Endeca® Content Assembler API

Developer's Guide for Java

Version 2.1.1 • December 2011

Contents

Copyright and disclaimer

Preface

Endeca® InFront enables businesses to deliver targeted experiences for any customer, every time, in any channel. Utilizing all underlying product data and content, businesses are able to influence customer behavior regardless of where or how customers choose to engage — online, in-store, or on-the-go. And with integrated analytics and agile business-user tools, InFront solutions help businesses adapt to changing market needs, influence customer behavior across channels, and dynamically manage a relevant and targeted experience for every customer, every time.

InFront Workbench with Experience Manager provides a single, flexible platform to create, deliver, and manage content-rich, multichannel customer experiences. Experience Manager allows non-technical users to control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

At the core of InFront is the Endeca MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. InFront Integrator provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. InFront Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Endeca InFront, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

About this guide

This guide describes the major tasks involved in developing an Endeca application using the Content Assembler API for Java.

This guide assumes that you have read the *Endeca Commerce Suite Getting Started Guide* or *Endeca Publishing Suite Getting Started Guide* and that you are familiar with Endeca's terminology and basic concepts.

This guide covers only the features of the Content Assembler API for Java, and is not a replacement for the available material documenting other Endeca products and features. For a list of recommended reading, please refer to the section "Who should use this guide."

Who should use this guide

This guide is intended for developers who are building Endeca applications using the Content Assembler API for Java.

If you are a new user of the Endeca Commerce Suite or Endeca Publishing Suite and you are not familiar with developing Endeca applications, Endeca recommends reading the following guides prior to this one:

1. *Endeca Commerce Suite Getting Started Guide* or *Endeca Publishing Suite Getting Started Guide*
2. *Endeca Basic Development Guide*

3. *Endeca Advanced Development Guide*
4. *Page Builder Developer's Guide*

If you are an existing user of the Endeca Commerce Suite or Endeca Publishing Suite and you are familiar with developing Endeca applications, Endeca recommends reading the following guides prior to this one:

1. *Endeca Commerce Suite Getting Started Guide* or *Endeca Publishing Suite Getting Started Guide*
2. *Page Builder Developer's Guide*

Remember: All documentation is available on the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

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 Endeca Customer Support

The Endeca Support Center provides registered users with important information regarding Endeca software, implementation questions, product and solution help, training and professional services consultation as well as overall news and updates from Endeca.

You can contact Endeca Standard Customer Support through the Support section of the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Chapter 1

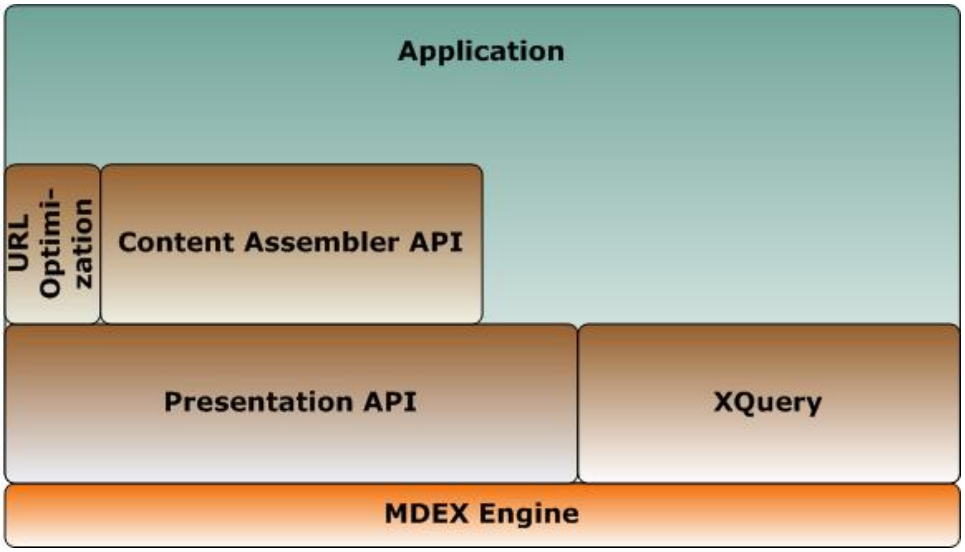
Introduction to the Content Assembler API

This section provides an overview of the Content Assembler API for Java and the associated reference application.

Overview of the Content Assembler API

The Content Assembler API for Java is used in conjunction with the Presentation API for Java and other Endeca APIs to build configurable Web applications.

The Content Assembler API is designed primarily for search and navigation queries and returns dynamic content if any dynamic pages are triggered by those queries. The Content Assembler API uses the Presentation API for Java to query the MDEX Engine and provides convenient methods for accessing the content tree that is returned as part of the query results. This content tree reflects the page configuration created by a content administrator in the Page Builder. The tree may contain results from additional queries executed by the Content Assembler that are used to populate page sections based on the configuration returned for the initial query.



Because the Content Assembler uses the `ENEQuery` and `ENEConnection` objects from the Presentation API, all queries to the MDEX Engine can be sent through the Content Assembler API. It is also possible to access the `ENEQueryResults` object through the Content Assembler API and use the Presentation API methods to process query results. Note that only search or navigation queries

that trigger a dynamic page return a content tree. All other types of queries, including record queries or dimension search queries, return a null content tree.

In addition, an Endeca application built with the Content Assembler API can also use the URL Optimization API, available as part of the optional Search Engine Optimization Module. The URL Optimization API also works with the Presentation API to enable developers to create application links using directory-style URLs with embedded keyword metadata.

Applications built on top of the MDEX Engine version 6.1 or later can also leverage the MDEX API through XQuery, available as part of the Advanced Query Module. There is no explicit support for XQuery within the current version of the Content Assembler; that is, the Content Assembler does not use the MDEX API through XQuery to process queries to the MDEX Engine. However, XQuery for Endeca enables developers to extend MDEX Engine functionality through custom XQuery modules.

API class model overview

The Content Assembler API consists of three packages, `com.endeca.content`, `com.endeca.content.ene`, and `com.endeca.content.assembler`.

The `com.endeca.content` package contains the core classes and interfaces for the Content Assembler API:

Class	Description
`ContentManager`	A service for creating and managing dynamic content queries.
`ContentQuery`	An object used for executing content queries.
`ContentResults`	An object containing the results of an executed `ContentQuery`.
`ContentItem`	A single content item from an instance of a `ContentResult` object.
`ContentItemList`	A list of content items which includes two additional properties: the type of the items and the maximum number of content items allowed.
`Property`	A property value contained within a `ContentItem` object.
`ContentExceptionFactory`	Provides convenience methods for constructing `ContentException` instances.
`ContentException`	Represents an exception when interacting with the Content Assembler API.
`InvalidQueryException`	Indicates that the specified query is invalid and cannot be executed.
`InitializationException`	Represents an unrecoverable exception while initializing a content assembler

The `com.endeca.content.ene` package contains an implementation of the Content Assembler API that uses the Presentation API.

Class	Description
`ENEContentManager`	An implementation of the `ContentManager` interface that creates an `ENEContentQuery`.

Class	Description
`ENEContentQuery`	An object used for executing content queries based on a `com.endeca.navigation.ENEQuery`.
`ENEContentResults`	An object containing the results of an executed `ENEContentQuery`, including the `ENEQueryResults` object returned for the query.
`RecordListProperty`	A property whose value is either an `ERecList` or an `AggrERecList`, depending on whether an aggregation key is present. Also contains the `ENEQuery` that was used to generate the record list and the corresponding `ENEQueryResults`. Information associated with the `ENEQuery` and the `ENEQueryResults` objects can be used to create a "see-all" query link for the `ERecList`.
`NavigationRecords`	This interface represents the subset of `Navigation` necessary to build a record list from the results. The results contained in instances of this interface do not necessarily match the results that are contained in the root `ENEQueryResults.getNavigation()`.

