Oracle® Big Data Discovery

Extensions Guide

Version 1.3.2 • August 2016

ORACLE

Copyright and disclaimer

Table of Contents

Preface

Oracle Big Data Discovery is a set of end-to-end visual analytic capabilities that leverage the power of Apache Spark to turn raw data into business insight in minutes, without the need to learn specialist big data tools or rely only on highly skilled resources. The visual user interface empowers business analysts to find, explore, transform, blend and analyze big data, and then easily share results.

About this guide

This guide describes how to use the Custom Visualization Component to develop unique visualizations for your particular business needs.

Audience

This guide is intended for developers who want to create custom components for Studio.

Conventions

The following conventions are used in this document.

Typographic conventions

The following table describes the typographic conventions used in this document.

Typeface	Meaning
User Interface Elements	This formatting is used for graphical user interface elements such as pages, dialog boxes, buttons, and fields.
`Code Sample`	This formatting is used for sample code segments within a paragraph.
Variable	This formatting is used for variable values. For variables within a code sample, the formatting is *`Variable`*.
`File Path`	This formatting is used for file names and paths.

Path variable conventions

This table describes the path variable conventions used in this document.

Path variable	Meaning
`$ORACLE_HOME`	Indicates the absolute path to your Oracle Middleware home directory, where BDD and WebLogic Server are installed.

Path variable	Meaning
`$BDD_HOME`	Indicates the absolute path to your Oracle Big Data Discovery home directory, `$ORACLE_HOME/BDD-<version>`.
`$DOMAIN_HOME`	Indicates the absolute path to your WebLogic domain home directory. For example, if your domain is named `bdd-<version>_domain`, then `$DOMAIN_HOME` is `$ORACLE_HOME/user_projects/domains/bdd-<version>_domain`.
`$DGRAPH_HOME`	Indicates the absolute path to your Dgraph home directory, `$BDD_HOME/dgraph`.

Contacting Oracle Customer Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. This includes important information regarding Oracle software, implementation questions, product and solution help, as well as overall news and updates from Oracle.

You can contact Oracle Customer Support through Oracle's Support portal, My Oracle Support at *https://support.oracle.com*.

Part I

Custom Visualization Component

Chapter 1

Overview of the Custom Visualization Component

This section defines a Custom Visualization Component and describes how to develop a Custom Visualization Component for use in Studio.

About the Custom Visualization Component

A Custom Visualization Component is an extension to Studio that lets you create customized visualizations in cases where the default components in Studio do not meet your specific data visualization needs. A developer creates a custom component, tests it, modifies it, and publishes it to become available to business users. A business user then creates and configures an instance of the custom component on a project page in Studio.

Elements of a Custom Visualization Component

A Custom Visualization Component is made of the following:

- A JavaScript file to define the features, rendering, and interaction of the custom component with its data. You code this file to conform to the Custom Visualization Portlet JavaScript API.
- An EQL statement to provide one or more result sets for the component. This may include EQL token configuration to define variables in an EQL query.

You specify the JavaScript, the EQL, and the EQL token configuration on the **Custom Visualizations** page of Studio along with additional configuration of the custom component.

Installed libraries and external libraries

BDD has D3 version 3 and jQuery version 2.0.3 installed by default. If you need to access additional JavaScript libraries for use in your component, you specify them as a list of external JavaScript libraries when you create the component.

Role privileges

You must have Administrator privileges to access the **Custom Visualizations** page in Studio. Once you publish a Custom Visualization Component, any user with project access can create an instance of the custom component on a project page.

Reference API documentation

The Custom Visualization Portlet JavaScript API has generated JavaScript documentation that is available as part of the full BDD documentation set. You can use this documentation as a reference to code the JavaScript file for a custom component.

For details, see the *Custom Visualization Portlet JavaScript API Reference*.

Requirements for using the Custom Visualization Component

Before developing a Custom Visualization Component, make sure that you meet the following requirements.

Supported platforms

While Big Data Discovery is always deployed on a Linux system, you can develop a Custom Visualization Component on either a Windows or Linux system.

Required knowledge and skills

In order to work with a Custom Visualization Component, you should be familiar with the following:

- JavaScript development and charting libraries.
- Writing EQL statements to provide one or more result sets for a custom component. For details about writing EQL queries, see the *EQL Reference*.

Installing the Custom Visualization Component

No additional installation tasks are required.

Downloading the Custom Visualization Component Sample

On the Oracle Technology Network, you can download a ZIP file containing the Custom Visualization Component Sample.

The ZIP file contains a complete example of a custom component, including:

- A JavaScript file that illustrates how to use the Custom Visualization Component API. The sample code implements a Donut Pie chart visualization and the code provides a reference for building your own component.
- A text file of configuration settings that you use to populate fields on the **Custom Visualization** page of Studio. This provides the configuration settings for the sample JavaScript code and a tokenized EQL statement to generate results for the component.

Chapter 2

Developing a Custom Visualization Component

This section describes how to create your own Custom Visualization Component and publish it for use in Studio.

Creating a Custom Visualization Component

You create a component by coding a JavaScript file that uses the Custom Visualization Component JavaScript API to initiate queries and render the returned data to the component. You upload the file as part of the component configuration. Also you write one or more EQL statements to provide the result set for the component.

Before performing this task, you should code the JavaScript file for the custom component to conform to the Custom Visualization Component JavaScript API. You upload the file in the steps below. For details about coding the JavaScript file, see the *Custom Visualization Component JavaScript API Reference*.

When a business user creates a custom component on a project page in Discover, Studio loads the JavaScript code to render the component.

The JavaScript code has two major requirements:

- It must extend `Oracle.BDD.Portlets.Visualization.Renderers.BaseRenderer`.
- It must implement the `init()` function. This is executed on each Discover page load and it typically queries for a result set and directs the responds in the component.

To create a Custom Visualization Component:

1. In the Studio header, click the **Configuration Options** icon and select **Control Panel>Custom Visualizations**.
2. Click **+ Component**.
3. Specify a name for the component.

   This is the display name of the component as it displays on the Component menu of the **Discover** page.

4. Optionally, click **Browse** to locate an icon image for the component and then click **Upload** and **Ok**.

   This is the icon image for the component as it displays on the Component menu of the **Discover** page.

5. In **JavaScript File**, click **Browse** to locate the JavaScript file that implements your Custom Visualization Component and click **Open**.

6. In **Renderer class**, specify the fully qualified name of the JavaScript class that renders the component.

   For example, in the following JavaScript snippet, the name of the renderer class is `Oracle.BDD.Portlets.Visualization.Renderers.DonutPieChart`:

```
Oracle.BDD.Portlets.Visualization.Renderers.DonutPieChart
= Oracle.BDD.Portlets.Visualization.Renderers.BaseRenderer.extend({

    init: function() {

        /**
         * Get the queryConfig for the initial query
         */
        var queryConfig = this.getQueryConfig("eql");
...
```

7. Select a **Sort type** of one of the following:
   - **None** - Specifies that there is no sort option available for the business user to configure in the component.
   - **Defined** - Specifies that the sort option for the component is defined by token replacement in an EQL statement. A business user then sorts by value of the token in either an ascending or descending order.
   - **Per Dimension** - Specifies that a sort option is available for any token defined as a dimension.

