EMC® Documentum® D2

Version 4.2

Developers Guide

EMC Corporation
Corporate Headquarters:
Hopkinton, MA 01748-9103
1-508-435-1000
www.EMC.com

Table of Contents

Preface

EMC® Documentum® D2 consists of two components:

- D2 Configuration: The web-based application, hereafter known as D2 Config, for administrators to use to configure settings such as automated content-handling processes and background settings for D2 Client.
- D2 Client: The web-based application, hereafter known as D2 Client, for users that provides the ability to interact with content in one or more repositories.

When this guide refers to D2, it refers to the product as a whole, not the individual components.

Intended audience

The information in this guide is for developers who create and configure extensions for the D2 using the D2 and D2 Foundation Services (D2FS) API.

Revision history

The following table lists changes in this guide.

Revision Date	Description
January 2014	Initial publication.

Chapter 1

Extending D2

This chapter contains the following topics:

Introduction to Extending D2

Before customizing D2 using extensions, carefully re-examine the functionality available through D2 and its official plug-ins. In almost all cases, using existing capabilities to achieve your use cases provide a cheaper and more reliable solution.

D2 offers the following methods of extending D2 using the D2 API:

- Building integrated solutions by using the D2FS services to interact with the D2 web application.
- Using Open Ajax Hub (OAH) and Javascript to create external web applications that can be manipulated as iframes from within the D2 web application.
- Using D2 Java packages to create and configure D2 plug-ins that add custom actions or override existing D2 services.

You can use any Integrated Development Environment (IDE) such as Eclipse for building the code.

Chapter 2

New Features and Changes

This chapter contains the following topics:

Content Checkin Service

D2 4.1 performed checkin for new content using two operations: performing he checkin with the `checkin()` service and then adding the new content using the `getUploadUrls()` service. The D2 4.2 checkin process uses the new `IDownloadService.getCheckinUrls()` service to perform the checkin operation as a single, atomic operation.

To use the new checkin process, cease usage of the `checkin()` and `getUploadUrls()` services and use the `getCheckinUrls()` service. You can continue to use the `checkin()` service for content-less objects and the `getUploadUrls()` method for non-checkin operations.

The *EMC Documentum D2 D2FS API JavaDoc* contains further information on `getCheckinUrls()` and its required attributes.

Chapter 3

Overviewing the Architecture of D2

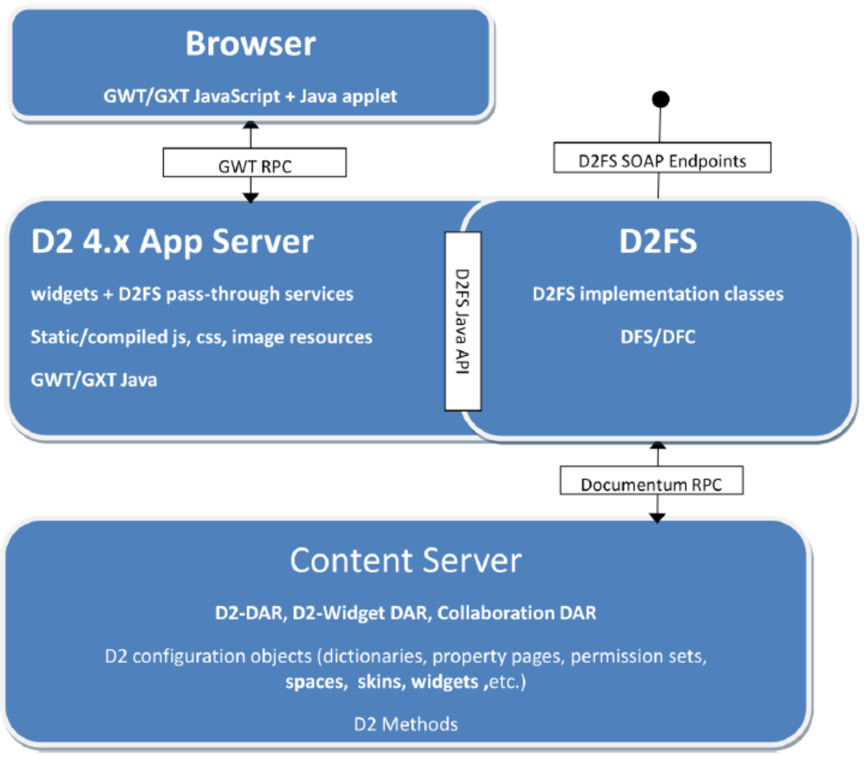
This chapter contains the following topics:

Understanding the D2 Client Architecture

D2 Client is comprised of three layers:

- The Browser layer denoting the web browsers used by end users in connecting to D2.
- The web application server layer corresponding to the installation and configuration of `D2.war`.
- The Documentum Content Server layer consisting of installed DAR files.