In addition to these packages, the `com.endeca.content.assembler` package provides access to core Content Assembler functionality that you can use to extend the Content Assembler. For more information, see "Extending the Content Assembler with Tag Handlers" in this guide.

Related Links

Extending the Content Assembler with Tag Handlers on page 29
The Content Assembler uses tag handlers to transform content XML into an object representation of a dynamic page. Tag handlers can be written by the Endeca community (including Endeca Professional Services, partners, or customers) in order to customize or extend the Content Assembler to process custom content XML and integrate with third-party systems.

Overview of the Content Assembler reference application

The Content Assembler reference front-end application demonstrates best practices for using the Content Assembler API to develop dynamic applications.

The Content Assembler reference application and sample project is designed to show a typical approach to building cartridges -- that is, templates and their associated rendering code -- and demonstrate how the configuration specified by the content administrator in the Page Builder can affect the display of content in the front-end application. The templates and application code are based on UI best practices developed by Endeca specifically for Guided Navigation applications.

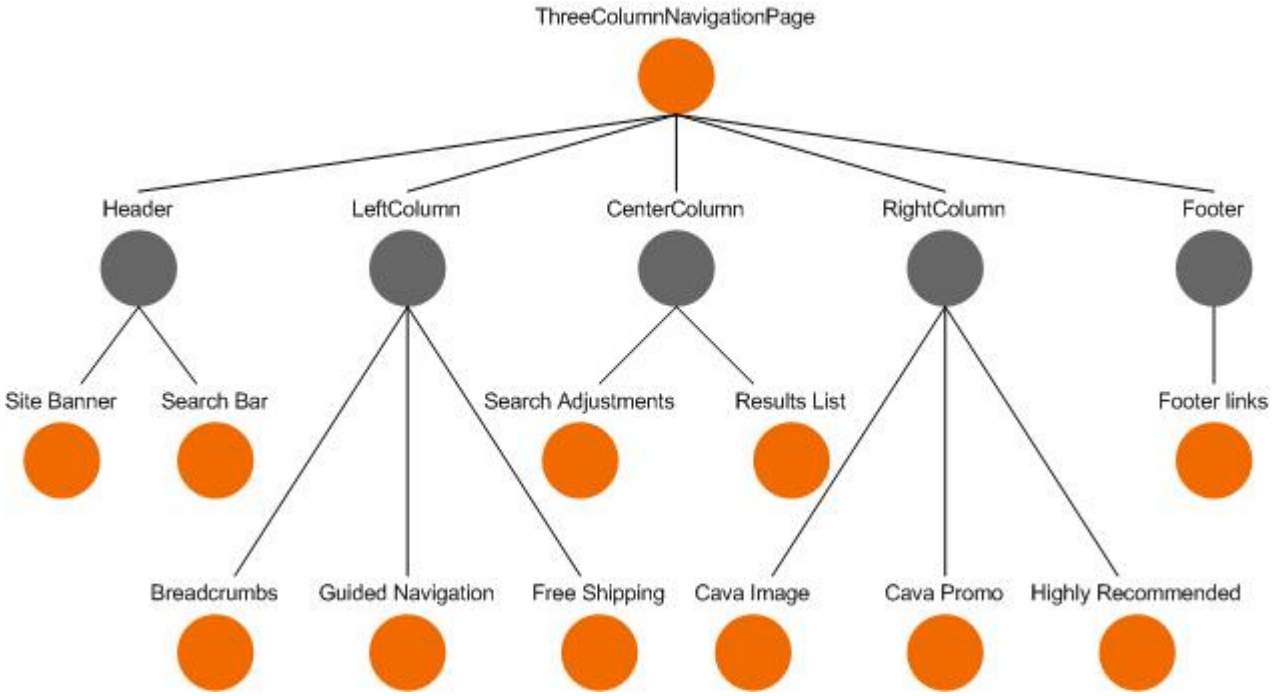
Unlike other Endeca reference applications, the Content Assembler reference application is not intended as a general-purpose data navigator. In order to show realistic examples of cartridge development, the reference application is closely tied to the sample wine data project that is provided with the Content Assembler. For this reason, it is not intended as a generic preview application for the Page Builder in Endeca Workbench.

The reference application may be used as a starting point for your own application code. You can customize it to suit your data and business requirements and extend its functionality as needed.

About handling dynamic content

Your application should contain logic to iterate through the content tree returned by the Content Assembler and pass the embedded content items to the appropriate code for rendering.

Recall that the structure of the templates you provide in the Page Builder determines the structure of the content in the page configuration. Templates enable you to specify `<ContentItem>` or `<ContentItemList>` elements that serve as place holders for the content configured by the content administrator. The diagram below shows an example of a fully configured dynamic page.



In this example, each orange dot represents a content item while the gray dots (such as Header and LeftColumn) represent content item lists. You can use both content items and content item lists in your templates, but generally only content items are actually rendered.

Because the template dictates the number and type of properties in a content item, you can write rendering code that is closely tailored to handle the content items based on a particular template. There are several ways that you can then match the content items in the content tree to the appropriate rendering code, for example:

- inspecting the `TemplateId` of the content item
- using a naming convention based on the template id
- using a string property in the template that specifies the name of the class to use for rendering content items based on the template

Content Assembler reference application for Java uses a mapping between the template id and the rendering code, specified in the `templateconfig.properties` file.

The reference application model for dynamic content

In the Java Content Assembler reference application, the controller servlet and the custom `cartridges:include` JSP tag manage the logic of finding the appropriate control to handle each content item.

The controller servlet loads the mapping of content template ids to the corresponding rendering code. This mapping is defined in the `templateconfig.properties` located in the `/WEB-INF/classes` subdirectory of your Content Assembler reference application. In a typical installation this is: `C:\Endeca\ContentAssemblerAPIs\Java\`*`version`*`\reference\ContentAssemblerRefApp\WEB-INF\classes`.

The mapping is loaded into a `java.util.Properties` object and added as an attribute on the servlet context. This `Properties` object is then accessed by `/WEB-INF/tags/cartridges/include.tag`.

The following example from the `templateconfig.properties` file shows the format of the mapping used by the controller between the template id and the path to the JSP designed to render cartridges based on that template.

```
ThreeColumnNavigationPage=/layout/ThreeColumnNavigationPage.jsp
ImageSiteBanner=/cartridges/Image.jsp
Breadcrumbs=/cartridges/Breadcrumbs.jsp
GuidedNavigation=/cartridges/GuidedNavigation.jsp
ResultsList=/cartridges/ResultsList.jsp
SearchBar=/cartridges/SearchBar.jsp
ImageBox=/cartridges/Image.jsp
ThreeRecordBox=/cartridges/ThreeRecordBox.jsp
TextBox=/cartridges/Text.jsp
TextBanner=/cartridges/Text.jsp
DimensionSearchResults=/cartridges/DimensionSearchResults.jsp
ImageBanner=/cartridges/Image.jsp
OneRecordBanner=/cartridges/OneRecordBanner.jsp
ThreeRecordBanner=/cartridges/ThreeRecordBanner.jsp
SearchAdjustments=/cartridges/SearchAdjustments.jsp
```

The `cartridges:include` custom JSP tag is defined by the `include.tag` file, located in `/WEB-INF/tags/cartridges` directory in the reference application folder. This tag dynamically loads the renderer to handle nested cartridge content based on the template id and the mapping specified in the `java.util.Properties` object.

Note that the same code may be used to handle more than one template, if the properties defined in the templates are sufficiently similar.

List of reference application cartridges

The reference application includes sample cartridges that enable configuration of a variety of front-end features.

For implementation details, refer to the templates (located in your reference application deployment at *`[appDir]`*`/config/page_builder_templates`) and the rendering code (located in `ContentAssemblerAPIs/Java/`*`version`*`/reference/ContentAssemblerRefApp/cartridges`).