8. Optionally, expand **Advanced Options** and specify the following:
   - **CSS** - Specifies visualization-specific CSS. Also note that the CSS is scoped to the visualization's DOM container.
   - **External CSS** - Specifies a list of URLs to external CSS files. Each URL should be line separated.
   - **External JavaScript** - Specifies a list of URLs to external JavaScript files. Each URL should be line separated. These JavaScript files might provide additional resources, such as third-party plug-ins, to support the JavaScript file.

   For example:

```
https://cdnjs.cloudflare.com/ajax/libs/EventEmitter/4.2.11/EventEmitter.min.js
https://cdnjs.cloudflare.com/ajax/libs/UAParser.js/0.7.9/ua-parser.min.js
```

9. Click **Next**.

10. In **EQL query name**, specify the name of the EQL query used in the JavaScript API function `getQueryConfig`.

    For example, if your Javascript file contains:

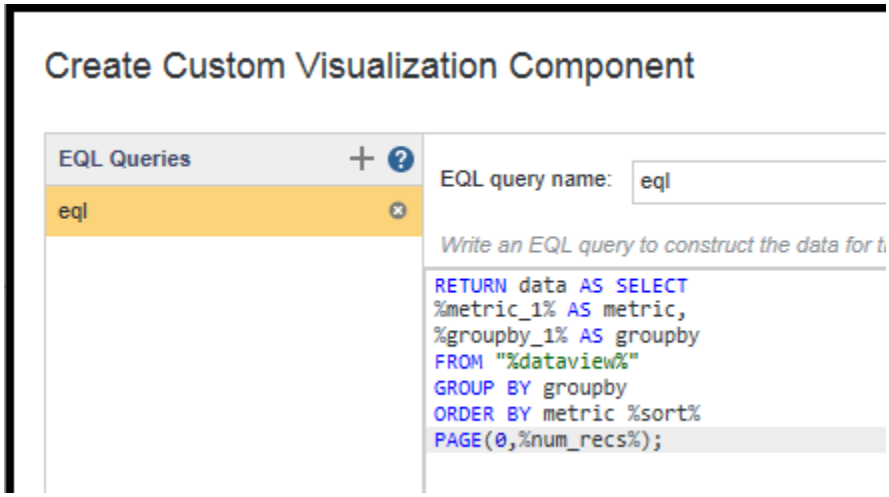
```
var queryConfig = this.getQueryConfig("eql");

...
```

    then the EQL query name you specify is `eql`.

11. Specify the EQL query for your component in the text box.

    For example:



    For more details about writing EQL statements, see the *EQL Reference*.

12. Optionally, you can add more EQL queries to the component by clicking **+** and specifying a new name and EQL query in the text box.

    Each query that you add generates an additional result set for the component to use.

13. Click **Detect Tokens**.

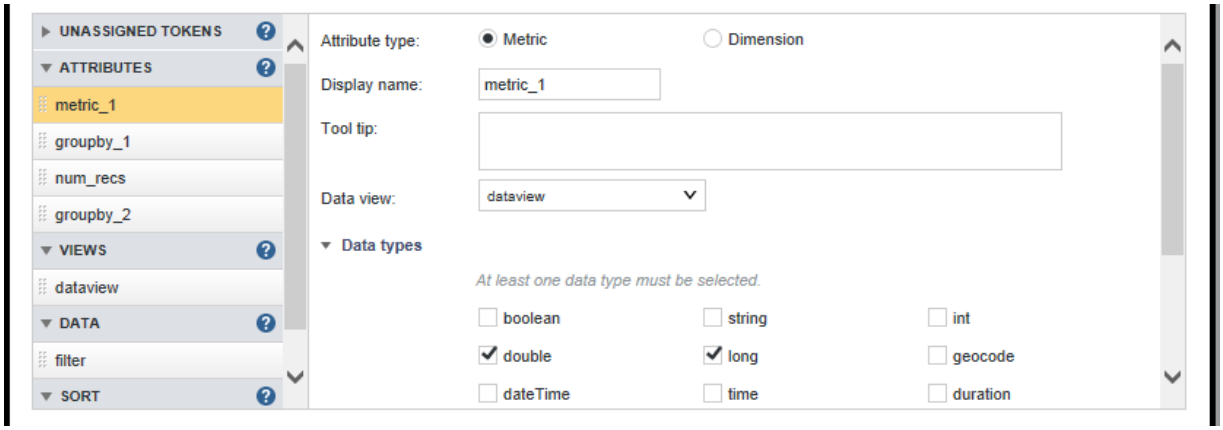
    Studio examines the tokens in the EQL query, and based on the syntax of how the tokens are used, Studio adds the tokens to **Attributes**, **Views**, **Data**, and **Sort** categories. For example, tokens in a `FROM` clause appear in **Views**. Tokens in a `WHERE` clause appear in **Data**.

14. If any of the tokens are incorrectly categorized, click the token and drag and drop it into the correct **Attributes**, **Views**, **Data**, or **Sort** category.

15. Click each token name and specify the details of how the token is used in the component.

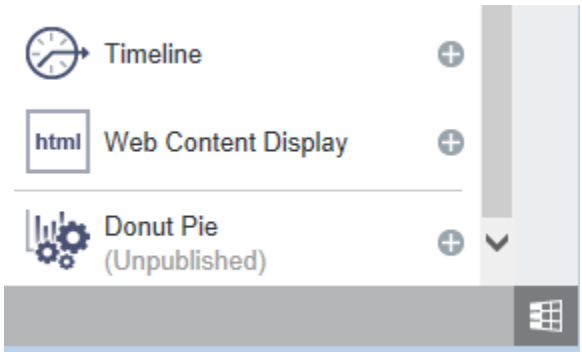
    This steps configures how a business user interacts with the token values on the Discover page. You specify whether a token is a metric or a dimension, its display name, its data view, its data type, and aggregation type.

    For example, the settings shown here configure the `metric_1` token:



16. Click **Save**.

The new component is available at the bottom of the **Add Component** menu on the **Discover** page. For example, here is the Donut Pie example used in the procedures:



The component is not available to business users until you publish it.

Before publishing, you should test the component by working with it as a business user would and adjusting the JavaScript or other configuration if necessary to modify the component's behavior. Once the component works correctly, you publish it for use by other Studio users.

Using tokens in an EQL query

The EQL queries for a Custom Visualization Component support token replacement in the EQL query. Tokens are simply variables in an EQL query that correspond to user-interface controls in the **Visualization Settings** panel the component. Controls include attributes (metrics or dimensions), views, data views and sorts. For

example, a sort token in an EQL query creates an ASC or DSC sort control in the component configuration for a project user to select. A dimension token creates a drop down menu of attributes for a project user to select.