The following graphic shows the composition and relationship of the three layers:



Browser
GWT/GXT JavaScript + Java applet
GWT RPC
D2FS SOAP Endpoints
D2 4.x App Server
widgets + D2FS pass-through services
Static/compiled js, css, image resources
GWT/GXT Java
D2FS Java API
D2FS
D2FS implementation classes
DFS/DFC
Documentum RPC
Content Server
D2-DAR, D2-Widget DAR, Collaboration DAR
D2 configuration objects (dictionaries, property pages, permission sets, spaces, skins, widgets ,etc.)
D2 Methods

Browser Layer

D2 Client is a web 2.0 browser-based application. End users navigate to the D2 Client web application URL and log in to use D2. D2 Client is:

- GWT/GXT-based: D2 Client uses Sencha GXT and the Google Web Toolkit (GWT) to manage JavaScript. As a result, the D2 Client JavaScript code is delivered hidden and in a single large file. You cannot extend or debug D2 Client without the original source.

  The Sencha website (http://www.sencha.com/products/gxt) contains further information on GXT.
- Loaded onto a single HTML page: D2 Client interprets and loads all JavaScript into browser memory at the start. As a result, D2 manipulates the HTML DOM to remove and replace interface portions rather than switching HTML pages upon user navigation.
- GWT RPC-based: D2 uses standard HTTP requests for all communication between the browser client and the D2 web application. For requests for static resources, such as images, D2 uses GWT RPC calls backed by the D2FS services layer to retrieve dynamic data.

  The Google developer site (https://developers.google.com/) has more information on GWT RPC.

Application Server Layer (D2.war)

D2 Client is packaged as a .war file that can be deployed on most J2EE application servers. The web application server is comprised of:

- The D2 web application: Includes most of the presentation logic, including the GWT and GXT libraries, custom interface controls, static web resources, and pass-through services to allow web browsers to invoke D2FS services. The layer does not link to the Documentum Foundation Classes (DFC) or Documentum Foundation Services (DFS) client libraries, because all communication between the D2 web application and the Documentum repository passes through the D2FS interface.
- The D2FS library: Contains all D2 business logic, including the communication between the Documentum Content Server through DFC and DFS, the logic for interpreting and applying D2 configurations to service requests, and a set of SOAP services.

  The application server layer uses the D2FS Java API while allowing external clients to use the SOAP interface because D2FS merged with D2. Plug-ins to D2 exist in the D2FS layer and can intercept existing calls to the D2FS interface to change or enhance D2 behaviors.

Content Server Layer

The D2 installation includes three DAR files:

- `D2-DAR.dar`: Includes most of the Documentum types necessary for storing configuration objects in the repository. Configuration objects are subtypes of `d2_module_config` or `d2_moduledoc_config` depending on whether the configuration is stored as object metadata or as XML content. D2 favors XML for representing forms, dialog boxes, mail templates, and other structured content, where a set of repeating attributes is insufficient for representing a configuration.

  The D2 DAR includes jobs and methods used by D2 for implementing some D2 configurations. These methods can be invoked by a scheduled job for completing background tasks, or invoked by the D2 web application in response to events. The *EMC Documentum D2 Administrator Guide* contains further information about D2 jobs and methods.

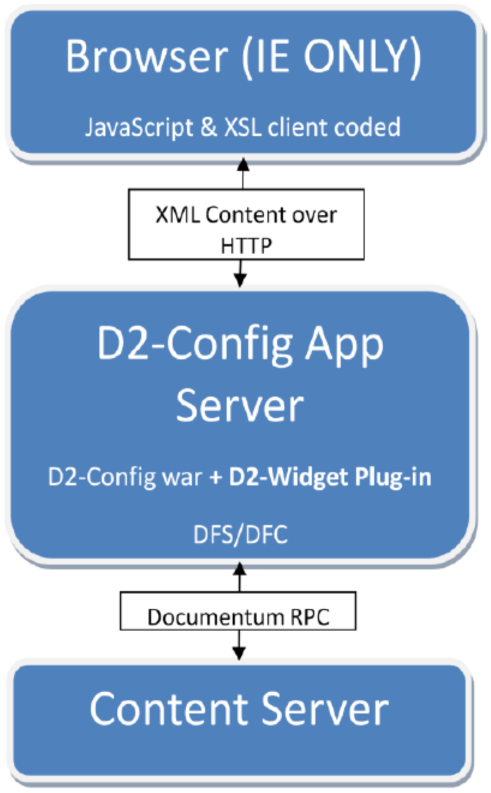
- `D2-Widget-DAR.dar`: Includes configuration types related to the D2 4.*x* interface, such as widgets, workspace layouts, and themes.
- `Collaboration_Services.dar`: Includes types and Documentum Business Object Framework modules for Documentum collaboration capabilities. D2 requires this DAR file to enable commenting within D2 Client.

Understanding the D2 Config Architecture

D2 Config is comprised of three layers:

- The Browser layer denoting the web browsers used by end users in connecting to D2 Config.
- The web application server layer corresponding to the installation and configuration of `D2-Config.war` and the D2-Widget plug-in.
- The Documentum Content Server layer.

The following graphic shows the composition and relationship of the three layers:



Browser Layer

D2 Config is a web 2.0 browser-based application. End users navigate to the D2 Config web application URL and log in to use D2. D2 Config is:

- ActiveX control-based: D2 Config uses ActiveX control like the old D2 3.*x* architecture and can only be accessed using Microsoft Internet Explorer browsers.

- XML-based: D2 Config uses the old D2 3.*x* architecture for D2 and delivers the interface to the web browser in XML representation.