Template name	Rendering code	Description
`FullWidthContent-ImageSiteBanner`	`Image.jsp`	Displays the site banner image with an optional link.
`FullWidthContent-SearchBar`	`SearchBar.jsp`	Displays the search bar.
`MainColumnContent-DimensionSearchResults`	`DimensionSearchRe¬sults.jsp`	Displays dimension search results. Content administrators can configure whether or not to display compound dimension search results.

Template name	Rendering code	Description
MainColumnContent-ImageBanner	Image.jsp	Displays an image banner with an optional link.
MainColumnContent-OneRecordBanner	OneRecordBanner.jsp	Displays one record spotlight with an image.
MainColumnContent-ResultsList	ResultsList.jsp	Displays search and navigation results in a list view.
MainColumnContent-SearchAdjustments	SearchAdjustments.jsp	Displays search adjustment messaging such as Did You Mean or spelling correction.
MainColumnContent-TextBanner	Text.jsp	Displays promotional text with a title and an optional link.
MainColumnContent-ThreeRecordBanner	ThreeRecordBanner.jsp	Displays a three record spotlight banner.
SidebarItem-Breadcrumbs	Breadcrumbs.jsp	Displays breadcrumbs appropriate to the current refinement state.
SidebarItem-GuidedNavigation	GuidedNavigation.jsp	Displays Endeca Guided Navigation with configurable display of dimensions.
SidebarItem-ImageBox	Image.jsp	Displays an image with an optional link.
SidebarItem-TextBox	Text.jsp	Displays promotional text with a title and an optional link.
SidebarItem-ThreeRecordBox	ThreeRecordBox.jsp	Displays a three record spotlight box.

Note: The JSP files in the reference application apply HTML escaping to the strings specified by the content administrator in the Page Builder. If you want to allow content administrators to enter HTML-formatted text in the Page Builder, create a separate cartridge with rendering code that does not escape HTML strings.

The reference application also includes a page template named `PageTemplate-ThreeColumnNavigationPage`, which controls the overall page content and `ThreeColumnNavigationPage.jsp` (located in `ContentAssemblerAPIs/Java/`*`version`*`/reference/ContentAssemblerRefApp/layout`), which controls the overall rendering of the page.

Connecting to a different MDEX Engine

By default the Content Assembler reference application attempts to connect to an MDEX Engine running on localhost port 15000 (the default port in the sample wine deployment). If you are running the MDEX Engine on a different host or port, you can update the configuration in the `web.xml` file.

To specify a different MDEX Engine host or port:

1. Navigate to the `WEB-INF` subdirectory of your Content Assembler reference application. In a typical installation, this is: `C:\Endeca\ContentAssemblerAPIs\Java\2.0.0\reference\ContentAssemblerRefApp\WEB-INF`.

2. Open the `web.xml` file and locate the following section:

```
<env-entry>
  <description>
  Hostname or IP address of the MDEX engine
  </description>
  <env-entry-name>MDEXHost</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>localhost</env-entry-value>
</env-entry>

<env-entry>
  <description>
  Port on which the MDEX engine is listening
  </description>
  <env-entry-name>MDEXPort</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>15000</env-entry-value>
</env-entry>
```

3. To change the host name of the MDEX Engine server, update the value of the `MDEXHost` parameter.
4. To change the port of the MDEX Engine server, update the value of the `MDEXPort` parameter.
5. Save and close the file.
6. Restart the Endeca Tools Service.

About skinning the reference application

The styling of the reference application is implemented through external CSS style sheets, which can be easily customized.

The style sheets are located in the `reference/ContentAssemblerRefApp/css` directory of your Content Assembler API installation. In a typical installation, this is `C:\Endeca\ContentAssemblerAPIs\Java\`*`version`*`\reference\ContentAssemblerRefApp\css` (on Windows) or `/usr/local/endeca/ContentAssemblerAPIs/Java/`*`version`*`/reference/ContentAssemblerRefApp/css` (on UNIX).

Each cartridge component (or type of component) in the reference application has a corresponding style sheet that controls the appearance of that component.

Chapter 2

Working with the Content Assembler API

This section provides information on writing dynamic applications in Java with the Endeca Content Assembler API.

Writing applications with the Content Assembler API

This section describes how to use the Content Assembler API for Java to query the MDEX Engine.

The Content Assembler API is used in conjunction with the Presentation API for Java; it does not replace the Presentation API. Queries submitted using the Content Assembler API must contain valid `ENEQuery` and `ENEConnection` objects from the Presentation API.

Importing API packages

There are three API packages that you must import.

- `com.endeca.content` contains the core classes and interfaces for the Content Assembler API.
- `com.endeca.content.ene` contains a `ContentManager` implementation for the Endeca Presentation API.
- `com.endeca.navigation` is the Endeca Presentation API and contains all implementation-specific classes and interfaces.

To import the necessary classes:

Add the following lines at the top of your code:

```
import com.endeca.content.*;
import com.endeca.content.ene.*;
import com.endeca.navigation.*;
```

Creating a ContentManager

You use a `ContentManager` to create a `ContentQuery` object and obtain `ContentResults`.

Note: The `ContentManager` should be scoped at a global or application level. You should not create new `ContentManager` instances for each request or query. A `ContentManager` instance is threadsafe.

To create a `ContentManager`:

1. Create a new `ENEContentManager`.

```
ENEContentManager contentManager = new ENEContentManager();
```

2. Optionally, you can enable XML validation of page configurations:

```
contentManager.setValidating(true);
```

Note: Validation can be useful in a testing environment for debugging purposes, particularly if templates are changing often. Because of the performance impact of validating content XML, this option should never be used in production. XML validation requires Java 1.5 or later, and is disabled by default.

Executing a content query and retrieving the results

A `ContentQuery` object sends dynamic content queries to the Endeca MDEX engine.

`ContentQuery` objects are created using the `ContentManager.createQuery()` method.

To execute a content query and retrieve content results:

Add code similar to the following example:

```
// Create a ContentQuery.
ENEContentQuery query = (ENEContentQuery)contentManager.createQuery();

// Configure the ContentQuery.
query.setENEQuery(new UrlENEQuery(request.getQueryString(), encoding));
query.setENEConnection(new HttpENEConnection(eneHost, enePort));

// Set the rule zone for the query. In most cases you only need one
// zone for all your landing pages. Using multiple zones can enable
// you to provide different perspectives on the same navigation state
// within your application.
query.setRuleZone("NavigationPageZone");

// Execute the query.
ENEContentResults results = query.execute();

// Get the root content item.
ContentItem content = results.getContent();

// Optionally, get the ENEQueryResults object.
ENEQueryResults eneResults = results.getENEQueryResults();
```

Note: This example uses an `ENEContentManager` to create an `ENEContentQuery` that returns `ENEContentResults` -- these are specific implementations of the Content Assembler API that use the standard Presentation API. For an example that uses the Endeca URL Optimization API, please refer to the Content Assembler reference application's `ContentQueryHandler.java` located in the `\reference\ContentAssembler¬ RefApp\WEB-INF\classes\com\endeca\content\reference` directory of your Content Assembler API installation.

This guide describes how to work with `ContentItem` objects returned by the ContentAssembler API. For information about how to work with `ENEQueryResults` objects, see the *Endeca Basic Development Guide*.

About implementing custom trigger conditions

Because the Content Assembler API retrieves page content based on Endeca's dynamic business rules functionality, pages can only be triggered on record-filtering dimension values, specific search terms, a date range, or a single user profile identifier.