In EQL query, tokens are strings enclosed by percentage signs (%), for example, `%metric_1%`. Here is an example EQL statement that contains five tokens (`%metric_1%`, `%groupby_1%`, `%dataview%`, `%sort%`, `%num_recs%`):

```
RETURN data AS SELECT
%metric_1% AS metric,
%groupby_1% AS groupby
FROM "%dataview%"
GROUP BY groupby
ORDER BY metric %sort%
PAGE(0,%num_recs%);
```

Studio replaces the tokens with a value based on the user configuration in the **Visualization Settings** panel the component. That value is used when Studio runs the query to generate results for the component.

Token types

Tokens are distinguished by the type of data the token represents and by how the component acquires the token's value. The following table shows each token type and explains how Studio replaces the token with a value before running the EQL query.

Token type	Value	Example replacement value
Attribute (metric or dimension sub-types)	Attribute tokens represent either a metric or a dimension. A *metric* token represents menus of metric and aggregation functions. During component configuration, you select a data type for the metric token to determine which attributes are available for a project user to select. Similarly, during component configuration, you also select aggregation functions (e.g. SUM) to determine which attributes are available for a project user to select. A metric token is replaced when a project user selects an attribute and aggregation function from the **Visualization Settings** panel of the component. A *dimension* token represents a drop-down menu of attributes for a project user to select. During component configuration, you select a data type for the dimension token to determine which attributes are available for a project user to select. For example, if you select data type of string, only string attributes are available for selection. A dimension token is replaced when a user selects an attribute from the **Visualization Settings** panel of the component. Both types of metric and dimension tokens are associated with a View token that dictates which data view the attributes are taken from to populate the user-interface menus. Attribute token values may also be set with the JavaScript API.	SUM(p_price)

Token type	Value	Example replacement value
View	A view token represents a drop-down menu of data views for a project user to select. The token is replaced when a user selects a data view from the **Visualization Settings** panel of the component. View token values may also be set with the JavaScript API.	p_price
Sort	A sort token represents an ASC or DESC sort control in the component configuration for a project user to select. The token is replaced when a user selects a sort direction (ASC or DESC) from the **Visualization Settings** panel of the component. ASC is the default value. Sort token values may also be set with the JavaScript API.	ASC
Data	Data token values must be set with the JavaScript API.	

For additional details about tokens, see the *Custom Visualization Component JavaScript API Reference*.

Token substitution example

Here is an example EQL query with tokens:

```
RETURN data AS SELECT
%metric_1% AS metric,
%groupby_1% AS groupby
FROM "%dataview%"
GROUP BY groupby
ORDER BY metric %sort%
PAGE(0,%num_recs%);
```

Here is the same EQL query with values substituted for the tokens:

```
RETURN data AS SELECT
SUM(p_price) AS metric,
p_color AS groupby
FROM "wine_dataset"
GROUP BY groupby
ORDER BY metric ASC
PAGE(0,20);
```

Editing JavaScript during development

As a troubleshooting convenience, Studio provides an inline JavaScript editor so you can modify a component's JavaScript directly. You do not have to upload the file again using the Add Component wizard.

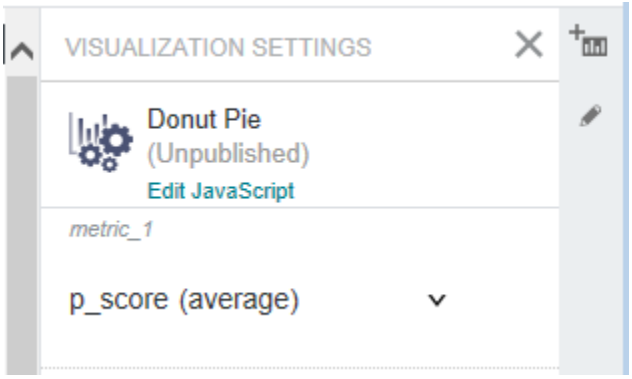
You modify the JavaScript inline as part of debugging the component. This JavaScript editor is available only while a component is unpublished.

You must have already created a custom visualization component and added it to a project before you can edit the component's JavaScript.

To edit JavaScript during development:

1. Open a project and add your Custom Visualization Component to a Discover page.
2. Click the pencil icon to edit the visualization settings.
3. Click the **Edit JavaScript** link.

   For example:



4. In the Edit JavaScript editor, modify the component's code as necessary to adjust its behavior.
5. Click **Save and exit**.

The code changes take place immediately.

Publishing a Custom Visualization Component

After you are satisfied that a Custom Visualization Component behaves as desired, you publish it to make it available on the Discover page of Studio to all Studio users. Unpublished components are available only to administrators.

To publish a Custom Visualization Component:

1. In the Studio header, click the **Configuration Options** icon and select **Custom Visualizations**.
2. Locate the component you want to make available and click **Published**.

The component immediately becomes available on the component menu of the Discover page.

Unpublishing a Custom Visualization Component

To remove a Custom Visualization Component from the component menu, you unpublish it. Previously created instances of the component are still available on the Discover page, but business users cannot create new instances of the component after it has been unpublished.

An unpublished component is still stored in Studio, so a developer with the administrator role can modify it if necessary and publish it again later.

To unpublish a Custom Visualization Component:

1. In the Studio header, click the **Configuration Options** icon and select **Custom Visualizations**.
2. Locate the component that you want to make unavailable and deselect **Published**.
3. Click **Unpublish**.

Deleting a Custom Visualization Component

Deleting removes the component from the component menu of the Discover page and removes the component from any project where it was used.

To delete a Custom Visualization Component:

1. In the Studio header, click the **Configuration Options** icon and select **Custom Visualizations**.
2. Locate the component you want to remove and click **Remove**.
3. In the confirmation dialog, click **Delete**.

Part II

Studio Component SDK

Chapter 3

Installing and Configuring the Component SDK

The Component SDK supports custom development for components and data security.

About the Component SDK

You can use the Component SDK to extend Studio by creating and deploying custom Security Managers and components.

You can create Security Managers to restrict access to specific data.

You can create custom components to visualize data in Studio. Once you deploy a custom component, it can be added to a project page.

As part of developing custom components, you can also create custom `QueryFunctions`, used to retrieve and display data on a component.

To see the full generated documentation for the Component SDK, see the *Component SDK API Reference (Javadoc)*.

Requirements for using the Component SDK

Before using the Component SDK, make sure that you meet the system and skill set requirements.

Required knowledge and skills

In order to work with the Component SDK, you should be familiar with Java development and JavaScript.

Components are extensions of a custom version of the Java Portlet class, so to develop a custom component, you should also have some understanding of Java portlets and the Portlet specification.

The Component SDK generates Eclipse projects, so it also helps to be familiar with Eclipse.

Supported platforms

While Big Data Discovery is always deployed on a Linux system, you can use the Component SDK from either a Windows or Linux system.