Application Server Layer (D2-Config.war)

D2 Config is packaged as a .war file that can be deployed on most J2EE application servers. The web application server is comprised of:

- The D2 Config web application: Includes most of the presentation logic, including the ActiveX controls. The layer includes the Documentum Foundation Classes (DFC) or Documentum Foundation Services (DFS) client libraries, as the layer controls communication between the D2 Config web application and the Documentum repository.
- The D2-Widget plug-in: Adds configuration components for D2 Client to D2 Config.

Content Server Layer

The D2 installation includes three DAR files:

- `D2-DAR.dar`: Includes most of the Documentum types necessary for storing configuration objects in the repository. Configuration objects are subtypes of `d2_module_config` or `d2_moduledoc_config` depending on whether the configuration is stored as object metadata or as XML content. D2 uses XML for representing forms, dialog boxes, mail templates, and other structured content, where a set of repeating attributes is insufficient for representing a configuration.

  The D2 DAR includes jobs and methods sed by D2 for implementing some D2 configurations. These methods can be invoked by a scheduled job for completing background tasks, or invoked by the D2 web application in response to events. The *EMC Documentum D2 Administrator Guide* contains further information about D2 jobs and methods.
- `D2-Widget-DAR.dar`: Includes configuration types related to the D2 4.*x* interface, such as widgets, workspace layouts, and themes.
- `Collaboration_Services.dar`: Includes types and Documentum Business Object Framework modules for Documentum collaboration capabilities. D2 requires this DAR file to enable commenting within D2 Client.

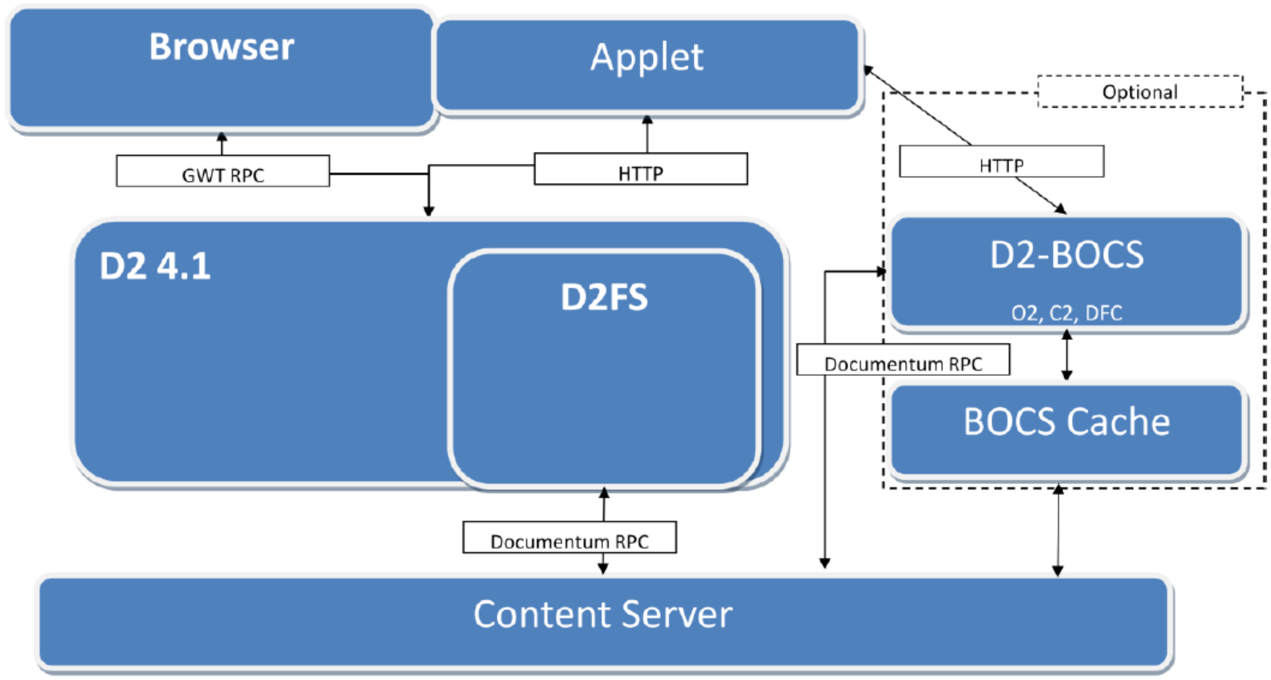
Understanding D2 File Transfer

D2 uses either the D2 Java applet or HTML5 (thin client mode) to perform content transfer, giving it several advantages over the standard HTML4 <fileinput> form control:

- Multiple-file upload.
- Upload of entire folder structures.
- Drag-and-drop of files from the desktop for upload.
- Client-side compression of content before the upload.

Browsers that support HTML5 can run D2 without the Java applet in thin client mode but do so with limited functionality. The *EMC Documentum D2 Installation Guide* contains information about the functionality restricted by thin client mode and instructions for setting the file transfer mode.

The following graphic shows the data flow of content transfer between the Documentum Content Server and D2 web applications:



In basic installations with most users in close geographic proximity, the D2 applet uses a standard HTTP connection to connect directly back to the D2 web application.

