request.
`endeca:sortKey`	`List <String>`	The sort keys for the request. Each key is a `String` with the format `fieldName|<Descending|Ascending>` .
`endeca:spotlights`	`List <String>`	The list of spotlights triggered for the request.