There are .bat and .sh versions of each of the Component SDK scripts.

Software requirements

All Component SDK work requires the following:

- Eclipse. You must use a version that supports JDK 1.7.
- JDK 1.7 or above
- Apache Ant 1.8.4 or higher, to build your custom items

For custom components, you may also need:

Software or License	Description
Ext JS	While Ext JS is not required, and the sample component provided with the Component SDK does not use it, most Big Data Discovery components were developed using *Ext JS 3.4*. Big Data Discovery does not include a license for Ext JS. If you want to use Ext JS for custom component development, you must obtain your own copy of it.
YUI Compressor 2.4.8	By default, when you compile a custom component, JavaScript minification is not used. While components do build successfully without JavaScript minification, for performance purposes you may want to enable it. If you enable minification, then files in the `docroot/js` directory of your custom components are minified. In order to be able to use minification to build components, you must obtain the .jar file for version 2.4.8 of YUI Compressor. The file is available at *https://github.com/yui/yuicompressor/releases/download/v2.4.8/yuicompressor-2.4.8.jar*.
JUnit	If you are planning to create unit tests for your custom components, you will need to first obtain `junit.jar`. The Component SDK can use JUnit for unit tests, but does not come with the `junit.jar` file.

Installing the Component SDK

The Component SDK is contained in a .zip file in the Big Data Discovery Media Pack.

To install the Component SDK:

1. From the Big Data Discovery Media Pack, download the Component SDK .zip file (`component-sdk-<versionNumber>.zip`).

2. Extract the Component SDK .zip file to a separate directory.

   The directory path to the Component SDK cannot contain spaces.

Once you have installed the Component SDK, you can continue with your custom development.

For information on developing custom Security Managers, see *Developing a Custom Security Manager on page 22*.

For information on developing custom components, see *Developing Custom Components on page 25*.

Preparing your system for Component SDK development

After installing the Component SDK, before you can start development, you must complete some initial preparation on your system.

This includes:

- Extracting the Studio .ear file and portal .war file
- Configuring build files to point to the directories for these extracted files
- Optionally, enabling JavaScript minification for custom components.

  If minification is enabled, then files in the `docroot/js` directory of custom components are minifiied.

To prepare your system for custom component development:

1. Extract the Studio .ear file and portal .war file:
   (a) From the Big Data Discovery Media Pack, download the .ear file.
   (b) Extract the .ear file to a directory on your machine.
   (c) From that directory, extract the file `endeca-portal.war` to a directory within the extracted .ear file directory.

   For example, if the .ear file is extracted to `/bdd_ear`, the contents of the extracted .war file might be in `/bdd_ear/portal/`.
2. Next, in the Component SDK, create and configure the build properties files:
   (a) Go to the `components` directory of the Component SDK.
   (b) In the `components` directory, create a file called `build.<user>.properties`, where `<user>` is the user name that you use to log in to the current machine.

   For example, if your user name is jsmith, then you would create a file called `build.jsmith.properties`.
   (c) Add the following properties to `build.<user>.properties`:

```
portal.base.dir=<extracted .ear file directory>
app.server.lib.global.dir=<extracted .ear file directory>/APP-INF/lib
app.server.portal.dir=<extracted portal .war file directory>
war.output.dir=<directory for generated components>
```

   The `war.output.dir` setting indicates where the build process should place the .war file that it generates when you compile a custom component. This can be any directory on your system.

So for example, if:

- You extracted the .ear file to a directory called `/bdd_ear`
- You extracted the portal .war file to a `portal` directory in `/bdd_ear`
- You want the generated .war files for custom components to be placed in `/generated_components`

the settings would be:

```
portal.base.dir=/bdd_ear
app.server.lib.global.dir=/bdd_ear/APP-INF/lib
app.server.portal.dir=/bdd_ear/portal
war.output.dir=/generated_components
```

(d) In the `components` directory, create a file called `build.shared.properties`.

(e) In `build.shared.properties`, add the following property:

```
portal.base.dir=<extracted .ear file directory>
```

3. To enable JavaScript minification when building custom components:

(a) If you haven't already, obtain the required YUI Compressor .jar file. See *Requirements for using the Component SDK on page 19*.

(b) In the `components` directory of the Component SDK, update `build.<user>.properties` to add the following property:

```
yui.compressor.jar=<path to YUI Compressor .jar file>
```

4. In Eclipse, create the following Eclipse classpath variables:

Name	Path
`DF_GLOBAL_LIB`	Path to the application server global library, which is: `<extracted .ear file directory>/APP-INF/lib`
`DF_PORTAL_LIB`	Path to the Web application library, which is: `<extracted portal .war file directory>`

Chapter 4

Developing a Custom Security Manager

Using the Component SDK, you can create a custom Security Manager to customize how Big Data Discovery filters the data that users see.

Creating and implementing a new Security Manager

The Component SDK includes a batch script for creating a new Security Manager.

To create a new Security Manager project:

1. From a command prompt, change to the `components/endeca-extensions` directory in the Component SDK.
2. Run the appropriate version of the `create-bddsecuritymanager` command.

   For Linux:

   ```
   ./create-bddsecuritymanager.sh <securityManagerName>
   ```

   For Windows:

   ```
   create-bddsecuritymanager.bat <securityManagerName>
   ```

   Where *<securityManagerName>* is the name you want to use for the security manager. For example:

   ```
   ./create-bddsecuritymanager.sh restrict-region-data
   ```

   The name cannot have spaces.

   This command creates a *<securityManagerName>* directory in `bddsecuritymanager`.

   This directory is an Eclipse project that you can import directly into Eclipse.

   It also contains a sample implementation that can help you understand how the Security Manager is used.

   Note: The sample implementation illustrates one way to use the API. The sample is not intended to provide a recommended design pattern for a production application.

3. Your Security Manager must implement the `applySecurity` method.

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

The Query class in this signature is `com.endeca.portal.data.Query`. This class provides a simple wrapper around a Conversation Service request.

Security Manager interface

The `com.endeca.portal.data.security.BddSecurityManager` interface represents a Security Manager capable of applying record-level security filters for BDD.

For additional details about `BddSecurityManager`, see the *Component SDK API Reference*.

Class Summary Item	Item Value or Description
Abstract base class	`com.endeca.portal.data.security.AbstractBddSecurityManager`
Concrete implementation class	`com.endeca.portal.data.security.AttributeAclSecurityManager`
Implementation behavior	The `AttributeAclSecurityManager` implementation filters records in a data set (collection) according to Access Control List (ACL) multi-assign attributes which have been added to each record during a data ingest. The class assumes that these attributes are named: • `__allow_user` for user-permissions • `__allow_group` for group-permissions • `__allow_role` for role-permissions This implementation requires a collection/data-set to have all three of these attributes if it is to be secured, even if one or more of them is not used. It is also required that each of these attributes must be multi-assign string attributes (i.e., type=mdex:string and isSingleAssign=false). Each record is filtered according to the name of the user and those of the groups/roles held by that user, the names of which need to be assigned to the above attributes.