These limitations can make it difficult to handle certain scenarios such as the following:

- **Search results pages.** Dynamic pages are generally configured to display based on a navigation trigger. This means however that the page for a particular location displays even if a user has entered a search term on your Web site from that location. For example, you may have set up a highly branded page to display as your site's home page (at location N=0) that does not include any record results. This page displays even if a user has performed a search from the home page location, unless a page has been configured specifically to trigger on that search term.
- **Record offset pages.** There is no simple way to explicitly trigger different content for the first page of record results (at offset=0) and for subsequent pages, with different page configurations specified by the content administrator in the Page Builder.
- **Alternate views on the same navigation state.** Use cases include A/B testing or toggling between a product details view and a customer reviews view. By default, the Content Assembler API returns a single content tree representing a dynamic page for any given navigation state or trigger condition.

There are various approaches that can be used to handle these use cases:

- Filtering landing pages based on rule properties
- Using hidden dimensions
- Using multiple rule zones
- Using multiple user profiles

Any of these strategies can be applied to the scenarios listed above. They can also be used to implement other custom trigger conditions that you may require. Which approach you use depends on the scenario you are trying to address and the specifics of your application. For guidance on selecting the appropriate option (or combination of options) and assistance with implementation, contact your Endeca representative.

About filtering landing pages based on rule properties

If you specify custom rule properties in a page template, you can use those properties to exclude certain landing pages from consideration by the MDEX Engine on a per-query basis.

Filtering based on rule properties can enable your application to implement more fine-grained trigger functionality than is available in the Page Builder.

Because the rule properties for a dynamic page are set based on the properties specified within the `<RuleInfo>` element in the page template, the content administrator must have set up a page intended for a particular trigger condition based on a template with the appropriate property. You can provide information in the template `id` (for example, `ThreeColumnPage-Search`) or `description` to help the content administrator select the appropriate template.

For the purposes of priority, pages based on templates with custom rule properties should be treated as if they have more specific trigger conditions than the same page with no such properties. (In general, pages with more specific triggering conditions should have higher priority than more generic pages.)

Because the Page Builder preview functionality cannot replicate your custom logic for filtering pages, the preview status messages may be misleading when you exclude certain pages from consideration. However, if your preview application includes the appropriate logic, the correct page displays in the preview pane even if the status messages indicate that a different page fired.

Use case: Search results

You can enable more robust handling of search results pages by creating a template that specifies a custom rule property with a key such as `search_results` and a value of `true`. The content administrator can then create search results pages based on this template. You can add logic to your application to consider these pages only for search queries (that is, queries that include `Ntt` and `Ntk` parameters). If there are no search parameters present, you can augment the query with a filter such as `Nmrf=not(search_results:true)` before you pass it to the MDEX Engine via the Content Assembler API.

For more information about working with rule properties, see "Promoting records with dynamic business rules" in the *Advanced Development Guide*.

About using hidden dimensions to trigger landing pages

You can create specialized dimensions in your application to expose additional trigger conditions.

This approach involves some additional work in your data pipeline to apply the dimension values to the records. Once this is done, the content administrator can select the trigger condition in the Page Builder using the same process as any navigation state.

Use case: Record offset

You can enable different landing pages based on record offset by creating a dimension such as Offset with dimension values such as First Page and Next Pages. During the ITL process, apply both the Offset > First Page and Offset > Next Pages dimension values to all records. The content administrator can then set up pages for each trigger condition.

You can add logic to your application to augment the navigation filter (`N` parameter) based on the record offset value (the `No` parameter).

For more information about working with dimensions, see the *Forge Guide*, *Basic Development Guide*, and the *Endeca Developer Studio Help*.

About using multiple rule zones for landing pages

Using multiple zones can enable you to provide different perspectives on the same navigation state within your application.

Because the zone for a page is set based on the `zone` attribute of the `<RuleInfo>` element in the page template, the content administrator must have set up a page intended for a particular display condition based on a template that uses the appropriate zone. You can provide information in the template `id` or `description` to help the content administrator select the appropriate template for each case.

Because the Page Builder preview functionality does not limit the query to a single zone, the preview status messages may be misleading when you use multiple zones. However, if your preview application includes the appropriate logic, the correct page should display in the preview pane even if the status messages indicate that more than one page fired.

Also note that although the Content Assembler API only retrieves the content tree from a specific zone, the results from all zones with triggered content are returned as part of the query response, so excessive use of multiple zones may lead to a noticeable increase in the size of the query response.

Use case: A/B testing

You can enable A/B testing scenarios by setting up different zones such as Control, VariableA, VariableB, and so on. You then create different templates for each zone, and the content administrator can create pages based on the different templates.

Your front-end application can set the zone for the content query based on various conditions for which you want to expose different views on the data.

For more information about setting up rule zones for landing pages, see the *Page Builder Developer's Guide*.

About using multiple user profiles for custom trigger conditions

You can use the user profile functionality to provide different views on the same navigation states.

You can set up specialized user profiles to enable content administrators to set up different pages in the Page Builder for different scenarios. However, if you are already using user profiles for other purposes, this usage may interfere with other user profile triggers.

Use case: Different front-end sites backed by the same data

You can present different views on the same data by creating different user profiles in Developer Studio such as SiteAUser and SiteBUser. In the Page Builder, the content administrator can set the user profile to use for each page.

You can add logic to your application to add the appropriate user profile to the query by using the `ENEQuery.setProfiles()` method.

For more information about setting up user profiles, see the *Endeca Developer Studio Help*. For more information about working with user profiles, see "Implementing User Profiles" in the *Advanced Development Guide*.

Building cartridges to render template-based content

Cartridges consist of cartridge templates and their associated rendering code, allowing you to separate the structure of dynamic page content from its presentation.

Building a front-end application based on cartridges involves the following tasks:

- Writing code to render content items based on each template.
- When rendering content items that contain nested content items, include code to dynamically call the appropriate code that is designed to render the nested content.

About working with content items

The `ContentResults.getContent()` method returns the root `ContentItem` object that contains dynamic page content.

The `ContentItem` class provides several methods that allow you to iterate over the content properties. However, because the properties are defined by the template on which a content item is based, you can access the content properties directly based on the property `name` attribute defined in the template.

You pass the property name as a `String` to the `ContentItem.getProperty()` method, which returns a `Property` object. A `Property` can contain any type of object returned by the MDEX Engine. The type of object depends on the property elements specified in the template. Common object types include:

- `String`
- `ERecList`
- `ContentItem`
- `ContentItemList`
- `NavigationRefinements`
- `ENEQueryResults`

Typically, you access a specific property value using `ContentItem.getProperty("`*name*`").getValue()` and cast it to the appropriate object type.

For more details about `ContentItem` methods, see the *Endeca API Reference for the Content Assembler API for Java*.

Rendering section content

Because a template defines the number and types of properties in a content item, you can write rendering code that is tailored to render the content driven by a specific template. This combination of a template and its renderer forms a cartridge.

Note: The following examples use Java Server Pages syntax for convenience. The Content Assembler API for Java is not limited to use in Java Server Pages applications and can be used with any Java application framework.

For example, if you have the following properties defined in a section template:

```
<ContentTemplate xmlns="http://endeca.com/schema/content-template/2008"
type="SidebarItem" id="TextBox">
  <!-- additional elements not shown in this example -->
  <ContentItem>
    <Name>New Text Box</Name>
    <Property name="title">
      <String/>
    </Property>
    <Property name="body">
      <String/>
    </Property>
    <Property name="link_text">
      <String/>
    </Property>
    <Property name="link_href">
      <String/>
    </Property>
  </ContentItem>
  <!-- additional elements not shown in this example -->
</ContentTemplate>
```

To render content based on this template:

1. Access the configured values from the `ContentItem` for the properties defined in the template.

```
<%
// Retrieve the appropriate ContentItem
ContentItem contentItem = (ContentItem)request.getAttribute("contentItem");

// Get the banner title from the ContentItem
String title = (String)contentItem.getProperty("title").getValue();

// Get the banner body from the ContentItem
String body = (String)contentItem.getProperty("body").getValue();

// Get the link text from the ContentItem
String textlink = (String)contentItem.getProperty("link_text").getValue();

// Get the link URL string from the ContentItem
String href = (String)contentItem.getProperty("link_href").getValue();
%>
```

2. Add code to render the page based on the design from the creative team, using the values specified in the `ContentItem` object.

```
        <div class="TextBanner">
            <div class="Title"><%= title %></div>
            <div class="Body"><%= body %></div>
            <div class="Link"><a href="<%= href %>"><%= textlink
%></a></div>
        </div>
```

The following example shows a content item list property in a page template and how the corresponding rendering code can display the results.