Use the Documentum Branch Office Caching Services (BOCS) cache to speed up transfer of large files to users in remote locations. D2 requires the D2-BOCS plug-in to be installed onto the same server as the BOCS cache to:

- Facilitate communication between the D2 applet and the BOCS cache.
- Allow D2 to manipulate content in and out of the repository. For example, the C2 plug-in can be used to insert configuration-based watermarks.

D2-BOCS must apply configuration rules to complete C2 and O2 content operations. As a result, D2-BOCS opens a DFC session with the repository, which makes Documentum RPC calls to the repository to retrieve configuration objects and content metadata. Due to the relatively high latency of the connection between the BOCS cache and the Content Server, administrators and end users should try to limit the number of applicable O2 and C2 configurations. Configure D2 so that only large files can be transferred using the BOCS cache to reduce load on the connection and to connect directly to the D2 application for the transfer of smaller files.

Chapter 4

Building Integrations Using D2FS

This chapter contains the following topics:

Using SOAP to Allow External Applications to Affect D2 Processes and Data

The D2FS API is exposed from the D2 web application as a set of SOAP services. Generating Java Stub Libraries to Call D2FS Via SOAP, page 17 contains instructions for generating the Java stub libraries required for invoking D2FS using SOAP services.

The D2 SDK package contains sample Java classes that use the Java stub libraries to invoke and modify D2 processes and data.

The *EMC Documentum D2 D2FS API JavaDoc* contains further information on the available D2FS services.

Generating Java Stub Libraries to Call D2FS Using SOAP

1. Deploy D2 to a local application server.
2. Using the following command, generate the Java source files for the stub libraries needed to connect to D2:

   ```
   wsimport —keep http://localhost:8080/D2/ws/d2fs.wsdl
   ```

   The `-keep` parameter saves the files so that you can copy them to your project.
3. If you generated stubs using a different D2 server than the one you used to call D2FS, modify the URL by opening `ModelPortService.java` and editing the following line:

   ```
   static { url = new URL("http://<D2 server address and
   port>/D2/ws/d2fs.wsdl"); }
   ```

D2FS Examples

Navigate to the `D2FS/Java/D2FSExamples/src/com/emc/d2/d2fs/examples` folder in the D2 SDK package for the following example Java classes:

Class	Description
ContextAndLoginExample.java	Demonstrates how to establish a D2FS login context with the D2 server.
ImportAndUploadDocumentExample.java	Demonstrates how to use `createproperties()` to import and upload content.
DestroyExample.java	Demonstrates how to destroy content.
SavePropertiesExample.java	Demonstrates how to set content attributes.
SearchQuickExample.java	Demonstrates how to run a quick search.
SearchAdvancedExample.java	Demonstrates how to: • Use SearchService and ContentService to search the repository. • Save or run an advanced search. • List saved searches. • Run a saved search.
CheckinContentExample.java	Demonstrates how to use `getCheckinUrls()` to : • Check out content. • Request a checkin URL. • Post new content with checkin parameters.
TaxonomyEnumAndExportUrlExample.java	Demonstrates how to use ContentService to: • Enumerate taxonomy objects. • Download and save the exported taxonomy.

Setting Up the D2FS Examples, page 18 contains instructions for setting up the D2FS examples after generating the Java stub libraries.

Setting Up the D2FS Examples

1. Using Eclipse, create a new workspace and import existing projects into the workspace from the `D2FS/Java/D2FSExamples` folder.
2. Delete the `com.emc.d2fs.*` and `org.w3.*` packages if they are already contained by the project.
3. Copy the generated `com` and `org` folders to your project.

   You can paste the folders directly from the local file directory to the `src/` folder of your IDE.
4. Update `LoginInfo.java` to specify your repository, login, password, and the URL of the web application.
5. Check the setup comments at the top of the example to confirm the required inputs.

Chapter 5

Using Open Ajax Hub (OAH) to Create and Configure a Sample External Widget

This chapter contains the following topics:

Understanding Open Ajax Hub (OAH) and External Widgets

Open Ajax Hub (OAH) is a standard JavaScript library for publishing and subscribing to web applications that is defined by the OpenAjax Alliance to address interoperability and security issues that arise when multiple Ajax libraries and components are used in the same web page. You can use OAH to:

- Publish to a channel to send a message.
- Subscribe to a channel to listen for a message.

A D2 external widget is a web application URL hosted in an iframe within the D2 Client application. D2 uses OAH to communicate between widgets and to broadcast events and actions such as content selection, object location, and content view.

- Events notify listeners about the D2 application and widget-specific attributes.
- Actions cause D2 or widgets to perform a specified operation.

The D2 implementation of OAH, called D2-OAH.js, provides the binding between a web page and the surrounding D2 application. D2 Config refers to these web pages as external widgets, which are hosted in an iframe within the D2 Client web application.

The `X3PubSubEvents` section of the *EMC Documentum D2 Open Ajax Hub (OAH) Event and Actions JavaDoc* contains a complete reference of the OAH message objects, the D2-OAH API, and the list of available actions and events.

Setting Up the Sample External Widget

1. Extract `UpdateDocList` folder from `D2 4.2 SDK.zip` to a temporary location.
2. Stop your web application server.
3. Copy the `X3-Ext-UpdateDoclist` folder to the `<web application server>/webapps/` folder.