The SDK package contains a `SampleBddSecurityManager.java` that is based on `AttributeAclSecurityManager`. The file is included in the `bddsecuritymanager.zip`, which is in the `components/endeca-extensions` directory in the Component SDK.

Building and deploying a new Security Manager

Before you can use your custom Security Manager, you must deploy it to Studio. To do this, you generate a .jar file for it, then add the .jar file to the Studio .ear file.

To build and deploy a custom Security Manager:

1. From the `<securityManagerName>-mdexsecuritymanager` directory you created for your new Security Manager, run the Ant build script.

   This generates a .jar file named `<your-security-manager-name>-bddsecuritymanager.jar`, and places it in the Security Manager project directory.
2. Add the .jar file to the `app-inf/lib` directory within the deployed .ear file for Studio.
3. Redeploy the .ear file.

Configuring Studio to use a different Security Manager

In order to for Studio to use your Security Manager, you must configure Studio to pick up and use the new class.

To configure Studio to use a different Security Manager:

1. On the **Control Panel** menu, click **Studio Settings**.
2. Change the value of `df.bddSecurityManager` to the full name of your class, similar to the following example:

   ```
   df.bddSecurityManager = com.endeca.portal.extensions.YourSecurityManagerClass
   ```
3. Click **Update Settings**.
4. To have the change take effect, restart Studio. You may also need to clear any cached user sessions.

Chapter 5

Developing Custom Components

The most common use of the Component SDK is to create and deploy custom components.

Generating the Eclipse project for the component

The Component SDK includes a script to generate an Eclipse project for a new component.

New components are extensions of the `EndecaPortlet` class, which is in turn an extension of the basic Java Portlet class.

To create a new component:

1. At a command prompt, change to the `components/portlets` directory in the Component SDK.
2. Run the appropriate `.sh` or `.bat` version of the `create` command:

   For example:

   ```
   create.sh <componentName> "<componentDisplayName>"
   ```

   Where:

Parameter	Description
<componentName>	The name of the component. The component name: • Must be all lower case. • Cannot have spaces. • Cannot include the string `-ext`, because it causes confusion with the ext plug-in extension. For example, `my-component-extension` would not be a valid name.
<componentDisplayName>	The display name for the component. If the display name contains spaces, it must be enclosed in quotation marks.

   For example:

```
create.sh my-test "My New Test Component"
```

The script creates in the `portlets` directory a new directory for the new component.

The directory is the component name, with `endeca-` pre-pended and `-portlet` appended automatically. For example, if you set the name to `my-test`, the directory is named `endeca-my-test-portlet`.

This directory is an Eclipse project that you can import directly into Eclipse.

3. Import the project into Eclipse.

   If your components depend on shared library projects located within the `/shared` directory, import those as well.

   Note that it takes some time for projects to build after they are imported.

After you generate and import the component project, you can begin the actual component development.

Obtaining query results for components

When developing a component, use the `QueryState` and `QueryResults` classes to request and receive data from data sets.

To specify the types of results the component needs, you must add the relevant `QueryConfigs` to the `QueryState`. For example:

```
QueryState query = getDataSource(request).getQueryState();
CollectionBaseView defaultBaseView = EndecaPortletUtil.getDefaultCollection(request);
query.addFunction(new NavConfig(), defaultBaseView, request.getLocale());
QueryResults results = getDataSource(request).execute(query);
```

You can then get the underlying Conversation Service API results in order to obtain the data required by your component.

```
Results discoveryResults = results.getDiscoveryServiceResults();
```

Before executing the query, you can also make other local modifications to your query state by adding filters or configurations to your query. For example:

```
String viewKey = request.getParameter(VIEW_KEY_PARAM);
DataSource ds = getDataSource(request);
QueryState query = ds.getQueryState();
SemanticView sView = ds.getCollectionOrSemanticView(viewKey, request.getLocale());
query.addFunction(new ResultsConfig(), sView, request.getLocale());
ExpressionBase expression = getDataSource(request).parseLQLExpression("Region = 'Midwest'");
query.addFunction(new SelectionFilter(expression), sView, request.getLocale());
QueryResults results = getDataSource(request).execute(query);
```

To persist `QueryState` changes to the user's session, which also updates the associated components, use `setQueryState`. For example:

```
String viewKey = request.getParameter(VIEW_KEY_PARAM);
DataSource ds = getDataSource(request);
QueryState query = ds.getQueryState();
SemanticView sView = ds.getCollectionOrSemanticView(viewKey, request.getLocale());
query.addFunction(new ResultsConfig(), sView, request.getLocale());
ExpressionBase expression = getDataSource(request).parseLQLExpression("Region = 'Midwest'");
query.addFunction(new SelectionFilter(expression), sView, request.getLocale());
ds.setQueryState(query);
```

For details on the `QueryConfig` and `QueryFunction` classes, see *Working with QueryFunction Classes on page 29*, and the *Component SDK API Reference*.

Building a component

After completing the component development, you set the build properties, then build the component in Eclipse.

To build a component:

1. Before building the component, you need to make sure the build properties are set correctly. Open the `build.xml` in the root directory of the component.

   By default, the build properties 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="" />
   ```

   These properties are used as follows:

Property	Description
`shared.libs`	Controls which projects in the `shared/` directory to include in the component. These shared projects are compiled and included as `.jar` files where appropriate.
`endeca-common-resources.includes`	Controls which files in the `shared/endeca-common-resources` project are copied into the component. The default value is `"**/*"`, indicating that all of the files are included. These files provide AJAX enhancements (`preRender.jspf` and `postRender.jspf`).
`endeca-common-resources.excludes`	Controls which files from the `shared/endeca-common-resources` project are excluded from the component. By default, the value is `""`, indicating that no files are excluded. If your component needs to override any of these files, you must use this build property to exclude them. If you do not exclude them, your code will be overwritten.

   You can specify the `includes` and `excludes` properties for any shared library. For example:

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

2. Once the build properties are set, then in your Eclipse project, open the `build.xml` file.
3. If the project is not configured to build automatically, then in the outline view, right-click the deploy task and select **Run as...>Ant Build**.

The build process generates the component .war file, and places it in the output directory you specified. The .war file has the same name as the component.

Deploying and removing custom components

Once you have built the component .war file, you can add the component to a Big Data Discovery instance. You can also remove a component.

To deploy and remove components:

1. To deploy a custom component:
   (a) Open the Studio .ear file.
   (b) Add the component .war file to the root of the .ear file, with the other component .war files.
   (c) In the `meta-inf` directory of the .ear file, open `application.xml`
   (d) Add an entry for the new component, then save the file.

   For example:

```
<module>
    <web>
        <web-uri>my-new-component-portlet.war</web-uri>
        <context-root>/eid/my-new-component-portlet/</context-root>
    </web>
</module>
```

   (e) Redeploy the .ear file.
   (f) Restart Big Data Discovery.

   During the startup process, you can check the Big Data Discovery logs to confirm that the component loaded successfully.
2. After redeploying the .ear file, to test that the component was added successfully:
   (a) Log in to Big Data Discovery.
   (b) From within a Big Data Discovery project, click the add component option.

   Your component should be included in the list of available components.
   (c) Drag and drop the new component onto the page.
3. To remove a component:
   (a) Open the Big Data Discovery .ear file.
   (b) Remove the component .war file.
   (c) In `meta-inf/application.xml`, remove the entry for the component.
   (d) Redeploy the .ear file.

Chapter 6

Working with QueryFunction Classes

When developing custom components, you can use the provided `QueryFunction` classes to filter and query data. You can also create and implement your own `QueryFunction` classes.

Provided QueryFunction filter classes

Big Data Discovery provides the following QueryFunction filter classes. Filters are used to change the current query state.

The available filter classes are:

- `DataSourceFilter`
- `RefinementFilter`
- `NegativeRefinementFilter`
- `RangeFilter`, including the following date/time-specific range filters that extend `RangeFilter`:
  - `DateRangeFilter`
  - `TimeRangeFilter`
  - `DurationRangeFilter`
- `DateFilter`
- `LastNDateFilter`
- `GeoFilter`
- `SearchFilter`

In addition to the information here, for more details on the `QueryFunction` filter classes, see the *Component SDK API Reference*.

DataSourceFilter

Uses an EQL snippet to provide the filtering. `DataSourceFilter` refinements are not added to the **Selected Refinements** panel.

The available properties are:

Property	Description
`filterString`	The EQL snippet containing the filter information. For a `DataSourceFilter`, this would be the content of a `WHERE` clause for an EQL statement. For details on the EQL syntax, see the *EQL Reference*.

For example, to filter data to only show records from the Napa Valley region with a price lower than 40 dollars:

```
ExpressionBase expression = dataSource.parseLQLExpression("Region='Napa Valley' and P_Price<40");
DataSourceFilter dataSourceFilter = new DataSourceFilter(expression);
```

RefinementFilter

Used to filter data to include only those records that have the provided attribute values. `RefinementFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `RefinementFilter` are:

Property	Description
`attributeValue`	String The attribute value to use for the refinement.
`attributeKey`	String The attribute key. Identifies the attribute to use for the refinement.
`sourceCollectionKey`	String The key of the data set. This is typically a long encoded value that starts with `default_edp`.

Property	Description
`multiSelect`	`AND |OR | NONE` For multi-select attributes, how to do the refinement if the filters include multiple values for the same attribute: • If set to `AND`, then matching records must contain all of the provided values. • If set to `OR`, then matching records must contain at least one of the provided values. • If set to `NONE`, then multi-select is not supported. Only the first value is used for the refinement. This setting must match the refinement behavior configured for the attribute in the data set. For information on using the **Views** page to view and configure the refinement behavior for an attribute, see the *Studio User's Guide*.

In the following example, the data is refined to only include records that have a value of 1999 for the Year attribute.

```
RefinementFilter refinementFilter = new RefinementFilter("1999", "Year", "default_edp_cc7ea");
```

NegativeRefinementFilter

Used to filter data to exclude records that have the provided attribute value. `NegativeRefinementFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `NegativeRefinementFilter` are:

Property	Description
`attributeValue`	String The attribute value to use for the refinement.
`attributeKey`	String The attribute key. Identifies the attribute to use for the refinement.
`attributeType`	`BOOLEAN | STRING | DOUBLE | LONG | GEOCODE | DATETIME | TIME | DURATION` The type of value to use for the refinement. The default is `STRING`. If the attribute is a type other than string, then you must provide the type.

Property	Description
`attributeValueName`	String Optional. The value to display on the **Selected Refinements** panel for the refinement. If you do not provide a value for `attributeValueName`, then the **Selected Refinements** panel displays the value of `attributeValue`.
`ancestors`	Not supported.
`isAttributeSingleAssign`	Boolean. If set to `true`, then the attribute can only have one value. If set to `false`, then the attribute is multi-value. For information on using the **Views** page to see whether an attribute is multi-value, see the *Studio User's Guide*.
`sourceCollectionKey`	String The key of the data set. This is typically a long encoded value that starts with `default_edp`.

In the following example, the data is refined to only include records that do NOT have a value of Washington for the Region attribute. Because Region is a string attribute, no other configuration is needed.

```
NegativeRefinementFilter negativeRefinementFilter
= new NegativeRefinementFilter("Region", "Washington");
```

In the following example, the data is refined to only include records that do NOT have a value of 1997 for the P_Year attribute, which is a single-assign attribute. Because P_Year is not a string attribute, the attribute type LONG is specified.

```
NegativeRefinementFilter negativeRefinementFilter
= new NegativeRefinementFilter("P_Year", "1997", PropertyType.LONG,
   true, "default_edp_cc7ea760");
```

RangeFilter

Used to filter data to include only those records that have attribute values within the specified range. `RangeFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `RangeFilter` are:

Property	Description
`attributeKey`	String The attribute key. Identifies the attribute to use for the filter.

Property	Description
`rangeOperator`	`LT | LTEQ |GT |GTEQ| BTWN |GCLT |GCGT | GCBTWN` The type of comparison to use. • `LT` - Less than • `LTEQ` - Less than or equal to • `GT` - Greater than • `GTEQ` - Greater than or equal to • `BTWN` - Between. Inclusive of the specified range values. • `GCLT` - Geocode less than • `GCGT` - Geocode greater than • `GCBTWN` - Geocode between
`rangeType`	`DECIMAL | INTEGER | DATE | GEOCODE | TIME | DURATION` The type of value that is being compared.
`value1`	Numeric The value to use for the comparison. For `BTWN`, this is the low value for the range. For the geocode range operators, the origin point for the comparison.
`value2`	Numeric For a `BTWN`, this is the high value for the range. For `GCLT` and `GCGT`, this is the value to use for the comparison. For `GCBTWN`, this is the low value for the range.
`value3`	Numeric Only used for the `GCBTWN` operator. The high value for the range.

In the following example, the data is refined to only include records where the value of P_Score is a number between 80 and 100:

```
RangeFilter rangeFilter
= new RangeFilter("P_Score", RangeType.INTEGER, RangeOperator.BTWN, "80", "100");
```

There are also date/time-specific range filters that extend `RangeFilter`:

- `DateRangeFilter`
- `TimeRangeFilter`
- `DurationRangeFilter`

DateFilter

Used to filter date values. Using a `DateFilter`, you can filter by subsets of the date/time value. For example, you can filter a date attribute to include all records with a specific year or specific month.

The properties for a `DateFilter` are:

Property	Description
`dateFilters`	A list of `DateFilterDimension` objects that represent the date filters to apply. Each `DateFilterDimension` object consists of: • `DatePart` constants identify each date part • Integer values to represent the values for each date part The filter only filters down to the most specific date part provided.

In the following example, the data is refined to only include records where SalesDate is June 15, 2006. The filter only provides the year, month, and day. Even if records have different hour-minute-second values for SalesDate, as long as they are within June 15, 2006, they still match this filter:

```
DateFilterDimension dfd = new DateFilterDimension();
dfd.addDatePartFilter(DatePart.YEAR, 2006);
dfd.addDatePartFilter(DatePart.MONTH, 6);
dfd.addDatePartFilter(DatePart.DAY_OF_MONTH, 15);
DateFilter dateFilter = new DateFilter("SalesDate", dfd);
```

LastNDateFilter

Used to filter the date to include records with a date attribute with a value in the last n years, months, or days.

The properties for a `LastNDateFilter` are:

Property	Description
`attributeKey`	The key name of the attribute.
`ticksBack`	The number of years, months, or days within which to include records in the results.
`datePart`	The date part to use for the filtering. The possible values are: • `YEAR` • `MONTH` • `DAY_OF_MONTH` • `HOUR` • `MINUTE` • `SECOND`

Property	Description
`sourceCollectionKey`	String The key of the data set. This is typically a long encoded value that starts with `default_edp`....

In the following example, the data is refined to only include records with SalesDate values from the last 3 years:

```
LastNDateFilter lastNDateFilter = new LastNDateFilter("SalesDate", 3, DatePart.YEAR);
```

GeoFilter

Used filter data to include records with a geocode value within a specific distance of a specific location.

The properties for a `GeoFilter` are:

Property	Description
`attributeKey`	The key name for the geocode attribute.
`rangeOperator`	The comparison operator.
`value1`	A geocode value to use as the starting point.
`radius`	The number of miles or kilometers within which to search.
`locationName`	The name of a location to use as the starting point.
`unit`	The unit of distance (mi or km) for the comparison.

SearchFilter

Used to filter the data to include records that have the provided search terms. `SearchFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `SearchFilter` are:

Property	Description
`searchInterface`	String Either the name of the search interface to use, or the name of an attribute that is enabled for text search.
`terms`	String The search terms.

Property	Description
`matchMode`	`ALL` | `PARTIAL` | `ANY` | `ALLANY` | `ALLPARTIAL` | `PARTIALMAX` | `BOOLEAN` The match mode to use for the search.
`enableSnippeting`	Boolean Whether to enable snippeting. Optional. If not provided, the default is `false`.
`snippetLength`	Integer The number of characters to include in the snippet. Required if `enableSnippeting` is `true`. To enable snippeting, set `enableSnippeting` to `true`, and provide a value for `snippetLength`.

In the following example, the filter uses the "default" search interface to search for the terms "California" and "red". The matching records must include all of the search terms. Snippeting is supported, with a 100-character snippet being displayed.

```
SearchFilter.Builder builder = new SearchFilter.Builder("default", "California red");
builder.matchMode(MatchMode.ALL);
builder.enableSnippeting(true);
builder.snippetLength(100);
SearchFilter searchFilter = builder.build();
```

Provided QueryConfig functions

Studio provides the following `QueryConfig` functions, used to manage the results returned by a query. These are more advanced functions for component development.

Each `QueryConfig` function generally has a corresponding function in `DiscoveryServiceUtils` to get the results.

`QueryConfig` functions are most often used to obtain results that are specific to a component. Because of this, `QueryConfig` functions should never be persisted to the application data domain using `setQueryState()`, as this would affect all of the components that are bound to the same data. Instead, `QueryConfig` functions should only be added to a component's local copy of the `QueryState` object.

The available `QueryConfig` functions are:

- `AttributeTextValueSearchConfig`
- `AttributeValueSearchConfig`
- `BreadcrumbsConfig`
- `LQLQueryConfig`
- `RecordDetailsConfig`
- `ResultsConfig`

- `ResultsSummaryConfig`
- `SearchAdjustmentsConfig`
- `SortConfig`

In addition to the information here, for more details on the `QueryConfig` functions, see the *Component SDK API Reference*.

AttributeTextValueSearchConfig

Used for text searches, such as in the **Available Refinements** panel and the **Search Box** functions.

`AttributeTextValueSearchConfig` has the following properties:

Property	Description
`searchTerm`	String The term to search for in the attribute values.
`attribute`	String (optional) The attribute key for the attribute to search in. Use the `attribute` property to search against a single attribute. To search against multiple attributes, use `searchWithin`.
`searchWithin`	List<String> (optional) A list of attributes to search in for matching values.
`languageId`	String (optional) The country code for a supported language (such as "en" for English).

The following example searches for the term "merlot":

```
AttributeTextValueSearchConfig attributeTextValueSearchConfig
= new AttributeTextValueSearchConfig("merlot");
```

AttributeValueSearchConfig

Used for type-ahead in a search field. For example, used for **Available Refinements** to narrow down the list of available values for an attribute.

`AttributeValueSearchConfig` has the following properties:

Property	Description
`searchTerm`	String The term to search for in the attribute values.

Property	Description
`maxValuesToReturn`	int (optional) The maximum number of matching values to return. If you do not provide a value, then the default is 10.
`attribute`	String (optional) The attribute key for the attribute to search in. Use the `attribute` property to search against a single attribute. To search against multiple attributes, use `searchWithin`.
`searchWithin`	List<String> (optional) A list of attributes to search in for matching values.
`matchMode`	`ALL|PARTIAL|ANY|ALLANY|ALLPARTIAL|PARTIALMAX|BOOLEAN` (optional) The match mode to use for the search.
`relevanceRankingStrategy`	String (optional) The name of the relevance ranking strategy to use during the search.
`languageId`	String (optional) The country code for a supported language (such as "en" for English).

The following example searches for the term "red" in the WineType attribute values:

```
AttributeValueSearchConfig attributeValueSearchConfig
= new AttributeValueSearchConfig("red", "WineType");
```

BreadcrumbsConfig

Used to return the refinements associated with the query.

`BreadcrumbsConfig` has the following property:

Property	Description
`id`	String (optional) The ID of the breadcrumbs to be instantiated.

This example returns the refinements:

```
BreadcrumbsConfig breadcrumbsConfig = new BreadcrumbsConfig();
```

LQLQueryConfig

Executes an EQL query on top of the current filter state.

`LQLQuery` has the following property:

Property	Description
`lqlQuery`	AST The EQL query to add. To retrieve the AST from the query string, call `DataSource.parseLQLQuery`.

The following example retrieves the average of the P_Price attribute grouped by Region:

```
Query query
= dataSource.parseLQLQuery("return mystatement as select avg(P_Price) as avgPrice group by Region",
true);
LQLQueryConfig lqlQueryConfig = new LQLQueryConfig(query);
```

RecordDetailsConfig

Sends an attribute key-value pair to assemble the details for a selected record. The complete set of attribute-value pairs must uniquely identify the record.

`RecordDetailsConfig` has the following property:

Property	Description
`recordSpecs`	List<RecordSpec> Each new `RecordDetailsConfig` is appended to the previous `RecordDetailsConfig`.

The following example sends the value of the P_WineID attribute:

```
List<RecordSpec> recordSpecs = new ArrayList<RecordSpec>();
recordSpecs.add(new RecordSpec("P_WineID", "37509"));
RecordDetailsConfig recordDetailsConfig = new RecordDetailsConfig(recordSpecs);
```

ResultsConfig

Used to manage the returned records. Allows for paging of the records.

`ResultsConfig` has the following properties:

Property	Description
`recordsPerPage`	Long The number of records to return at a time.

Property	Description
`offset`	Long (optional) The position in the list to start at. The very first record is at position 0. For example, if `recordsPerPage` is 10, then to get the second page of results, the offset would be 10.
`columns`	String[] (optional) The columns to include in the results. If not specified, then the results include all of the columns.
`numBulkRecords`	Integer (optional) The number of records to return. Overrides the value of `recordsPerPage`.

The following example returns a selected set of columns for the third page of records, where each page contains 50 records:

```
ResultsConfig resultsConfig = new ResultsConfig();
resultsConfig.setOffset(100);
resultsConfig.setRecordsPerPage(50);
String[] columns = {"WineID", "Name", "Description", "WineType", "Winery", "Vintage"};
resultsConfig.setColumns(columns);
```

ResultsSummaryConfig

Gets the number of records returned from a query.

```
ResultsSummaryConfig resultsSummaryConfig = new ResultsSummaryConfig();
```

SearchAdjustmentsConfig

Returns DYM (Did You Mean) and auto-correction items for a search.

```
SearchAdjustmentsConfig searchAdjustmentsConfig = new SearchAdjustmentsConfig();
```

SortConfig

Used to sort the results of a query. Used in conjunction with `ResultsConfig`.

`SortConfig` has the following properties:

Property	Description
`ownerId`	String (optional) The ID of the `ResultsConfig` that this `SortConfig` applies to. If not provided, uses the default `ResultsConfig` ID. If you configure a different ID, then you must provide a value for `ownerId`.
`property`	String The attribute to use for the sort.
`ascending`	Boolean Whether to sort in ascending order. If set to `false`, then the results are sorted in descending order.

For example, with the following `SortConfig`, the results are sorted by the P_Score attribute in descending order:

```
SortConfig sortConfig = new SortConfig("P_Score", false);
```

Creating and deploying a custom QueryFunction class

The Component SDK allows you to create custom `QueryFunction` classes.

Generating the Eclipse project for the QueryFunction class

The Component SDK includes a script to generate the Eclipse project for the `QueryFunction` class.

To generate the Eclipse project for a new `QueryFunction` class:

1. From the command line, change to the `components/endeca-extensions` subdirectory of the Component SDK.
2. To create a `QueryFilter` class, run the appropriate `.sh` or `.bat` version of the `create-queryfilter` command.

   For example on Linux:

   ```
   ./create-queryfilter.sh <queryFilterName>
   ```

Where <*queryFilterName*> is the name you want to use for the `QueryConfig` class. The name cannot have spaces.

The command creates a new directory called <*queryFilterName*>`-QueryFilter` in the `endeca-extensions` directory.

This directory is an Eclipse project that you can import directly into Eclipse.

It contains an empty sample implementation of a `QueryFilter`.

3. To create a `QueryConfig` class, run the appropriate `.sh` or `.bat` version of the `create-queryconfig` command.

   For example on Linux:

   ```
   ./create-queryconfig.sh <queryConfigName>
   ```

   Where <*queryConfigName*> is the name you want to use for the `QueryConfig` class. The name cannot have spaces.

   The command creates a new directory called <*queryConfigName*>`-QueryConfig` in the `endeca-extensions` directory.

   This directory is an Eclipse project that you can import directly into Eclipse.

   It contains an empty sample implementation of a `QueryConfig`.

For both `QueryFilter` and `QueryConfig` classes, the skeleton implementation:

- Extends either `QueryFilter` or `QueryConfig`.
- Creates stubs for the `applyToDiscoveryServiceQuery`, `toString`, and `beforeQueryStateAdd` methods.

  `applyToDiscoveryServiceQuery` and `toString` are required methods that you must implement.

  `beforeQueryStateAdd` is an optional method to verify the query state before the function is added. This method is used to prevent invalid query states such as duplicate refinements.
- Creates a no-argument, protected, empty constructor. The protected access modifier is optional, but recommended.
- Creates a private member variable for logging.

Implementing a custom QueryFunction class

After you create your new `QueryFunction` class, you then implement it.

To implement your new `QueryFunction`, you must:

- Add private filter or configuration properties.
- Create getters and setters for any filter properties you add.
- Define a no-argument constructor (protected access modifier optional, but recommended).
- Implement the `applyToDiscoveryServiceQuery` method.

  This method is called with the following arguments:

  - The Conversation Service query
  - A `stateName` string

Your custom function should use the Conversation Service API to apply itself to the conversation service query argument.

The `stateName` argument provides the value to use for state name references in Conversation Service filters or content element configs that your custom function adds to the query.

- Implement the `toString` method, which is used to compare `QueryFunction` instances for equality.

  `toString` should be consistent and deterministic in order to accurately determine if two instances of your custom `QueryFunction` are identical or distinct.

- Optionally, implement the `beforeQueryStateAdd(QueryState state)` method to check the current query state before the function is added.

Building and deploying a custom QueryFunction class

When you have finished development on your custom `QueryFunction` class, you build it, then add the resulting .jar file to the .ear file.

To build and deploy a `QueryFunction`:

1. In your Eclipse project for the `QueryFunction`, open the `build.xml` file.
2. If the project is not configured to build automatically, then in the outline view, right-click the deploy task and select **Run as...>Ant Build**.

   The Component SDK builds the `QueryFunction`, and places the resulting .jar file in the output directory you specified.
3. To make the `QueryFunction` available to all of your custom components, place the .jar file in the `app-inf/lib` directory of the extracted .ear file.
4. To add the `QueryFunction` to the Big Data Discovery instance:

   (a) Add the .jar file to the `app-inf/lib` directory of the .ear file.

   (b) Re-deploy the .ear file.

Adding a custom QueryFunction to a custom component project

If you just want to use a custom `QueryFunction` in a specific custom component, you add its `.jar` file to the component's Eclipse build path.

To add the `QueryFunction` to a custom component project:

1. In Eclipse, right-click the component project, then select **Build Path>Configure Build Path**.
2. Click the **Libraries** tab.
3. Click **Add Variable**.
4. Select **DF_GLOBAL_LIB**.

   You should have added this variable when you set up the Component SDK. See *Preparing your system for Component SDK development on page 21*.
5. Click **Extend**.
6. Open the `ext/` directory.

7. Select the `.jar` file for your custom `QueryFunction`.
8. Click **OK**.

After adding the `.jar` file to the build path, you can import the class, and use your custom `QueryFilter` or `QueryConfig` to modify your `QueryState`.

Index