Endeca® Discovery Framework

Extension Guide

Contents

Copyright and disclaimer

Endeca Discovery Framework Extension Guide • July 2010

Preface

The Endeca® Information Access Platform is the foundation for building applications that help people understand complex information, fostering discovery and improving daily decision-making. These applications instantly summarize data and content for users—even for unanticipated requests. The Guided Summarization™ experience leads to unexpected insights in millions of everyday decisions, increasing revenue, decreasing costs, and accelerating operations.

The Endeca Information Access Platform is powered by MDEX Engine™ technology, a new class of database designed for exploring information, not managing transactions. The MDEX Engine is supported by:

- An adaptive application component library that enables the rapid development of information access applications that automatically adapt to changes in the data and content.
- A Web-based management suite that empowers managers to highlight the right information at the right time to end users through adaptive presentation rules and dynamic pages.

These essential capabilities are delivered as an enterprise-class platform, with the scalability, reliability, and security that leading organizations demand.

About this guide

This guide contains information about extending the Endeca Discovery Framework on Windows and Linux. It also introduces the Discovery Framework Component SDK.

The Endeca Discovery Framework enables rapid configuration of search applications—often in just minutes—that offer the highly interactive Guided Navigation® user experience across a full range of structured and unstructured enterprise data. It includes a Component SDK, which is a packaged development environment for portlets, themes, layout templates, and other portal element. Endeca has modified Liferay's version of its Plugins SDK to include the Endeca enhancements, such as the `EndecaPortlet` core class.

Who should use this guide

This guide is intended for developers who are building applications using the Endeca Discovery Framework on Windows or Linux.

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

Extending the Discovery Framework

Out of the box, the Endeca Discovery Framework includes numerous components that you can use to quickly develop an enterprise-quality search application. In addition, the Discovery Framework provides a number of extension points for managing query and portlet operations, along with default implementations of the various interfaces that you can modify.

Developer tasks in the Discovery Framework

Data source configuration tasks include:

- Adding data sources.
- Adjusting security.
- Customizing how data sources interact with each other.

Component customization tasks include:

- Adding or modifying portlet components based on the `EndecaPortlet` class, using the Discovery Framework Component SDK.
- Localizing components.

This guide covers all of these developer tasks.

Licensing requirement for component development

Discovery Framework component development may require the purchase of a third party license.

If you want to modify any Discovery Framework components that use the Ext JS API, or use that API to create your own components, you must purchase a developer's license from *Ext JS*.

Obtaining more information

Because the Discovery Framework is built upon the Liferay Portal, you can access Liferay's documentation for more information about how to perform administrative and developer tasks.

Specifically, the *Liferay Portal Administrator's Guide* provides extensive information about installing, configuring, and maintaining a portal. To access a free PDF download of this guide, go to *http://www.liferay.com* and navigate to Documentation.

Liferay developer resources

The *Discovery Framework Extension Guide* (this guide) only covers Endeca extensions to the Liferay Portal. For additional developer support, Liferay provides blogs, wikis, and forums. To access this, go to *http://www.liferay.com* and navigate to Community.

The Endeca Developer Network (EDeN)

You can obtain more information about the Discovery Framework and other Endeca products at the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Additional Endeca documentation

The Discovery Framework `doc` directory also contains javadoc, as well as *Discovery Framework Installation Guide* and *Discovery Framework Component Catalog*.

Chapter 2

About data source configuration

This section discusses how to create and configure data sources, and outlines the default interaction model between related data sources. Data sources and query states are how the Discovery Framework represents connections to MDEX Engines and queries to those MDEX Engines.

About data source files

Data source files are written in the JSON language.

Every data source must include `server` and `port` keys. All other settings are optional. If `name` and `id` are not included, they are both given the name of the file defining the data source by default. For example, a file named `parts.json` results in a data source with the `name` and `id` "parts." The `name` setting is normally the only user-visible identification of a data source. The `id` setting is only used internally, and the `description` setting is only used for logging and debugging.

Note: The `id` cannot contain spaces.

Note: SSL configuration file paths are relative to the directory containing the JSON data source files. This is typically, but not always, the `endeca-portal\data\endeca-data-sources` directory.

Data source example

The following example shows a JSON data source file based on wine merchant data.

```
{
 "server":"server01.lab.acme.com",
 "port":"15000",
 "sslConfig": {
  "caFile":"truststore.ks",
  "caPassword":"endeca",
  "certFile":"keystore.ks",
  "certPassword":"endeca"
 }
 "id":"ds-id",
 "name":"Descriptive DataSource name",
 "description":"Detailed information about this DataSource",
 "baseFunctions": [
  {
   "class":"com.endeca.portal.data.functions.RecordFilter"
```

```
   "recordFilter":"Regions:Midwest"
  },
  {
   "class":"com.endeca.portal.data.functions.RangeFilter"
   "property":"P_Price",
   "rangeOperator":"GTEQ",
   "value1": "25"
  },
  {
   "class":"com.endeca.portal.data.functions.RefinementFilter"
   "dimValId":"123",
   "dimensionId":"121"
  }
       ],
 "securityEnabled": "true",
 "inheritSecurity": "true",
 "securityFilters": {
  "frenchFilter": {
   "class":"com.endeca.portal.data.functions.RecordFilter",
   "recordFilter":"OR(Region:Bordeaux,Region:Burgundy)"
  },
  "expensiveFilter": {
   "class":"com.endeca.portal.data.functions.RangeFilter",
   "property":"P_Price",
   "rangeOperator":"GTEQ",
   "value1": "25"
  }
 },
 "rolePermissions": {
  "French Wine": ["frenchFilter"],
  "French Premium": ["frenchFilter","expensiveFilter"]
 }
}
```

The default data source

Among the data sources in your Discovery Framework application, you must always include a default data source. This data source is automatically assigned to all data-source-backed components when they are initially added to a page.