4. Start your web application server.
5. Log in to D2 Config:
   a. Navigate to **File** > **Import configuration** from the menu bar.
   b. Import `X3-Ext-UpdateDoclist — Config.zip`

      The *EMC Documentum D2 Administration Guide* contains further instructions on importing configurations.
   c. Navigate to **Widget view** > **Widget** from the menu bar.
   d. Select the `WG EXT UpdateDoclist` widget.
   e. Modify the **Widget url** field to match the location of your deployed web application server. For example:

      `http://myserver:8080/X3-Ext-UpdateDoclist`

      The sample external widget also contains a second URL that displays an Open Ajax Hub publish and subscribe application:

      `http://myserver:8080/X3-Ext-UpdateDoclist/devindex.html`
6. Log in to D2 Client and add the new widget to your workspace.

Walking Through the Sample External Widget

The `X3-Ext-UpdateDoclist` external widget shows how to:

- Create a D2OpenAjaxHub object to connect the external widget to D2.
- Use the Open Ajax Message Hub to execute a query form and update the doclist widget.
- Use the Open Ajax Hub to subscribe to D2 events and receive messages.

Open `X3-Ext-UpdateDoclist/index.html` in a text editor to view the sample code.

The `X3PubSubEvents` section of the *EMC Documentum D2 Open Ajax Hub (OAH) Event and Actions JavaDoc* contains a complete reference to the Open Ajax Hub (OAH) classes and methods as well as a complete list of D2 events, actions, and their corresponding attributes.

Importing OAH Scripts

You must import the following two Javascript scripts:

- Use one of the following:
  - `http://`*`YourServer`*`:8080/D2/container/OpenAjaxManagedHub-all-obf.js`
  - `http://`*`YourServer`*`:8080/D2/container/external-api/OpenAjaxManagedHub-all.js`

  `OpenAjaxManagedHub-all-obf.js` is the smaller version used by D2 for production purposes and `OpenAjaxManagedHub-all.js` is the full version for development purposes.
- `http://`*`YourServer`*`:8080/D2/container/external-api/D2-OAH.js`

Creating and Connecting an Open Ajax Hub

Create and connect a `D2OpenAjaxHub()` object. The sample code uses the following code:

```
var d2OpenAjaxHub = new D2OpenAjaxHub();
// connect hub providing callback functions.
d2OpenAjaxHub.connectHub(connectCompleted, onInitWidget, onActiveWidget);
```

You can then set a callback for a successful connection to the hub to perform initialization tasks. For example, the sample waits for the hub client to connect and then initiates subscribing to events.

```
 function connectCompleted(hubClient, success, error) {
  if (success) {

   logit("Hub client connected");

   // subscribe to events
   subscribeEvents();


  } else
   logit("Hub client NOT connected - please check console");
 }
```

The sample widget always subscribes to events even if you do not request that the widget track and display objects of the selected type.

Subscribing to Events

You can interact with the external widget to set or change subscriptions to D2 events. You can then use the subscription to listen to messages and extract information. Select **Track selected type** to track content selected by users in D2 Client, received notifications, and view the document type in the external widget form.

Call the `subscribeToChannel()` method to use this feature of Open Ajax Hub as shown in the sample code:

```
function subscribeEvents() {
  logit("subscribe to events...");
  d2OpenAjaxHub.subscribeToChannel ("D2_EVENT_SELECT_OBJECT",
   selectObjectCallback, true);
 }
```

The `selectObjectCallback` method is an example method in the sample that dictates how the widget uses the message received through this event.

Sending Messages

You can interact with the external widget both as a widget in a D2 Client workspace or by accessing the URL directly through a web browser. The sample web application contains an HTML form that contains the same fields as the `QF HR a_status` query form that is imported when the external widget is installed. Type a document type, select a document status, and click the **Update Doclist** button. The external widget sends an Open Ajax Message to D2 to execute the `QF HR a_status` query form search. The query form then functions as a Doclist widget filter because it updates the Doclist widget with the search results.

Call the `sendMessage()` method to use this feature of Open Ajax Hub as shown in the sample code:

```
function updateDoclist() {

  //Query form configuration name is defined below
  var queryFormConfigName = "QF HR a_status";

  //To update the doclist a new OpenAjax message will be build to be posted
   //in the Hub using the D2-OAH API
  var messageToSend = new OpenAjaxMessage();

  // Specify a non-null ID (to pass request validation)
  messageToSend.put("oam_id", "");

  //In the message, we need to define what properties will be sent.
   //Here a_status and r_object_type
  messageToSend.put("list", "a_status¬r_object_type");

  //We set the a_status value
  messageToSend.put("a_status", a_status.value);
  //We set the r_object_type value
  messageToSend.put("r_object_type", r_object_type.value);

  //set the query form config name which will be used to update the doclist
  messageToSend.put("config", queryFormConfigName);

  //Then we define what service and what method in the service will be called.
  //We call the Search service and the runQueryFormSearch method.
  //Calling this service will update the user's last search object
  messageToSend.put("eService", "Search");
  messageToSend.put("eMethod", "runQueryFormSearch");

  // When the service call completes, we can define an action to be executed.
   //Here, an event will be posted.
  messageToSend.put("rType", "EVENT");

  // As the last search has been updated by the web service call, we will post
  //    the D2_ACTION_SEARCH_DOCUMENT event to display the search results
  messageToSend.put("rAction", "D2_ACTION_SEARCH_DOCUMENT::oam_id==node_last_search");

  //The message is now ready, it can be posted in the Hub
  d2OpenAjaxHub.sendMessage("D2_ACTION_EXECUTE", messageToSend);

 }
```

Chapter 6

Creating and Configuring a Custom Plug-in

This chapter contains the following topics:

Understanding Custom Plug-ins

You can create and configure custom plug-ins using the Java classes and resources packaged with the D2 API framework to:

- Add features as custom actions.

