Composer Guide for Flex Development

Version 9 Release 7
<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before using this information and the product that it supports, read the information in “Notices and trademarks” on page 69.</td>
</tr>
</tbody>
</table>
Introduction

IBM® Initiate Master Data Service® Composer is a unified development environment for quickly building lightweight but robust applications to display, access, and manipulate the data managed by the IBM Initiate Master Data Service.

Each application is an assembly of components configured within the Composer interface and assembled within a Dojo development environment (for JavaScript implementations) or within Adobe Flash Builder (for Flex implementations). For example, a developer might quickly assemble components into an application that allows a user to search for an entity, see a sorted list of results, and view the relationships to other entities for any item in the list.

Composer is an Eclipse plug-in included by default within the IBM Initiate® Workbench environment. Developers creating Dojo-based applications, can use the Eclipse-based Rational® Application Developer or an IDE of their choice. For developers creating Flex-based applications, by installing the Flash Builder Plugin into the Workbench Eclipse environment, you can create, configure, and test applications within a single development environment.

Features

IBM Initiate Master Data Service Composer offers components and blueprints for quickly assembling applications, while also offering the ability to customize the appearance and behavior of applications to suit your particular needs.

Composer offers:

- **Component Library** — A full suite of reusable components are provided to enable common use cases of MDM technology, such as adding, editing, searching, relationship management, and duplicate prevention.
- **Solution Blueprints** — Solution blueprints are included that combine components together into end-to-end applications.
- **Configuration** — Composer components and blueprints can be customized without any coding. For example, users can select the fields to be displayed on a search form, or custom icons to be displayed on a Relationship Viewer component.

Key benefits

IBM Initiate Master Data Service Composer helps both developers and users to achieve greater efficiency without sacrificing quality or transparency.

In particular, Composer applications provide:

- **Fast time to value** — Build solutions quickly with reusable components and solution blueprints.
- **Reduce implementation costs** — Enable your developers to customize UI components instead of building interfaces and APIs from scratch.
- **Increase capabilities of existing applications** — Improve your existing applications by integrating robust search and data management functions.
- **Reduced architectural complexity** — Work with any data model so the functionality and user interfaces can be reused across business processes.
Extensibility — Swiftly customize the behavior and appearance of your applications.

Summary steps for creating an application

Creating an application using IBM Initiate Master Data Service Composer requires administrators to configure the Composer and Adobe Flash Builder environment, to generate and customize application blueprints, and to deploy the application to a Tomcat, WebSphere®, or WebLogic Application Server.

The following steps are discussed in detail within this Composer documentation:

- Download and install the Flash Builder plug-in.
- Download and install Apache Tomcat.
- Configure Master Data Engine connection parameters.

Within Composer and Flash Builder:

- Generate the default Composer configuration.
- Configure the settings for blueprints and generate them.
- Optionally configure component and blueprint settings within Flash Builder.
- Test the application locally on your machine.
- Debug your application within the Flash Debug perspective.
- Deploy your application to a J2EE Tomcat or WebSphere environment.

Architecture

IBM Initiate Master Data Service Composer is included by default as an Eclipse plug-in within the IBM Initiate Workbench environment. It is recommended that you install Flash Builder into Workbench as well so that all necessary tools reside in a single desktop environment.
As part of the deployment process, you deploy the Composer services to the application server to manage communication between your applications and the IBM Initiate Master Data Service. At run time, the application you deploy is available to users via the browser.

Composer provides generic components that read in configuration settings at run time. By separating the configuration settings from the component code itself, Composer allows administrators to make adjustments quickly and account for changes like new attributes within the underlying data model -- without having to recreate applications from the ground up. The advantages of this dynamic model extend beyond day-to-day development to include the upgrade process: Administrators can take advantage of new features in subsequent versions of Composer without configuring the new components from scratch.

Within the Composer configuration file (composer.icc), administrators can specify data settings such as which fields to display within an application, how to label components, and how to map attributes in the underlying data model to the fields that display in applications. Additional configuration information for the application comes from MXML context files, which indicate components the application includes, how those components are wired together, and other
application-level settings. By combining data configuration information with context settings for the application, Composer generates the application against its library of generic components.

A REST service layer governs how the resulting application communicates with the IBM Initiate Master Data Service.

Configuration for your applications is a combination of context configuration (which establishes the wiring between the Composer components) and data configuration (which indicates the MDM data you want to be available to the application). The application reads the data configuration at run time to populate the components. To make queries to the Master Data Engine, the application uses REST service calls.

The Composer Library contains the components such as Search Form, Search Results, and Details, which serve as the raw materials for applications. It does not include screens, contexts, and other infrastructure elements used for wiring components together.