When you initially start the Discovery Framework, the `id` of this data source is "default." While this setting remains unchanged, you must include a data source with the `id` “default."

Alternatively, you can change the default data source name. To do so:

1. In the **Liferay Control Panel**, go to the **Discovery Framework Settings** page.
2. In the **Discovery Framework Settings** page, change the `df.defaultDataSource` setting to a different value.
3. Restart the **Discovery Framework**.
4. Make sure you include a data source with that new value as its `id`.

Note: For more information about **Discovery Framework Settings**, see the *Discovery Framework Installation Guide*.

Configuring the Discovery Framework to connect to a secured MDEX Engine

This topic provides a high-level description of how you can set up the Discovery Framework to connect to a secured (HTTPS) MDEX Engine.

Several of the steps below refer to chapter 3 of the *Endeca Platform Services Security Guide*, entitled "Using Endeca SSL Certificate Utilities." The *Endeca Platform Services Security Guide* is available as part of the Platform Services documentation set on EDeN. Before attempting these steps, make sure you have a copy of that guide at hand.

Note: The steps below assume you are using the Discovery Framework Tomcat bundle.

1. Generate the SSL certificate files for the Dgraph using the `enecerts` utility. This utility is available in both the Platform Services and MDEX Engine installation. For instructions, refer to the section "Generating SSL certificates" in the *Endeca Platform Services Security Guide*.
2. Generate the Java KeyStore (JKS) files using the `endeca-key-importer.jar` utility from the Endeca Platform Services installation. For instructions, refer to the topic "Converting PEM-format keys to JKS format" in chapter 3 of the *Endeca Platform Services Security Guide*.
3. Place the JKS keys into the directory containing the JSON data source files. This is typically, but not always, the `endeca-portal\data\endeca-data-sources` directory.
4. Specify the `caFile`, `certFile`, `caPassword`, and `certPassword` in the appropriate JSON data source file.
   The following example is extracted from the data source file example in the previous topic.

```
"sslConfig": {
  "caFile":"truststore.ks",
  "caPassword":"endeca",
  "certFile":"keystore.ks",
  "certPassword":"endeca"
 }
```

5. Restart the Discovery Framework.

Provided QueryFunction classes in the Discovery Framework

This topic defines the `QueryFunction` classes included in the Discovery Framework.

Filters

Function Class	ENEQuery Equivalent	Configuration Properties	Notes
`EQLFilter`	`setNavRecord¬ StructureExpr()`	`eqlFilter:` String	Applying an EQL filter to a set of QueryState always overwrites previous filters.

Function Class	ENEQuery Equivalent	Configuration Properties	Notes
RangeFilter	setNavRangeFilters()	property: String rangeOperator: (LT|LTEQ|GT|GTEQ¬ |BTWN|GCLT|GCGT|GCBTWN) value1: numeric value2: numeric (optional) value3: numeric (optional)	
RecordFilter	setNavRecordFilter()	recordFilter: String	
RefinementFilter	setNavDescriptors()	dimValId: long dimensionId: long multiSelect: (AND|OR|NONE) (optional) navigable: (true|false) (optional)	
SearchFilter	setNavERecSearches()	searchInterface: String terms: String matchMode:	

Configuration functions

Function Class	ENEQuery Equivalent	Configuration Properties	Notes
AnalyticsQueryConfig	setAnalyticsQuery()	analyticsQuery: String	
ExposeRefinement	setNavExposed¬ Refinements()	dimValId: String dimensionId: String ownerId: String (optional)	At least one of dimValId or dimensionId is required. ownerId can be the ID of a NavConfig instance.
NavConfig	setNavAllRefinements	exposeAllRefinements: boolean	
ResultsConfig	setNavNumERecs() setNavERecsOffset()	recordsPerPage: int offset: int (optional)	

Function Class	ENEQuery Equivalent	Configuration Properties	Notes
	`setSelection`	`columns:` String[] (optional)	
`SortConfig`	`setNavActiveSortKeys`	`property:` String `ascending:` boolean	

Data source role-based security

The default `MDEXSecurityManager` implementation supports the configuration of filters associated with Liferay roles that have been assigned to a user's login.

The supported filters are as follows:

- **securityEnabled:** enabling/disabling security filters on all queries issued to this data source.
- **securityFilters:** definition of all security filters to be used by this data source (any extension of `QueryFilter`).
- **rolePermissions:** role mappings to the security filters that have been defined for the data source.
- **inheritSecurity:** enabling/disabling of security filter inheritance, based on data source relationships defined via the `parentDataSource` property.
- **parentDataSource:** when `inheritSecurity` is true, this property is used to find all ancestors of each data source being secured. For each data source, the list of security filters to be applied is a combination of all security filters configured for every ancestor data source, plus any found with the data source configuration itself.