  Configuring a Custom Action, page 27 contains instructions for creating and configuring a custom action.
- Modify existing features by changing pre-processing, post-processing, and overriding D2 services.

  Setting Up D2 Service Overrides, page 30 contains instructions for modifying the D2 services and Understanding Service Overrides, page 28 contains information about the various overrides you can use.

The *EMC Documentum D2 D2FS API JavaDoc* contains more information on the available services and actions.

Overview of the Template Plug-in Examples

The template plug-in located in the `Plugins` folder of the `D2 4.2 SDK.zip` archive contains the following three examples:

Customizing Content Export

This example for customizing content export shows how to use condition service override, how to add a custom service (`IPluginAction`), and how to configure error reporting. The resulting plug-in allows end users to use a menu item in D2 Client to download dynamically-generated custom content using the D2 web service. To do this, the plug-in uses the D2 OpenAjaxHub to post an event named `D2_ACTION_EXPORT_FROM_URL` to trigger content download from the URL contained in a parameter.

The plug-in consists of the following classes:

- `D2CustomService` implements the `IPluginAction` interface to allow being called as a standard D2 web service. The class contains the `getCustomDownloadURL` method that returns a URL with dynamic IDs to the `ExportContent` servlet.
- `D2ExportServicePlugin` intercepts the standard `D2ExportService` web service that is called by the `ExportContent` servlet. The class overrides the `exportTo` method to send custom content to D2.
- D2 uses `D2PluginTemplateVersion` and `D2PluginVersion` for plug-in detection.

After configuring a menu item using D2 Config, an end user can click the menu item in D2 Client to trigger the following chain of events:

1. The plug-in calls the `getCustomDownloadURL` method in the `D2CustomService` service.
2. The method returns a URL to the `ExportContent` servlet in an `oam_value` attribute.
3. The servlet posts the URL as a parameter of the `D2_ACTION_EXPORT_FROM_URL` event.
4. The D2 applet detects the event and triggers a download using the `ExportContent` servlet.
5. The `ExportContent` servlet retrives content from the `D2ExportService` service.
6. The `D2ExportServicePlugin` intercepts the call and sends the custom content.

Overriding a Service with Post-Processing

This example performs a D2 Service but adds post-processing to make `object_name` values upper case.

The plug-in consists of the `D2ContenServicePlugin` class.

Overriding a Service with Pre-Processing

This example performs a D2 Service but adds pre-processing to add a timestamp suffix to the `title` attribute.

The plug-in consiss of the `D2CreationServicePlugin` class.

Downloading and Setting Up the Template Plug-in

1. Download `D2 4.2 SDK.zip` to a temporary location.

2. Extract the `Plugins` folder from `D2 4.2 SDK.zip`
3. Open your IDE and select the extracted `Plugins` folder as a workspace.

   If you select the `YourCo-PluginName` folder, the IDE might not recognize the project.
4. Import the project to the workspace. In Eclipse:
   a. Navigate to **File** > **Import**.
   b. Select **General** > **Existing Projects into Workspace** and click **Next**.
   c. In **Select root directory**, click **Browse** and select the `YourCo-PluginName` folder.
   d. Click **Finish**.
5. Configure the Java Build Paths for D2. In Eclipse:
   a. Right-click `YourCo-PluginName` in the Package Explorer and click **Properties**.
   b. Click **Java Build Path** in the **Properties for** `YourCo-PluginName` dialog box.
   c. Click the **Libraries** tab.
   d. Click **Add Variable**, then click **Configure Variables**.
   e. Click **New** and fill out the form as described in the following table:

Field	**Description**
Name	Type `D2_4x_LIB`
Path	Type the path to the `WEB-INF/lib` folder in the D2 web application. For example: `C:/Program Files/Apache Software Foundation/Tomcat 6.0/webapps/D2/WEB-INF/lib`

   f. Click **New** and fill out the form as described in the following table:

Field	**Description**
Name	Type `D2_4x_CLASSES`
Path	Type the path to the `WEB-INF/classes` folder in the D2 web application. For example: `C:/Program Files/Apache Software Foundation/Tomcat 6.0/webapps/D2/WEB-INF/classes`

   g. Click **OK** until you are back at the Package Explorer window.

Building, Installing, and Verifying the Temple Plug-in

1. Build the plug-in:
   a. In Eclipse, expand the `build` folder in the Package Explorer.
   b. Right-click `build.xml` and navigate to **Run As** > **Ant build**

      When your IDE finishes building the .jar file, the `dist` folder contains `YourCo-PluginName.jar`.
2. Stop the web application server.
3. Copy `YourCo-PluginName.jar` to the `WEB-INF/lib` folder of the D2 web application. For example,

```
C:/Program Files/Apache Software Foundation/Tomcat
6.0/webapps/D2/WEB-INF/lib
```

4. Start the web application server.
5. Verify the plug-in installation:
   a. Log in to D2 Client.
   b. Navigate to **Help** > **About D2** from the menu bar.
   c. Look for `YourCo-PluginName v1.0.0 (0001)` in the list of **Plugins**.