If the template (`PageTemplate-ThreeColumnNavigationPage.xml`) includes the following:

```
<ContentTemplate xmlns="http://endeca.com/schema/content-template/2008"
type="PageTemplate" id="ThreeColumnNavigationPage">
  <!-- additional elements deleted from this example -->
  <ContentItem>
    <Name>New Three-Column Navigation Page</Name>
    <Property name="LeftColumn">
      <ContentItemList type="SidebarItem" />
    </Property>
<!-- additional elements deleted from this example -->
  </ContentItem>
<!-- additional elements deleted from this example -->
</ContentTemplate>
```

The associated rendering code (`ThreeColumnNavigationPage.jsp`) may look similar to the following:

```
<%
// Get the root content Item. Here contentResults is the
// ENEContentResults object.
ContentItem contentItem = contentResults.getContent();

%>
<html>
<body>
 <!-- Rendering code omitted -->
 <div id="LeftColumn">
 <%
```

```
  // Get the ContentItemList off the ContentItem
  ContentItemList leftColumnItems = (ContentItemList)contentItem.getProper¬
ty("LeftColumn").getValue();

  Iterator i = leftColumnItems.iterator();

  while (i.hasNext()) {
   ContentItem item = (ContentItem)i.next();
   %>
   <!-- Access the appropriate content item properties for rendering
   this content item. Use the method of your choice to dynamically include

   rendering code that corresponds to each content item. -->
   <%

  }
 %>
 </div>
 <!-- Rendering code omitted -->
</body>
</html>
```

About rendering customized navigation refinements

Rendering customized navigation refinements requires accessing the configured `DimensionList` values from the `ContentItem` for the `NavigationRefinements` properties defined in the template.

For example:

```
ContentItem contentItem = (ContentItem)request.getAttribute("contentItem");

DimensionList orderedDimensions = (DimensionList)contentItem.getProperty("re¬
finements").getValue();
```

This code is the equivalent of getting the `DimensionList` by using `Navigation.getRefinement¬Dimensions()`, except that the dimensions returned reflect the content administrator's configuration specified in the Page Builder.

Note: If you have precedence rules defined in your application, they still apply to the customized `DimensionList`. This means that if the landing page definition specifies certain dimensions for display that should not display for that navigation state (whether it is due to precedence rules or because it is not a valid refinement), those invalid dimensions are not included in the `Dimen¬sionList` object.

The Content Assembler reference application provides a sample Endeca Guided Navigation cartridge (including rendering code) that uses a `NavigationRefinements` property.

About rendering customized results lists

If you enable content administrators to customize the display of record results, the results object returned by the Content Assembler API is different from the object returned by the Presentation API.

Recall that you can specify a `<NavigationRecords>` property in a template with a `<Navigation¬RecordsEditor>` that enables a content administrator to specify sort order, relevance ranking, and the number of records to display per page.

To render the customized navigation results, retrieve the list of records from the navigation records property, which is of type `NavigationRecords`. For example:

```
ContentItem contentItem = (ContentItem)
  request.getAttribute("contentItem");
NavigationRecords navRecs = (NavigationRecords)
  contentItem.getProperty("navigation_records").getValue();
ERecList recs = navRecs.getERecs();
```

This code is equivalent to calling `ENEQueryResults.getERecs()`, except that the records returned reflect the content administrator's configuration specified in the Page Builder.

The `NavigationRecords` object also exposes methods to access the number of records per page (and aggregated records per page) that were specified in the modified query used to retrieve the customized results. When working with customized results lists, use the `NavigationRecords` methods, rather than the analogous `ENEQuery` methods.

For example, when rendering a pager component for a customized record list, you should use `navRecs.getERecsPerPage()` because the content administrator may have specified a different number of records per page from the main query (which is reflected in `ENEQuery.getNavNu¬mERecs()`).

For further details, refer to the *Endeca API Reference for the Content Assembler API for Java*.

About customized results

The Content Assembler handles sort order, relevance ranking, and records-per-page customization slightly differently than the Java Presentation API. See the sections below for details about how the Content Assembler handles each configuration option.

The Content Assembler performs an additional query in order to retrieve the customized record results from the MDEX Engine. If no custom behavior was specified in the Page Builder, no additional query is made.

Sort order

The sort order specified by the content administrator in the Page Builder is used as a default. End users of the Web application can override this setting if you enable a control for users to specify sort order.

Relevance ranking

If the content administrator specifies both a sort order and a relevance ranking strategy for a single landing page and the query that triggers the page contains a search, the Content Assembler passes only the relevance ranking strategy on to the query to retrieve the customized navigation records. If no search is present, both the sort order and the relevance ranking strategy are passed on to the second query. In this case, the sort order overrides the relevance ranking.

The relevance ranking strategy specified by the content administrator for a landing page always overrides any other relevance ranking setting (whether it is coded as default behavior in the application or -- less typically -- specified by an end user).

Records per page

The `NavigationRecordsEditor` provides an optional interface for the content administrator to specify the number of records to return per page for a given navigation state.

The case where a content administrator has configured a value for records per page and an end user also specifies a value can lead to undefined and unexpected behaviors. For this reason, if you enable configuration of records-per-page display in the Page Builder, it is not recommended that you enable a control for end users to specify records per page in the application.

About rendering record lists

Record list properties represent the results of supplemental queries, for example, to populate promotions or Content Spotlighting cartridges.

Properties containing record list values are returned as instances of `RecordListProperty`, which is a sub-interface of `Property`. Content administrators can designate either specific records or a navigation query that returns records for spotlighting. A `RecordListProperty` that is configured to display specific featured records always returns an `ERecList`.

Note: When a cartridge is configured to display specific featured records and any of the specified record IDs are invalid, the Content Assembler API for Java returns null for that `RecordList¬Property` value.

A `RecordListProperty` that is populated with an `ENEQuery` returns either an `ERecList` or an `AggrERecList`, depending on whether the `ENEQuery` that triggered a landing page has the `NavRollupKey` property set.

If you use rollup keys for aggregated records in your application, then you must check the type of list being returned for any `RecordListProperty` in one of two ways:

- Check the type of the object returned by the record list `Property.getValue()` to determine whether it is an `ERecList` or an `AggrERecList`.
- Cast the `Property` to a `RecordListProperty`, and check the `boolean` value of `containsAg¬gregatedRecords()`.

If you prefer to render records rather than aggregated records for a Content Spotlighting cartridge on a page with a rollup key, you can render a representative record from the list of constituent records. For example, for each aggregated record, the application can retrieve the representative record as follows:

```
ERec rec = aggrERec.getRepresentative();
```

where `aggrERec` is the `AggrERec` object.

The Content Assembler reference application provides several sample spotlight cartridges that demonstrate how to render a record list property.

Generating see-all links

You can provide front-end users with a "see-all" link to display the full results set of a record query or a navigation query that was used to populate a spotlight cartridge.

The `RecordListProperty` interface has additional public fields for the corresponding `ENEQuery` and `ENEQueryResults` objects that aid in creating see-all links.

To create a see-all link:

1. Use the `ContentItem getProperty()` method to retrieve the record list Property object off a `ContentItem`:

```
Property property = contentItem.getProperty("products");

ENEQuery eneQuery = null;

if (property instanceof ERecListProperty) {

     //Cast the Property to ERecListProperty

     ERecListProperty erecListProperty = (ERecListProperty)property;

     ENEQuery eneQuery = erecListProperty.getENEQuery();


}
```

2. Use the `ENEQuery` object to create a URL link to the target record set:

```
if (null !=eneQuery) && eneQuery.containsNavQuery()) {
     String url = UrlENEQuery.toQueryString(eneQuery, request.getCharac¬
terEncoding());

}
```

Note: See-all links can only be created if the record list is generated using a navigation query. See-all links cannot be generated for result lists that are returned from record queries.