Note: For more information on security, see Chapter 3, "Security extensions to the Discovery Framework."

About data source relationships

This topic describes the relationship between parent and child data sources.

Important: The way data sources interact with each other can vary based on the Data Source State Manager your portal is using. The information below only applies to the default implementation of the State Manager provided with the released version of the Discovery Framework.

Discovery Framework data sources can have parents and children. These relationships can be set in a data source's JSON file through the `parentDataSource:parent-ds-id` property.

In behavior, child data sources act like immutable filter sets. If you are using a child data source, the `baseFunctions` setting in the data source file should be specified with some set of filters. When a component connected to a child data source attempts to get the current query state, the query state is the query state of the data source's parent with all of the static filters on the child (as specified in configuration as `baseFunctions`) appended to it.

When a component connected to a child data source attempts to change that data source's query state (by setting a refinement from Guided Navigation, for example), the change actually happens on

the parent data source. Such an operation keeps applying to a data source's parent until it finds a data source without a parent. When any data source's query state is changed, the query states of all of its children are effectively changed as well, and any components connected to them will refresh.

Note: Role-based security filters can be inherited by descendant data sources when using the Security Manager, via the `parentDataSource` property.

Obtaining data source results

The `ENEQueryResults` class from the Presentation API is used to represent results of queries.

Portlets are encouraged to always add the relevant `QueryConfig` to specify what types of results they need. Calls to `DataSource.execute()`, without any arguments, will continue to work on ENE Presentation API data sources, but are deprecated.:

```
QueryState query = getDataSource(request).getQueryState();
query.addFunction(new NavConfig());
QueryResults results = getDataSource(request).execute(query);
```

You can then get the underlying API results and do whatever manipulation is required by your portlet.

```
ENEQueryResults eneResults = results.getENEQueryResults();
```

You can also make other local modifications to your query state before executing by adding filters or configurations to your query:

```
QueryState query = getDataSource(request).getQueryState();
query.addFunction(new ResultsConfig());
query.addFunction(new RecordFilter("Region:Midwest"));
QueryResults results = getDataSource(request).execute(query);
```

If you need to make modifications to your query that can't be represented on a `QueryState`, you can use `ENEQuery` instances directly:

```
DataSource ds = getDataSource(request);
ENEQuery eneQuery = ds.createENEQuery();
//modify query...
ENEQueryResults eneResults = ds.execute(eneQuery);
```

When you need to update a data source's state so that all associated portlets are updated, you cannot use `ENEQuery` instances—you must use `QueryState` instances.

```
DataSource ds = getDataSource(request);
QueryState query = ds.getQueryState();
query.addOperation(new RecordFilter("Region:Midwest"));
ds.setQueryState(query);
```

Chapter 3

Security extensions to the Discovery Framework

You may require more than the default data source role-based security discussed in the previous chapter. If so, you can customize the automated filtering of data from the MDEX Engine (based on user profile details such as the user's role or group association) by creating a custom `MDEX Security Manager`.

Security Manager class summary

This topic summarizes the `Security Manager` class.

An `MDEX Security Manager` is any concrete class that implements the `com.endeca.portal.data.security.MDEXSecurityManager`.

Abstract base class	`com.endeca.portal.data.security.MDEXSecurityManager`
Default implementation class	`com.endeca.portal.data.DefaultMDEXSecurityManager`
Description	Handles pre-execution query modification based on the user, role, or group-based security configuration of filters.
Default implementation behavior	The default `Security Manager` implementation makes use of the `securityEnabled`, `securityFilters`, `rolePermissions`, `inheritSecurity`, and `parentDataSource` properties. These properties are defined in data source configurations in order to apply role-based security filters to every query issued to the MDEX Engine backing a given data source. Users are assigned to Liferay roles in the Control Panel, and the related associations are made available to every portlet through the user's session. The `Security Manager` is responsible for maintaining an internal map of security filters for each data source that should always be applied to queries issued for that user's session.

Creating a new MDEX Security Manager

This topic describes the steps required to create an `MDEX Security Manager`.

To create a new `MDEX Security Manager` project:

1. In a terminal, change your directory to `endeca-extensions` within the Component SDK's root directory (normally called `components`).
2. Run one of the following commands:
   - On Windows: `./create-mdexsecuritymanager.bat <your-security-manager-name>`
   - On Linux: `./create-mdexsecuritymanager.sh <your-security-manager-name>`

   This command creates a `your-security-manager-name` directory under `endeca-extensions`. This directory is an Eclipse project that can be imported directly into Eclipse if you use that as your IDE.

Note: This directory also contains a sample implementation, which is essentially identical to the default implementation of the `Security Manager` used by the Discovery Framework. You can use this sample implementation to help you understand how the `Security Manager` can be used.

Implementing a new MDEX Security Manager

Your `Security Manager` must implement the `applySecurity` method described in this topic.

There are two versions of the `applySecurity` method, one of which your `Security Manager` must implement:

```
public void applySecurity(PortletRequest request, MDEXState mdexState, Query
 query) throws MDEXSecurityException;
```

The Query class in this signature is `com.endeca.portal.data.Query`. This class provides a simple wrapper around an `ENEQuery`.