Configuring a Menu Item for the Template Plug-in

Configure a menu item using D2 Config to allow end users to call the custom download action.

1. Log in to D2 Config and navigate to **Go to** > **Menu D2** to open the D2 Client menu configuration page.
2. Add a new menu item to the menu in which you want the button to appear. The *EMC Documentum D2 Administration Guide* contains further instructions for configuring a D2 Client menu.
3. Fill out the form for the new menu item as described in the following table.

Field	**Description**
Label en	Export Custom Content
Shortcut	Ctrl+D
Action	Calling service's method
Service	D2CustomService
Method	getCustomDownloadURL
Selection	Select `MULTI` to allow end users to select multiple content for the download action.
Type	Select `EVENT` to post an event if the web service call is successful.
Action	D2_ACTION_EXPORT_FROM_URL

4. Click **Save**.

Creating a New Plug-in Using the Template

You must declare and insert the necessary resources using the following folder tree for D2 to recognize the custom plug-in:

```
<company name>-<plug-in name>
|--src
    |--com
        |--<your namespace>
            |--<plug-in name>
                |--> D2PluginVersion.java
                |--> <plug-in name>Version.java
                |--> <plug-in name>Version.properties
```

1. Rename the `YourCo-PluginName` project.

   For example: `DaveCo-CoolPlugin`
2. Rename `com.yourdomainhere.yourplugin` in Eclipse to your domain namespace and package name for each package in the plug-in template.

   For example: `com.daveco.coolplugin`
3. Optionally, rename `PluginNameVersion.java` and `PluginNameVersion.properties`

   For example: `CoolPluginVersion.java` and `CoolPluginVersion.properties`

   If you rename the files, edit `build.properties` to reflect the name change.
4. Edit `build.properties` and change `project.version.file` to reflect your package structure and the plug-in properties file name.

   For example: `/src/com/daveco/coolplugin/CoolPluginVersion.properties`
5. Remove the example plug-in classes and replace them with your plug-in implementation.

   For example: `*.webfs.services.content` and `*.webfs.services.create`
6. Re-run `.\build\build.xml` to rebuild your plug-in.

Configuring a Custom Action

1. Create a Java class in the *<project folder>*`/src/com/`*<your namespace>*`/`*<plug-in name>*`/webfs/services/custom/` folder.
2. Import the following resources:

   ```
   import java.util.List;
   import com.emc.d2fs.dctm.plugin.IPluginAction;
   import com.emc.d2fs.dctm.web.services.D2fsContext;
   import com.emc.d2fs.models.attribute.Attribute;
   ```

   The *EMC Documentum D2 D2FS API JavaDoc* contains more information about the packages that you can use to configure a custom action.
3. Implement `IPluginAction`:

   ```
   public class <name of the class> implements IPluginAction
   ```
4. The class must contain a method that has at minimum the following format:

   ```
   public List<attribute> <methodName>(D2fsContext context);
   ```

   For example, the YourCo-PluginName plug-in sample contains the following method declaration for `D2CustomService.java`:

   ```
   public List<Attribute> getCustomDownloadURL(D2fsContext context)
   throws UnsupportedEncodingException, DfException, D2fsException
   ```

5. After building and installing the plug-in, create a menu button in D2 Config to allow end users to perform the custom action. The *EMC Documentum D2 Administration Guide* contains further instructions for configuring a D2 Client menu.

   For example, Configuring a Menu Item for the Sample Plug-in, page 26 contains the instructions for creating a menu button for the YourCo-PluginName plug-in sample.

Understanding Service Interface Overrides

You can override all D2 services interfaces using the `@override` annotation.

Pre-Processing

You can add data processing before a service by adding code before the super call. The following example code adds a custom timestamp to the end of a content title before the service creates content properties. The pre-processing ensures that content created using the plug-in have a timestamp at the end of the title:

```
/** Override createProperties to add custom pre-processing.
  *
  */
 @Override
 public String createProperties(Context context,
     java.util.List<Attribute> parameters) throws Exception {
  // pre-processing -- look for ’title’ attribute and add a custom
         //timestamp at the end.
  for(Attribute attr : parameters){
   if (attr.getName().equals("title")){
    String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").
         format(Calendar.getInstance().getTime());
    attr.setValue(attr.getValue() + " : " + timeStamp);
    break;
   }
  }
  return super.createProperties(context, parameters);
 };
```

Post-Processing

You can add data processing after a service by adding code after the super call. The following example code runs the result of the service through a capitalization process. The post-processing ensures that the plug-in returns the `object_name` attribute in upper-case letters.

```
/** Override getContent() to demonstrate post processing
 *    (make object_name attributes upper case).
 *
 * NOTE: If overriding getContent() you may want to similarly override
 *       getFilteredContent().
 *
 */
 @Override
 public DocItems getContent(Context context, String contentId,
     String contentTypeName, String viewMode, String checkChildren)
         throws Exception {

     LOGGER.debug("D2ContentServicePlugin calling super.getContent()...");

     // Call base class implementation to get the default results.
     DocItems result = super.getContent(context, contentId, contentTypeName,
         viewMode, checkChildren);

     LOGGER.debug("D2ContentServicePlugin post processing items...");

     // Return post processed results
     return postProcessItems(result);
 }
```