If you plan to construct URLs using the `UrlFormatter` object from the URL Optimization API, please refer to the *URL Optimization API for Java Developer's Guide* for more information. To see an example cartridge that uses a `UrlFormatter`, refer to the `ThreeRecordBanner.jsp` file located in the `cartridges` directory of your Content Assembler reference application installation directory.

Using dynamic includes to render page content

If you are using JavaServer Pages technology, you can use the `RequestDispatcher` functionality from the Java Servlet API to dynamically include rendering code to handle cartridge content.

The following example assumes that the `templateIDToRenderer` object is a `java.util.Proper¬ties` object that contains a mapping of template ID to rendering code for each of the cartridge and page templates. This template ID to rendering code mapping is loaded from an external file.

```
<ContentTemplate xmlns="http://endeca.com/schema/content-template/2008"
type="PageTemplate" id="ThreeColumnNavigationPage">
  <!-- additional elements not shown in this example -->
  <ContentItem>
    <Name>New Three-Column Navigation Page</Name>
    <Property name="Header">
      <ContentItemList type="FullWidthContent"/>
    </Property>
<!-- additional properties not shown in this example -->
    <Property name="Footer">
      <ContentItem type="FullWidthContent"/>
    </Property>
  </ContentItem>
  <!-- additional elements not shown in this example -->
</ContentTemplate>
```

You may choose not to use an external mapping file to match a template to its associated rendering code. Alternatives include constructing the name of the relevant JSP file programmatically based on properties such as the `type` and `id` of the template, or specifying the JSP code that is intended to render a template via a hidden property.

Some simple cartridges, such as image hotspots or sub-sections of other cartridges, may be rendered by their parent cartridge code. In these cases you do not need to use this procedure.

To render dynamic pages using a `RequestDispatcher`:

1. Import the `RequestDispatcher` class:

```
import javax.servlet.RequestDispatcher;
```

2. For each nested `ContentItem`, use a `RequestDispatcher` to dynamically include the appropriate code to render the content. For example:

```
ContentItem pageContent = contentRequest.getContent();
Properties templateIdToRenderer = (Properties)application.getAttribute("tem¬
plateIdToRenderer");

// Render the Header ContentItemList
ContentItemList header = (ContentItemList)pageContent.getProperty("Head¬
er").getValue();
Iterator i = leftColumnItems.iterator();

 while (i.hasNext()) {
  ContentItem item = (ContentItem)i.next();
  if(null != item) {
    String jspPath = (String)templateIdToRenderer.getProperty(item.get¬
TemplateId());
   RequestDispatcher dispatcher = request.getRequestDispatcher(jspPath);

    // pass the ContentItem object to be rendered
    request.setAttribute("LocalContentItem", item);
    dispatcher.include(request, response);

  }
 }

// Render the Footer ContentItem
ContentItem footer = (ContentItem)pageContent.getProperty("Footer").get¬
Value();
if(null != footer) {
  String jspPath = (String)templateIdToRenderer.getProperty(footer.get¬
TemplateId());
  RequestDispatcher dispatcher = request.getRequestDispatcher(jspPath);
  // pass the ContentItem object to be rendered
  request.setAttribute("LocalContentItem", footer);
  dispatcher.include(request, response);

}
```

Note: For more details about using the `RequestDispatcher`, see Sun's Java Servlet API documentation.

Chapter 3

Extending the Content Assembler with Tag Handlers

The Content Assembler uses tag handlers to transform content XML into an object representation of a dynamic page. Tag handlers can be written by the Endeca community (including Endeca Professional Services, partners, or customers) in order to customize or extend the Content Assembler to process custom content XML and integrate with third-party systems.

About tag handlers in the Content Assembler

A tag handler enables you to define your own processing logic for the content that is configured by content administrators in Page Builder.

When your application queries the MDEX Engine using the Content Assembler API, the corresponding landing page configuration is returned as part of the response in the form of content XML. The Content Assembler processes this XML, executing additional queries as needed, and returns a tree of `ContentItem` objects and their associated properties.

Each of the standard property types in Page Builder is represented by an element in XML, such as `<String>`, `<RecordList>`, or `<ContentItem>`. For each of the standard types, the Content Assembler has a standard tag handler associated with that element that processes the element into a Java object.

You can take advantage of the same mechanism to write a tag handler that processes a specific element in the content XML and returns a native Java object to the application. Community tag handlers process elements outside of the Endeca content XML namespaces (that is, `http://endeca.com/schema/content/2008` and `http://endeca.com/schema/content-tags/2008`). These elements may be either pass-through XML defined in the template or custom XML generated by a community editor. For more information about pass-through XML and community editors, refer to the *Page Builder Developer's Guide*.

The combination of custom XML and a community tag handler enables you to extend the query processing logic in the Content Assembler — for example, by executing additional queries against the MDEX Engine, or interfacing with a third-party system to return data — before returning the results to the application. Use cases for community tag handlers include the following:

- Given some XML that specifies a rollup key for a navigation query or aggregated record query, pass this key with the query to the MDEX Engine to return records for Content Spotlighting.
- Implement A/B testing for Content Spotlighting by executing different queries to the MDEX Engine for identical requests. The results of the queries are then transparently passed on to the applicaton.

- Query a third-party source for information to display on a product detail page. Examples include RSS feeds, content stored in another repository (such as a CMS), inventory information, or a recommendation engine.

It is not necessary to implement a tag handler to use custom XML. If no tag handler is registered to handle a particular element, the Content Assembler passes the XML through to the application as an `org.w3c.dom.Element`, which can then be handled by your rendering logic. A tag handler provides a mechanism to encapsulate any processing you need to do for a particular element and abstract this processing from the rendering code.

Scenarios for extending Page Builder and the Content Assembler

You can use either community editors on their own, community tag handlers on their own, or both of them in combination to extend the functionality of Page Builder.

Following are some common scenarios and their implications for community editors or tag handlers:

Scenario	Use community editor?	Use community tag handler?
Include application-specific information in the template as a pass-through XML property. *Example:* Information that the application uses to render the cartridge, but is of no interest to the content administrator.	**No** If content administrators do not need to modify the configuration of a property on a per-page basis, you do not need to write a specialized editor.	**No** The Content Assembler returns the XML to the rendering code for your application.
Include external configuration in the template as a pass-through XML property. *Example:* Hard-coded configuration for a third-party system that applies to any page that uses this template.	**No** If content administrators do not need to modify the content of a property on a per-page basis, you do not need to write a specialized editor.	**Yes** The Content Assembler uses the information contained in the XML to query a third-party system, and returns the results to the rendering code.
Provide a new interface for content administrators to configure existing Page Builder properties. *Example:* A variation of the record selector dialog box that enables content administrators to browse for featured records, instead of entering a record ID.	**Yes** This editor is bound to a standard property. (In the example, the editor modifies a `<RecordList>` property.)	**No** The community editor outputs standard Endeca content XML, which is processed by the standard tag handler for record lists. No additional work is necessary.

Scenario	Use community editor?	Use community tag handler?
Provide an interface to configure functionality that is not supported by Page Builder out-of-the-box. *Example:* An editor that enables content administrators to specify reviews to display for a particular navigation state, including number of reviews, sort order, and additional filtering options.	**Yes** The editor provides a specialized interface for selecting data to populate a cartridge. The configuration is saved as a custom XML property.	There are two options: **No** The Content Assembler returns the XML to the application's rendering code, which can then fetch the reviews from the CMS where they are stored. **Yes** (preferred) The Content Assembler fetches the reviews from the CMS before returning the content results to the rendering code for your application. Similarly, you can use a tag handler and community editor to send customized queries to an MDEX Engine and return results to the rendering code.