Using the MDEX Security Manager

In order to use your `MDEX Security Manager`, you must specify a new class for the Discovery Framework to pick up and use in place of the default `Security Manager` implementation.

The `your-security-manager-name` directory you created contains an ant build file. The `ant deploy` task places a .jar file containing your `State Manager` into the `portal/tomcat-<version>/common/lib/ext` directory.

To specify your new class to the Discovery Framework:

1. Point the cursor at the **Dock** in the upper-right corner of the page.
2. In the drop-down menu, choose **Control Panel**.
3. In the **Portal** section of the **Control Panel** navigation panel, select **Discovery Framework Settings**.
4. Change the `df.mdexSecurityManager` property to the full name of your class, similar to following example:

   ```
   df.mdexSecurityManager = com.endeca.portal.extensions.YourSecurityManager¬
   Class
   ```

5. Click **Update Settings**.

6. Restart the Discovery Framework so the change can take effect. You may also need to clear any cached user sessions.

Chapter 4

Managing data source state in the Discovery Framework

The Discovery Framework provides an extension point that allows you to define your own interaction model by creating a custom `MDEX State Manager. In addition,` Chapter 2, "About data source configuration," describes the default interaction model between related data sources.

State Manager class summary

This topic summarizes the `State Manager` class.

An `MDEX State Manager` is any concrete class that extends from `com.endeca.portal.data.AbstractMDEXStateManager`. This class serves as a data source state manager that can be used to customize how data sources interact with each other during updates and query construction.

Abstract base class	`com.endeca.portal.data.AbstractMDEXStateManager`
Default implementation class	`com.endeca.portal.data.DefaultMDEXStateManager`
Description	Handles data source state updates and pre-execution query modification, based on data source relationships and configuration.
Default implementation behavior	The default state manager implementation makes use of the `ParentDataSource` property defined in data source configurations in order to propagate state changes throughout a hierarchy of data source relationships. When a portlet modifies the query state of its data source, that modification is applied to its parent data source and is also applied to all children of that parent. It is recursive in that it will apply all the way up and back down an ancestor tree. This allows application developers to create more advanced interfaces, such as tabbed result sets where a single Guided Navigation component should control the query state for Results Table components in individual tabs, by establishing a relationship hierarchy in data source configurations.

Creating a new MDEX State Manager

This topic describes the steps required to create an `MDEX State Manager`.

To create a new `MDEX State Manager` project:

1. In a terminal, change your directory to `endeca-extensions` within the Component SDK's root directory (normally called `components`).
2. Run one of the following commands:
   - On Windows: `./create-mdexstatemanager.bat <your-state-manager-name>`
   - On Linux: `./create-mdexstatemanager.sh <your-state-manager-name>`

   This command creates a `your-state-manager-name` directory under `endeca-extensions`. This directory is an Eclipse project that can be imported directly into Eclipse if you use that as your IDE.

Note: This directory also contains a sample implementation, which is essentially identical to the default implementation of the `State Manager` used by the Discovery Framework. You can use this sample implementation to help you understand how the `State Manager` can be used.

Implementing an MDEX State Manager

Your `State Manager` must implement the two methods described in this topic.

```
public void handleStateUpdate(PortletRequest request, MDEXState mdexState,
 QueryState newQueryState) throws QueryStateException;

public QueryState handleStateMerge(PortletRequest request, MDEXState
mdexState) throws QueryStateException;
```

- `handleStateUpdate()` is called when a portlet calls `DataSource.setQueryState(qs)`. This method should eventually call `mdexState.setQueryState()`. (However, if it determines that, for whatever reason, the `MDEXState's QueryState` should not change, it is not required to make this call.) `handleStateUpdate()` is also responsible for marking any data sources impacted by the update (which could depend upon your implementation of `handleStateMerge()`) so that portlets that listen to them on the page will properly update. For this reason, the `addEventTrigger(PortletRequest request, MDEXState ds)` method is provided for you to call, with the passed in request object and any `MDEXState` objects that are considered changed.
- `handleStateMerge()` is called when a portlet calls `DataSource.getQueryState()`. You are expected to return the `QueryState` that the portlet should get access to for the data source represented by the `mdexState`, taking into account any data source relationships or other aspects of your `State Manager` that might impact query state.

Using the MDEX State Manager

In order to use your `MDEX State Manager`, you must specify a new class for the Discovery Framework to pick up and use in place of the default `State Manager` implementation.

The `your-state-manager-name` directory you created contains an ant build file. The `ant deploy` task places a `.jar` file containing your `State Manager` into the `portal/tomcat-<version>/common/lib/ext` directory.

To specify your new class to the Discovery Framework:

1. Point the cursor at the **Dock** in the upper-right corner of the page.
2. In the drop-down menu, choose **Control Panel**.
3. In the **Portal** section of the **Control Panel** navigation panel, select **Discovery Framework Settings**.
4. Change the `df.mdexStateManager` property to the full name of your class, similar to following example:

   ```
   df.mdexStateManager = com.endeca.portal.extensions.YourStateManagerClass
   ```

5. Click **Update Settings**.
6. Restart the Discovery Framework so the change can take effect. You may also need to clear any cached user sessions.