Overriding the Process

You can perform a replacement override by forcing the service to return a different value than the original super method. The following example overrides the original `exportTo` class and returns `dh = new DataHandler(new FileDataSource(new File(myFile)))`.

```
    @InjectSession
    public FileContent exportTo(Context context, String parentId, String typeName,
        String colType, String exportType, String fileName) throws Exception
    {
        FileContent result = null;

        D2fsContext d2fsContext = (D2fsContext) context;
        ParameterParser parameterParser = d2fsContext.getParameterParser();

        LOGGER.info ("D2ExportServicePlugin Before exportTo");

        // If the parameter custom_call is sent and is true,
        //    the customization will be used.
        if (parameterParser.getBooleanParameter("custom_call"))
        {
            // overridden
            LOGGER.info ("custom_call, overridden...");

            List<String> ids = StringUtil.split(parameterParser.
                getStringParameter("id"), "-");
            LOGGER.info ("SELECTED IDs = " + ids);

            //Retrieve user session
            IDfSession  session = d2fsContext.getSession();

            //Prepare file content for result
            result = new FileContent();
            result.setName("MyFile.pdf");
            result.setMime(IDfFormatEx.PDF);
            result.setFormat(IDfFormatEx.PDF);

            //SETUP Here!!!: This file must exist on your server
            //(with sufficient permissions).
            String myFile = "c:\\temp\\MyFile.pdf";
            DataHandler dh = new DataHandler(new FileDataSource(new File(myFile)));
            result.setFileContent(dh);
        }
        else {
            //not overridden; proceed with default behavior.

            LOGGER.info ("(not overridden) calling super.exportTo()...");

            result = super.exportTo (context, parentId, typeName,
                colType, exportType, fileName);
        }

        return result;
    }
```

Throwing Exceptions

You can use `Exception()` and `D2fsException()` to report exceptions or errors encountered in your Service extension or Custom Action code back to D2 and the end user. The following example shows how to use `D2fsException()`:

```
    public List<Attribute> getCustomDownloadURL(D2fsContext context) throws
        UnsupportedEncodingException, DfException, D2fsException
    {
        LOGGER.debug("getCustomDownloadUrl()...");

        /**
        *  Error reporting -- Note that when you need to report an error to D2
        *    and the user, you can simply throw an Exception() or D2fsException(),
        *    whichever is compatible with the api your are overriding
        *    For example, this custom action throws D2fsException()
        *
        */
        Boolean bNeedToReportAnError = false;
        //
        if (bNeedToReportAnError) {
            // Error detected; Build your localized message and throw the
            //Exception() to notify user.
            throw new D2fsException("A problem was encountered with your
                custom export...");
         }

        // Code omitted for example clarity

         return result;
     }
```

Setting Up D2 Service Overrides

Use the following folder structure when creating a class to override D2 services:

```
<company name>-<plug-in name>
|--src
    |--com
        |--<your namespace>
            |--<plug-in name>
                |--webfs
                    |--services
                        |--<name of the service package>
                            |--> <name of the service>Plugin.java
```

1. Determine the *<name of the service>*.

   The *com.emc.d2fs.interfaces* section of the *EMC Documentum D2 D2FS API JavaDoc* contains a list of D2FS interfaces prefixed by the letter `I`. For example, the sample `D2ExportServicePlugin` class is based on the `IExportService` interface.

   To determine the *<name of the service>*, replace the prefix `I` with `D2`. For example, in the sample the interface `IExportService` uses the D2 service `D2ExportService`.

2. Create a .java file named *<name of the service>*`Plugin.java` for the services class to configure the process in the *<company name>-<plug-in name>*`/src/com/`*<your namespace>*`/`*<plug-in name>*`/webfs/services/`*<name of the service package>*`/` folder.
3. Create and name the class *<name of the service>*`Plugin`.

   For example, the sample uses the service `D2ExportService` to name its class `D2ExportServicePlugin`.

4. Extend the service class with the *<name of the service>* and implement the `ID2fsPlugin` interface.

   For example, the sample declares the class:

   ```
   public class D2ExportServicePlugin extends D2ExportService
   implements ID2fsPlugin
   ```

   The `ID2fsPlugin` interface forces the implementation of two methods:

   - `getFullName`: Returns the full plug-in name with its version number. The method is called by the **About** dialog box in D2.
   - `getProductName`: Returns the plug-in name to determine whether the plug-in should be executed based on the D2 configuration matrix.

Deploying D2 Plug-ins

1. Create the plug-in .jar file.
2. Copy the plug-in .jar file to the *<install path of D2>*`/WEB-INF/lib/` folder. For example:

   `C:\apache-tomcat-`*<version>*`\webapps\D2\WEB-INF\lib`

3. Restart the web application server.
4. Verify the plug-in installation by logging in to D2 Client and navigating to **Help** > **About**.