Life cycle of a Content Assembler query

This section describes the sequence of events that occur when the Content Assembler processes a query.

Recall that the application sends a query through the Content Assembler by specifying an `ENEQuery`, a host and port for an MDEX Engine, and a zone from which to retrieve the content results.

```
ENEContentQuery query = (ENEContentQuery)contentManager.createQuery();

query.setENEQuery(new UrlENEQuery(request.getQueryString(), encoding));
query.setENEConnection(new HttpENEConnection(eneHost, enePort));
query.setRuleZone("NavigationPageZone");

ENEContentResults results = query.execute();
```

The following sequence of events occurs when the query is executed:

1. The Content Assembler sends the query to the MDEX Engine and retrieves the `ContentResource` from the query results.

   This is the content XML (stored in the properties of the first rule returned in the specified zone) that represents the landing page configuration created in Page Builder.

2. The Content Assembler calls `ContentAssembler.assemble()`.

   This method marshals the `ContentResource` into an `org.w3c.dom.Element`, then calls `ContentAssembler.evaluate()`, passing in a `ContentContext` that contains the relevant resources for the `ContentQuery` and the `Element` representing the root `ContentItem`.

3. `ContentAssembler.evaluate()` calls the `evaluate()` method of the appropriate tag handler (in the case of the root element, this is `com.endeca.content.assembler.tags.ContentItem¬Tag`). This method takes two arguments: the current `ContentContext`, and the `Element` to be evaluated.
4. The tag handler marshals the `Element` into a Java object.

   As part of the `evaluate()` method, the tag handler may execute additional queries against an MDEX Engine or a third-party system. The Content Assembler also provides a mechanism for a tag handler to invoke additional tag handlers.

   For example, `ContentItemTag` invokes `PropertyTag`, which in turn invokes tag handlers for specific property types to populate the property values.

When the Content Assembler has finished processing the root `ContentItem` element including all its children, it has transformed the content XML tree into a tree of `ContentItem` objects with properties and nested `ContentItem` objects. This object tree is then returned to the application for rendering.

Class overview

The `com.endeca.content.assembler` package contains the classes and interfaces that make up the core Content Assembler implementation and enable extension of Content Assembler functionality through tag handlers.

Class	Description
`ContentResource`	The byte representation of the content XML returned in the MDEX query results. The `ContentAssembler` marshals this into an XML `Element` object for processing by tag handlers.
`ContentResourceLocator`	Used internally to fetch a `ContentResource` object. For each `ContentManager`, there is a single `ContentResourceLocator`; therefore, there should be a single `ContentResourceLocator` across the entire application.
`TagHandler`	Transforms a specific element in the content XML into an object. The Content Assembler ships with several standard tag handlers. You can implement your own tag handlers to process custom XML elements.
`ContentAssembler`	Used to transform a `ContentResource` into an object model representation of its content item. For each `ContentManager`, there is a single `ContentAssembler`; therefore, there should be

Class	Description
	a single `ContentAssembler` across the entire application.
`ContentContext`	Provides access to resources that are shared across tag handlers.

Related Links

Resources managed by the ContentContext object on page 33
The `ContentContext` object provides access to resources that are shared across tag handlers.

Implementing the tag handler interface

A tag handler takes an element in the content XML and transforms it into a Java object. In the typical use case, you write a tag handler to return the value of a particular property.

Your tag handler can do as much or as little processing during the course of marshaling XML into objects, including executing one or more queries to an MDEX Engine or another third-party system.

Important:

All tag handlers are instantiated when the application's `ContentManager` is created, and then reused for each element that the Content Assembler processes. Because multiple invocations of a tag handler may be executed concurrently, tag handlers must be reentrant.

For performance reasons, tag handlers should not contain large blocks of synchronized code.

To implement the tag handler interface:

1. Include the following import statements in your code:

```
import com.endeca.content.assembler.*;
import com.endeca.content.ContentException;
import org.w3c.dom.Element;
```

2. Implement the `evaluate()` method.

   This method takes an XML element to process and a `ContentContext`, and returns an `Object` to the tag handler that invoked it (typically, a `PropertyTag` handler).

Note: Java 1.5 provides improved facilities for working with XML. If you are using Java 1.4, you may want to use a third-party library that provides similar XML functionality or write your own implementation of some of these utility classes.

Resources managed by the ContentContext object

The `ContentContext` object provides access to resources that are shared across tag handlers.

A new `ContentContext` is instantiated for each `ContentQuery`. The `ContentContext` class provides the following methods:

Method	Description
`getContentAssembler()`	Returns a reference to the active `ContentAssembler`. You can use the `ContentAssembler` to invoke additional tag handlers by calling its `evaluate()` method.
`getLocalPropertyName()`	Returns the name of the `Property` on which processing has most recently begun. This property is set by the `PropertyTag` handler when it begins to process a <`Property`> element.
`getContentItemStack()`	Returns a reference to the stack of `ContentItem` objects currently being assembled. When the `ContentItemTag` handler begins to process a <`ContentItem`> element, it adds the `ContentItem` to this stack. The tag handler pops the `ContentItem` from the stack when it is done. **Note:** Other tag handlers should not modify the content item stack.
`getContentResourceLocator()`	Returns a reference to the active `ContentResourceLocator`. Typically, you do not need to use the `ContentResourceLocator` unless you want to retrieve a `ContentResource` from a zone other than the one that you passed to the `ContentQuery`. You can pass a `ContentResource` to the `ContentAssembler.assemble()` method to transform the content XML into a `ContentItem` object (which may contain other `ContentItem` objects).
`getContentResources()` or `getContentResourceStack()`	Provides access to the `ContentResource` objects being assembled. Typically, there is only one `ContentResource` for any given `ContentQuery`, and you do not need to access it directly. Rather, tag handlers work on the XML that is passed through the `evaluate()` method.

Related Links

Class overview on page 32

The `com.endeca.content.assembler` package contains the classes and interfaces that make up the core Content Assembler implementation and enable extension of Content Assembler functionality through tag handlers.

About invoking other tag handlers

You can write tag handlers that invoke other tag handlers (either standard tag handlers or other community tag handlers).

You invoke another tag handler by calling `ContentAssembler.evaluate()` and passing in the element to be processed, along with a reference to the current `ContentContext`.

```
  final ContentAssembler contentAssembler = pContentContext
    .getContentAssembler();
  return contentAssembler.evaluate(pContentContext, childElement);
```

The `ContentAssembler.evaluate()` method identifies the appropriate tag handler for the element, if one exists, and calls its `evaluate()` method. It is important that you pass a child element, rather than the current element being processed by your tag handler, to the `ContentAssembler.evaluate()` method, otherwise the `ContentAssembler` would invoke the same tag handler with the same element in an infinite loop.

It is not necessary to invoke other tag handlers from within your own tag handler, even if you have nested elements within your custom XML. There are two cases in which this may be especially useful.

Multiple combinations of valid child elements

You may have optional elements or different possible combinations of elements within your custom XML. In such a case, rather than adding logic to check for each element that your tag handler may have to process, you can write separate tag handlers for each possible child element. The parent tag handler can simply iterate through the child elements and call `ContentAssembler.evaluate()` on each child.

For example, the standard `RecordList` element can contain either a `RecordQuery` (for featured records) or a `NavQuery` (for dynamic records). The `RecordListTag` handler invokes either the `RecordQueryTag` handler or the `NavQueryTag` handler to perform a query against the MDEX Engine that returns the records for a Content Spotlighting cartridge.