Chapter 5

Installing and using the Component SDK

You can customize the Discovery Framework even further by creating your own components. The Discovery Framework Component SDK is a packaged development environment that you can use to add or modify portlets, themes, and layout templates. The Component SDK is a modified version of the Liferay Plugins SDK. The Endeca version includes enhancements such as the `EndecaPortlet` core class.

Downloading and configuring the Component SDK

You can download the Discovery Framework Component SDK from the Downloads section of the Endeca Developer Network (EDeN).

Before installing the Component SDK, download and unzip `endeca-portal-1.2.zip`, as described in the *Discovery Framework Installation Guide*. This is the base Discovery Framework code, which the Component SDK depends upon. You do not have to start the Discovery Framework.

To install the Component SDK:

1. Download and unzip `components-sdk-1.2.zip` to a separate directory. This is the Component SDK itself. Perform the following steps within the Component SDK.
2. Create a file `components/build.<user>.properties`, where `<user>` is the user name with which you logged on to this machine.
3. Within that `properties` file, add a single property `portal.base.dir=<absolute_path_to_portal>` , where `<absolute_path_to_portal>` is the path to the unzipped `endeca-portal-1.2.zip`.
4. Create a `shared.properties` file in the `shared/` directory.
5. Edit `shared/shared.properties` and set the single property `portal.base.dir=<absolute_path_to_portal>` , where `<absolute_path_to_portal>` is the path to the unzipped `endeca-portal-1.2.zip`.

Configuring Eclipse for component development

Before developing Discovery Framework components in Eclipse using the Component SDK, two Eclipse classpath variables need to be created.

 **Note:** Depending on your version of Eclipse, the steps below may vary slightly.

To configure the Eclipse classpath variables for Endeca Discovery Framework component development:

In Eclipse, go to **Window** > **Preferences** > **Java** > **Build Path** > **Classpath Variables** and create two new variables:

Name	Path	Example
DF_GLOBAL_LIB	Path to the application server global library.	`C:/endeca-portal/tomcat-<version>/¬ common/lib`
DF_PORTAL_LIB	Path to the Liferay ROOT Web application library.	`C:/endeca-portal/tomcat-<version>/¬ webapps/ROOT/WEB-INF/lib`

Once these variables have been created, the components generated by the Component SDK are ready to be imported into Eclipse.

Component development overview

This topic provides a high-level overview of the component development process. Subsequent topics explain each step given here in greater detail.

To develop a new Discovery Framework component:

1. Create the component.
2. Import the project in Eclipse.
3. Build and test the new component.

Creating a new component

New Discovery Framework components are extensions of the `EndecaPortlet` class.

To create a new component:

1. At a command prompt, navigate to the Component SDK directory, and from there to `components/portlets`.
2. Run the command `create.bat` *`a-portlet-name-without-spaces`* `"A Friendly Portlet Name"` where:
   - The first argument must not have spaces. The string `-portlet` is automatically appended to the name.
   - The second argument is intended to be a more human-friendly name. Spaces are allowed, but if the name has spaces, it must be enclosed in quotation marks.

   An example command would be `create.bat jons-test "Jon's Test Portlet"`

Importing the project in Eclipse

Before beginning component development, you have to import the component project you just created into Eclipse.

To import the Discovery Framework Component SDK project you just created into Eclipse:

1. Within Eclipse, choose **File** > **Import** > **General** > **Existing Projects into Workspace**.
2. As the root directory from which to import, select the directory where you installed the Component SDK. You should see multiple projects to import.
3. Import the portlets you need to work with. If your portlets depend on shared library projects located within the `/shared` directory, import those as well.

Note: It takes some time for projects to build after they are imported.

Building and testing your new component

Next, you can build your new component in Eclipse and ensure that it appears in the Discovery Framework.

To build your new component in Eclipse:

1. In your new project, open the `build.xml` file at the top level.
2. In the outline view, right-click the deploy task and select **Run as...** > **Ant Build**.

Note: This step is only necessary if you do not have **Build Automatically** checked in the Eclipse **Project** menu.

3. If the Discovery Framework is not already running, log on to the Discovery Framework and sign in.
4. Look at the Discovery Framework logs to confirm that the component was picked up successfully.
5. Test your new component within the Discovery Framework by choosing **Add Application** and looking in the **Sample** category. Add the new component to your page by dragging and dropping it.

Modifying Endeca enhancements to the Component SDK

The `build.xml` file in the root directory of each component created by the Component SDK contains three lines that control Endeca's build enhancements.

By default, these three lines are:

```
<property name="shared.libs" value="endeca-common-resources,endeca-discovery-
taglib" />
 <property name="endeca-common-resources.includes" value="**/*" />
 <property name="endeca-common-resources.excludes" value="" />
```

The properties control the behavior described below:

- The `shared.libs` property controls which of the projects in the `shared/` directory are included in your component. These shared projects are compiled and included as `.jar` files where appropriate.

- The `endeca-common-resources` `include` and `exclude` properties control which files in the `shared/endeca-common-resources` project are copied into your component. By default, all `endeca-common-resources` files are included, giving your component the Endeca AJAX enhancements (`preRender.jspf` and `postRender.jspf`) and the ability to switch between data sources in your component's preferences (`dataSourceSelector.jspf`). If your component needs to override any of these files, you must exclude them via these build properties or your code will be overwritten.

  These `include` and `exclude` properties can be specified for any shared library, as shown in the following example:

  ```
  <property name="endeca-discovery-taglib.includes" value="**/*" />
   <property name="endeca-discovery-taglib.excludes" value="" />
  ```

  When unspecified, `includes` default to `"**/*"` and `excludes` default to `""`.

Chapter 6

Localizing the Discovery Framework

The Discovery Framework is an internationalized application that can be adapted for use in different locales. This section describes how to localize your Discovery Framework components.

Discovery Framework localization scenarios

Discovery Framework localization refers to two sets of tasks.

The first case is translating a component that has already been localized. In this scenario, you are applying the translation to components whose message strings have already been externalized to a resource bundle. Details on modifying and deploying a translated component appear in the next section.

The second, more involved case is developing or updating a component so that it supports localization. For details, see the section beginning with the topic "Setting up a component for localization."

About adding a translation to a released component

This section discusses translating a component that has already been localized.

In this scenario, the component's English-language message strings have been externalized into the portlet WAR file's resource bundle. These strings can be translated to the target language and then made available to the Discovery Framework.

Note: If you are working with a double-byte, extended character set language, consult the section "Working with non-Unicode characters" that appears later in this chapter before following the procedure below.

Adding a translation to a released component

This procedure can be followed whether you want to translate the content yourself or obtain the translation from a third party.

To add translated message strings to a released component:

1. Unzip the WAR file of the localized component you want to modify.
2. Edit its `portlet.xml` file to enable the additional locale you want to support. For example, to add French, include `<supported-locale>fr</supported-locale>`.

3. In `WEB-INF/classes/com/endeca/` (or other location, based on your component's class structure), generate a `Resource_[locale].properties` file for the new language. This file should contain target-language values of the properties used in the component. To see the supported properties, refer to the `WEB-INF/classes/com/endeca/Resource_en.properties` file already in the component. Your file should contain a version of each of those messages in your target language.
4. Re-zip the WAR file of the component and place it in the `endeca-portal/deploy` directory. Liferay hot-deploys the component.
5. Repeat steps 1 through 4 for each component you want to enable for your target language.
6. Start the Discovery Framework and add your components, as well as the **Language** component, to the page.
7. In the **Language** component, click the flag associated with your target language.
   The Discovery Framework displays the component messages from your resource bundle in your target language. In addition, because the portal itself is also localized, menus and other portal controls also appear in your target language.
8. In the **Language** component, click the United States flag to switch back to English.

Setting up a component for localization

This topic describes the steps needed to develop or update a component so that it supports localization.

To set up a portlet for localization:

1. Update the `portlet.xml` file to specify the locales this portlet will support.

   The following example enables English and German:

   ```
   <supported-locale>en</supported-locale>
   <supported-locale>de</supported-locale>
   ```

2. Update `portlet.xml` to specify the location of the portlet's resource bundle. (The resource bundle is the mechanism the Liferay Portal uses to add localized content to a portlet.)

   Continuing our example, we will include resource files `Resource_en.properties` and `Resource_de.properties` in the sample portlet's `com/endeca/portlet/sample/` directory:

   ```
   <resource-bundle>com.endeca.portlet.sample.Resource</resource-bundle>
   ```

3. Create resource bundles for your supported languages in `WEB-INF/src/[path/to/resource/bundle]_[locale].properties` (for example, the bundle for English for an Endeca component would be `WEB-INF/src/com/endeca/portlet/sample/Resource_en.properties`). For the most part, this is a simple `properties` file with key/value pairs for message IDs and their locale-specific messages.
4. Update your portlet's implementation to use the `LanguageUtils` class to retrieve messages from the resource bundle, rather than hard-coding message strings. This should be done for all messages displayed to the user, including form labels, portlet titles (and other metadata), warning and error messages, preferences pages, help text, and so on. See below for details on how to use the `LanguageUtils` class.

Note: See the sections below for details about portlet-specific messages and messages with tokens.

Note: You may note that the `resource-bundle` attribute is different from the file path you edit messages in. This is because the portlet build process combines common message strings from shared libraries with your portlet-specific messages to create the final `com/endeca/Resource_[locale].properties` file in the compiled portlet WAR. For more information, see the topic below on build process interaction with localization.

Build process interaction with localization

You should edit localization messages in a different resource file from the one you configure the portlet to read messages from.

The build process combines resource files into a single resource file that the component reads messages from. The build combines the component's `com/endeca/PluginResource_[locale].properties` file and any file found in a shared library's directory matching `com/endeca/*Resource_[locale].properties` into a single `com/endeca/Resource_[locale].properties` file. The messages from your component's `PluginResource_[locale].properties` appear at the top of the final `Resource_[locale].properties`, so you can easily override any messages from shared libraries. However, if your component includes more than one shared library, no guarantee can be made about the order in which the resource files from shared libraries will be appended.