Same element nested under more than one parent

Your use of custom XML within your application may produce a structure similar the following:

```
<Property>
  <TagA>
    <TagC/>
  </TagA>
</Property>
<Property>
  <TagB>
    <TagC/>
  </TagB>
</Property>
```

In this case, you can write separate tag handlers for `<TagA>` and `<TagB>` that each invoke a third handler for `<TagC>`. This ensures consistent handling of `<TagC>` regardless of its parent element.

Integrating a tag handler into the Content Assembler

The Content Assembler provides a simple interface for registering tag handlers.

This procedure assumes that you have already written a tag handler class that implements `com.endeca.content.assembler.TagHandler`.

To integrate a tag handler with the Content Assembler for Java:

1. Associate the tag handler class with an element in the content XML by writing a handler map.

2. Package the tag handler and its handler map in a Java archive (JAR) or a Web archive (WAR).

   The handler map must be located in `META-INF/tmgr/handler-map.xml` within the tag handler package.

3. Install the tag handler.

   - For a Java archive: Add the tag handler JAR to your application server's classpath.
   - For a Web archive: Add the tag handler WAR to the WEB-INF/classes directory of your application.

4. Restart the application server.

In order for the Content Assembler to make use of the new tag handler, the content XML must contain the element that the tag handler is intended to process. You can achieve this in one of the following ways:

- Specify pass-through XML in a page template or cartridge template.
- Specify a custom property type in a template and bind it to an editor that generates custom XML.

About working with handler maps

You associate a tag handler with the element that it is intended to process by creating a file named `handler-map.xml`. Each handler map contains a registry of tag handlers identified by a fully qualified XML element name.

At initialization time, the Content Assembler loads and merges all classpath resources named `META-INF/tmgr/handler-map.xml`. A handler map may contain one or more `<tag>` elements in the following format:

```
<handler-map>
  <tag name="Integer"
       namespace="http://endeca.com/sample-schema/2010"
       class="com.endeca.content.assembler.tags.sample.IntegerTag"/>
</handler-map>
```

As the Content Assembler processes the content XML that represents a landing page, it invokes the appropriate tag handler for each element. In the sample content XML excerpt below, the Content Assembler would call the `evaluate()` method of the `com.endeca.content.assembler.tags.sam¬ple.IntegerTag` handler to process the `<Integer>` element. In this case, it returns a `Property` object with an `Integer` value of 17.

```
<ContentItem type="PageTemplate">
  <TemplateId>IntegerTagSample</TemplateId>
  <Name>Integer demo</Name>
  <Property name="humbug">
    <Integer xmlns="http://endeca.com/sample-schema/2010">17</Integer>
  </Property>
</ContentItem>
```

Note: Tag handlers are loaded by the Content Assembler in classpath order. To avoid conflicts between tag handlers, ensure that each element specified in a handler map has a unique QName across your entire application.

Standard tag handlers in the Content Assembler

The Content Assembler API package includes a handler map that defines the tag handlers associated with the standard property types.

The handler map defines the following handlers for the Page Builder content types:

XML element	Tag handler implementation
`http://endeca.com/schema/content/2008:Boolean`	`com.endeca.content.assembler.tags.BooleanTag`
`http://endeca.com/schema/content/2008:ContentItem`	`com.endeca.content.assembler.tags.ContentItemTag`
`http://endeca.com/schema/content/2008:ContentItemList`	`com.endeca.content.assembler.tags.ContentItemListTag`
`http://endeca.com/schema/content-tags/2008:DimensionList`	`com.endeca.content.ene.tags.DimensionListTag`
`http://endeca.com/schema/content-tags/2008:NavigationRecords`	`com.endeca.content.ene.tags.NavigationRecordsTag`
`http://endeca.com/schema/content-tags/2008:NavigationRefinements`	`com.endeca.content.ene.tags.NavigationRefinementsTag`
`http://endeca.com/schema/content-tags/2008:NavQuery`	`com.endeca.content.ene.tags.NavQueryTag`
`http://endeca.com/schema/content/2008:Property`	`com.endeca.content.assembler.tags.PropertyTag`
`http://endeca.com/schema/content/2008:RecordList`	`com.endeca.content.assembler.tags.RecordListTag`
`http://endeca.com/schema/content-tags/2008:RecordQuery`	`com.endeca.content.ene.tags.RecordQueryTag`
`http://endeca.com/schema/content/2008:String`	`com.endeca.content.assembler.tags.StringTag`
`http://endeca.com/schema/content-tags/2008:Supplement`	`com.endeca.content.ene.tags.SupplementTag`

About the sample tag handler

The Content Assembler API package includes a sample tag handler implementation.

The sample is located in `ContentAssemblerAPIs/Java/`*`version`*`/reference/tag_handlers` and includes the following:

File	Description
`endeca-content-`*`version`*`-samples.jar`	A sample tag handler that transforms the contents of an `<Integer>` element into an `Integer` object.

File	Description
`PageTemplate-IntegerTagSample.xml`	A sample page template that contains an `<Integer>` property.

The sample package is intended to demonstrate the integration points between tag handlers and the Content Assembler. It does not include any rendering code for the reference application to make use of the `Integer` property returned by the Content Assembler.

Note: The sample tag handler does not support Java 1.4.

Accessing the source and configuration files

You can unpack the sample tag handler JAR to access the source code and associated configuration files. The JAR package includes:

File	Description
`com/endeca/content/assembler/tags/sample/IntegerTag.class`	The compiled `IntegerTag` handler.
`META-INF/tmgr/handler-map.xml`	The handler map associating the `IntegerTag` handler with the `<Integer>` element.
`META-INF/tmgr/schema-map.xml`	The schema map enabling Content Assembler validation of the `<Integer>` element.
`META-INF/schema/sample.xsd`	The schema for the `<Integer>` element within the `http://endeca.com/sample-schema/2010` namespace.
`src/com/endeca/content/assembler/tags/sample/IntegerTag.java`	The source code for the `IntegerTag` handler.

Installing the sample tag handler

The sample tag handler is provided as a Java archive along with a simple page template that defines an `<Integer>` property with a default value.

To install the sample tag handler:

1. Add the sample tag handler JAR to your application server's classpath.

   The sample JAR includes the requisite handler map for registering the tag handler with the Content Assembler. No additional configuration is necessary.
2. Restart your application server.
3. Copy the sample template to your local templates directory and upload it using the emgr_update utility. For example:

   ```
   emgr_update --action set_templates --host localhost:8006
   --app_name My_application --dir /apps/endeca/templates/
   ```

   The template does not define any editors associated with the integer property. The `<Integer>` element is treated as pass-through XML.

About extending the Content Assembler to validate custom XML

You can configure the Content Assembler to validate content XML, including custom XML.

Recall that you can enable XML validation in the `ContentManager` as follows:

```
  // create the global content manager
  ENEContentManager contentManager = new ENEContentManager();
  // enable runtime validation of XML
  contentManager.setValidating(true);
```

If validation is enabled, the Content Assembler performs schema validation as it processes the content XML. By default, the Content Assembler validates any elements within the Endeca content XML namespaces (`http://endeca.com/schema/content/2008` and `http://ende¬ca.com/schema/content-tags/2008`) that are defined in the associated schemas.

You can specify additional schemas that the Content Assembler uses to validate content XML by creating a schema map. As with the handler map, the Content Assembler loads and merges all classpath resources named `META-INF/tmgr/schema-map.xml` at initialization time. A schema map may contain one or more `<entry>` elements in the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="http://endeca.com/sample-schema/2010">/META-INF/schema/sam¬
ple.xsd</entry>
</properties>
```

At runtime, the Content Assembler matches the namespace of each element in the content XML against the namespaces defined in any `META-INF/tmgr/schema-map.xml` resource. If the associated schema file defines the element being processed, the Content Assembler validates that element against the schema.

Note: Validation can be useful in a testing environment for debugging purposes, particularly if you are working with a community editor that generates custom XML. Because of the performance impact of validating content XML, this option should never be used in production. XML validation requires Java 1.5 or later, and is disabled by default.

Index