Localizing your own shared libraries

If you have included localized messages in your shared libraries, make sure you choose a prefix other than Plugin for the resource file `com/endeca/[prefix]Resource_[locale].properties`. If you do not, this file will override your component's `com/endeca/PluginResource_[locale].properties` file during the build, and your final `com/endeca/Resource_[locale].properties` will be incorrect. Endeca recommends that you choose a prefix for your library's resource file that is distinct and similar to your library's name to avoid file name conflicts with components or other shared libraries.

Switching the locale of a component

The Discovery Framework includes resources that you can use to switch a component's locale.

The **Language** component, described in the next topic, can be used to change the locale of a portlet.

There are also controls available in the **Display Settings** section of Liferay's Control Panel (as well as configuration properties in the `portal.properties` file) for setting the default container locale and the available locales. For full details on using these Liferay features, see the *Liferay Portal documentation*.

Adding the Language component

To change the locale of the server, Endeca recommends using the **Language** component to select an alternate language.

The **Language** component is included in the default **Add Components** menu.

To add the **Language** component:

1. Point the cursor at the **Dock** in the upper-right corner of the page. The **Dock** is labeled "Welcome *<user name>*!"
2. In the drop-down menu, select **Add Component**.
   The **Add Component** dialog box opens.

3. In the **Add Component** dialog box, expand the **Tools** category.
   A list of the available **Tools** components appears.
4. Click **Add**, or drag the **Languages** component to your portal page.

5. Click the flag representing the language you want to use. The portal will switch to that language, replacing English with the target language.

   For example, after clicking the Spanish flag, the **Dock** drop-down menu looks like this:

Including common externalized strings

All Discovery Framework components tend to include common messages, like those associated with the data source selector and those associated with saving preferences. The default localizations for these messages are automatically included in your compiled component.

The messages below are the default values. You can change or override these by including the same keys in your `PluginResource_[locale].properties` file.

```
### Common messages

df.portlet-does-not-support-datasource-api=Portlet does not support the API
 used by this data source.

# Data source selector messages
df.select-a-datasource=Select a data source
df.update-datasource=Update data source

df.no-data-source-selected=No data source selected for this portlet. Go to
 Preferences and select a data source.
df.no-data-source-specified=Error updating data source binding. No data
source was specified in the request.
```

```
df.data-source-binding-unchanged=Data source binding was not changed from
\"{0}\".
df.data-source-binding-unsupported-api=Data source binding was not changed
 from \"{0}\". Portlet does not support the API used by the data source
\"{1}\".
df.data-source-binding-changed-successfully=Data source binding successfully
 changed to data source \"{0}\".
df.data-source-binding-error=Error updating data source binding with new
data source name \"{0}\"; please notify your system administrator.

# Save preferences messages
df.save-prefs-success=Preferences updated successfully.
df.save-prefs-error=There was an error saving your preferences.
df.save-analytics-prefs-success=Analytics preferences updated successfully.
df.save-analytics-prefs-error=There was an error saving your analytics
preferences.
```

Note: The Discovery Framework retrieves these localized messages with their English defaults. If the messages are not included in a portlet's resource bundle, the Discovery Framework uses the hard-coded English defaults without displaying an error.

Including component-specific messages

Resource bundles should include a handful of component-specific messages that allow the Discovery Framework to localize the name, description, keywords, and category of the component.

To localize the component's metadata, include the following messages:

```
javax.portlet.title=Sample Endeca Portlet
javax.portlet.short-title=Sample Endeca Portlet
javax.portlet.keywords=Sample, Endeca, Portlet
```

Additionally, if your component is displayed in the **Add Components** menu as part of a custom category (or sub-category), you may need to localize the name of the category. Take the following categories as an example:

```
<display>
  <category name="my.new.category">
    <category name="my.new.sub-category">
      <portlet id="portlet_A" />
    </category>
  </category>
</display>
```

To localize the category names, have your component's resource bundle include the following messages:

```
my.new.category=My Category
my.new.sub-category=My Sub-Category
```

If multiple components declare the same categories, they should all include these messages, since the component container uses the localized messages from the first component that specifies them.

Using tokens in message strings

Message strings can include tokens that are substituted at run-time.

For example, a search breadcrumb may need to display a spelling correction message like *"No matches found for 'bearign'; showing results for 'bearing'"*. This message would appear in a `.properties` file with tokens for the two terms, as in the following example:

```
autocorrect-msg=No matches found for \'{0}\'; showing results for \'{1}\'
```

When including this message in your portlet with the `LanguageUtils` utility, you pass in a list of parameters to substitute for these tokens. This substitution uses the class `java.text.MessageFormat`. Refer to the javadoc for that class for the options available with token substitution. Tokens may also do advanced substitution, such as date substitution formatted appropriately for the locale.

Using the Discovery Framework LanguageUtils class

The core class provided by the Discovery Framework to access localized messages is `com.endeca.portlet.util.LanguageUtils`. There are several ways to use this class.

Calling static methods from the Java
You can access `LanguageUtils` by calling static methods from your Java class.

The following example shows the static use of the `getMessage` methods to retrieve messages (with token substitution in the third line).

```
LanguageUtils.getMessage(request, "reset");
LanguageUtils.getMessage(request, "num-records");
LanguageUtils.getMessage(request, "search-for", new String[]{ "American"
});
```

A number of convenience method signatures are provided, allowing the user to specify the portlet request and message ID, and optionally to include parameters for token substitution and a default string. The default string may be useful for shared localized messages, allowing portlets to function with a default (un-localized) message if the localized message is not retrieved from the resource bundle.

All method signatures require specifying the `PortletRequest`.

Using the Discovery taglib in JSP
The Discovery `taglib` provides a tag for retrieving localized messages. This is the recommended way to retrieve localized messages in JSPs.

The following is an example using the `taglib`:

```
<%@ taglib uri='http://endeca.com/discovery' prefix="edisc"%>
<edisc:getMessage messageName="no-matching-values"/>

<edisc:getMessage messageName="message-with-params">
    <edisc:param value="test" />
</edisc:getMessage>
```

Using the LanguageUtils class from JSP
You can access `LanguageUtils` to retrieve localized messages in JSP pages.

This is similar to accessing `LanguageUtils` from Java.

```
<%@ page import="com.endeca.portlet.util.LanguageUtils" />
<portlet:defineObjects />
<%= LanguageUtils.getMessage(renderRequest, "reset") %>
```

Instantiate the object and call instance methods from Java/JSP
You can instantiate the `LanguageUtils` object and call methods from Java/JSP.

This approach provides the same convenience methods as the static approach, but simplifies the method signatures by removing the need to specify the request on every call. This may be useful for developers who make many calls for localized strings and would prefer to instantiate the object once and simplify the subsequent method calls.

```
<%@ page import="com.endeca.portlet.util.LanguageUtils" %>
<%
LanguageUtils lang = new LanguageUtils(renderRequest);
%>
<%= lang.getMessage("reset") %>
<%= lang.getMessage("num-records", "Num records:") %>
<%= lang.getMessage("search-for", "Search for \"{0}\"", new String[]{
"American" }) %>
```

Retrieve all messages from the resource bundle in one call from Java/JSP
You can retrieve all messages at once, in a single call from Java/JSP.

This approach may improve performance in portlets that require frequent access to the resource bundle and want to consolidate the message retrieval to a single call. The rest of the page then makes lookups into the loaded map.

```
<%@ page import="com.endeca.portlet.util.LanguageUtils" %>
<%@ page import="java.util.Map" %>
<%
Map<String, String> messages = LanguageUtils.getAllPortletMessages(render¬
Request);
%>
<%= messages.get("reset") %>
<%= messages.get("num-records") %>
<%= LanguageUtils.replaceMessageTokens(messages.get("search-for"), new
String[]{ "American" }) %>
```

Working with non-Unicode characters

This section describes how to work with non-Unicode characters in the Discovery Framework.

Because the Discovery Framework is Java-based, it can only read Unicode or Latin-1 characters. In the case of other characters, you can work around this limitation by converting the native file to ASCII, using a converter such as *native2ascii*, which is freely available as part of the JDK.

Keep in mind the following guidelines:

1. Use UTF-8 as your encoding. Lesser encodings cannot properly represent Japanese characters.
2. Pick a valid character set, such as Shift-JIS or UTF-8/Unicode, and stick with it. You cannot change character sets midstream—if you change character sets, you must re-enter your values.
3. Make sure the character set in your text editor matches the character set in `native2ascii`.

More information about working with non-Unicode characters can be found on the Liferay Portal Website.

Localizing a component to a non-Unicode language
The following example demonstrates how to localize a component to a double-byte, extended character language.

If you want to use this example as a learning exercise but do not have non-Unicode text of your own to deploy, you can machine-translate your English-language file and use that text in step 5 below.

To localize your portlet to a non-Unicode language (such as Japanese):

1. Within your portlet, create a file `PluginResource_<locale-code>.properties.native` at the appropriate location. For example, if you are working with Japanese, the file name would be `PluginResource_ja.properties.native`.
2. Commit both the `.native` and `.properties` file to your portlet. The `.properties` file is used by the portlet, but because that file uses escaped Unicode notation, it is extremely hard for humans to read. It is easier to make any necessary changes in the `.native` file.
3. Open the `.native` file in an encoding- and character-set-aware text editor such as Notepad++. Make sure the `.native` file uses UTF-8 as its encoding and Shift-JIS as its character set.
4. Copy the contents of the English resource bundle into the `.native` file.
5. Within your text editor, using your translation service, replace the English values with the Japanese values.
6. Save the file.
7. From the command line, run Java's `native2ascii` converter. This tool is typically included in the JDK. In the `encoding` argument, specify Shift_JIS as the character set, your `.native` file as the input, and your final `.properties` file as the output.

   ```
   native2ascii -encoding Shift_JIS PluginResource_ja.properties.native
   PluginResource_ja.properties
   ```

8. Commit both the `.native` and `.properties` file to your portlet. The `.properties` file is used by the portlet, but uses escaped Unicode notation, which is hard to read. The `.native` file is easier to modify.

Obtaining more information about portal localization

This topic provides links to additional information about localization provided by Liferay.

For information about editing Liferay's `Language_<langcode>.properties` file, which Liferay uses to localize the portal's strings, see the section "Languages and Time Zones" in the *Liferay Portal Administrator's Guide*. You can use this information to modify Liferay's translations as necessary.

For extensive documentation on Liferay language display customization, see this *wiki page*.

Index

M

N

O

P

Q

R

S

T

U

W