**Initiate Composer components**

Within your IBM Initiate Master Data Service Composer applications, Composer components interact with the IBM Initiate Master Data Service to add member records, search for entities, display relationships, and so on.
Each Composer component represents a complete piece of functionality that manages a specific task such as searching or editing. Before exploring interconnected components, become familiar with the individual components and their configuration options by browsing the "Component reference" on page 81.

Because the Composer components are delivered within a Flex SWC library, the component code is not exposed. Instead you manipulate the behavior of components by setting parameter values within context files. With the Composer blueprints, context files govern the behavior for screens, tabs, or entire applications. For example, within the Multi-Domain Entity Search blueprint, the TabbedEntitySearchScreenContext gathers together three components:

```xml
<fx:Declarations>
  ...
  <components:SearchFormPM searchType="{SearchEvent.SEARCH_TYPE_ENTITY}"/>
  <components:SearchResultsPM/>
  <components:AddRecordFormPM addType="{AddEvent.ADD_TYPE_RECORD}"
    groupAttributes="false"/>
</fx:Declarations>
```

Here, searchType, addType, and groupAttributes are parameters that control the behavior of the components. The parameters are not set on the components directly but on PMs. PMs are presentation models. Each component has a corresponding PM that manages its configuration options and the flow of events for the component. For more information, see the "Component reference" on page 81.

### Initiate Composer blueprints

The IBM Initiate Master Data Service Composer blueprints gather together individual Composer components into MDM applications, which you can quickly configure and customize to meet your particular needs.

- **Quick View** — Combines the Search Form, Search Results, and Details components into a basic application for searching and viewing. The Quick View blueprint is offered primarily as a sample application. The QuickViewBlueprint.mxml and QuickViewBlueprintContext.mxml files each contain extensive documentation that explains the Parsley infrastructure, event handling, and more.
- **Multi-Domain Entity Search** — Combines the Search Form, Search Results, Add Form, Entity Details, Record Details, and Record List components into a production-grade application ready to be customized and deployed to enable users to search, compare, edit, and add records and entities. The Entity Details view also includes the Relationship Viewer and Related Entities components.

### Accessing the ActionScript API documentation (ASDoc)

IBM Initiate Master Data Service Composer includes HTML-based documentation for commands, components, and so on using ASDoc.

#### About this task

After you have extracted the composerdocs.zip file as described below, users can view the ASDoc documentation using a browser.

#### Procedure

1. Navigate in the file system to the `...untime\doc\` directory within the Composer plug-in in your IBM Initiate Workbench installation. For example:
2. Copy the composerdocs.zip file.
3. Extract composerdocs.zip to a location of your choosing.
4. In the new folder, navigate to the asdoc directory and open index.html in
   browser.
5. To view configuration options (Public Properties) for the Composer
   components, click to open the com.initiate.composer.components package.

---

**System requirements**

In general, the system requirements for IBM Initiate Master Data Service Composer are the same as those for the IBM Initiate Workbench.

By default Composer is embedded within Workbench, and so all Composer system requirements for operating system, memory, and so on are identical to the requirements for Workbench. Refer to IBM Initiate Master Data Service System Requirements for detailed information about system requirements for Workbench, including supported web application servers and their versions.

With this release, Composer is supported on the IBM WebSphere Application Server version 7.0, Apache Tomcat 6 application server, and the Oracle WebLogic 11g application server. If you use WebSphere 7.0, you must use IBM JDK 1.6 (SR8); a bug within earlier versions of the IBM JDK interferes with certain Composer functionality.

For developers creating Dojo-based applications, Composer uses JS/Dojo version 1.5. This documentation specifically describes using the Rational Application Developer version 8.0. With the Rational Application Developer, you can quickly deploy applications to WebSphere for testing or production.

For developers creating Flex-based applications, Composer requires users to install Adobe Flash Builder version 4. See the Flash Builder documentation for specific system requirements.
Chapter 1. Getting started

A sequence of topics intended to get you up and running with IBM Initiate Master Data Service Composer.

About this task

This section describes all steps required for configuring Composer, installing the Adobe Flash Builder plug-in, and enabling Composer components, blueprints, and services for use within Flash Builder.

Once these steps are complete, you will be ready to quickly build your own MDM applications.

Opening Initiate Composer

To open IBM Initiate Master Data Service Composer open the IBM Initiate Workbench and choose from the Composer options in the main menu bar.

About this task

By default, Workbench includes Composer.

Procedure

1. Start the IBM Initiate Workbench.
2. In the top menu, choose Composer.
3. Verify that the list of options includes Create Default Composer Configuration and Generate Composer Blueprints.
4. If you have not already done so, connect Workbench to an Engine. If necessary, create a new Initiate project and import a hub configuration. After you create the new project, you may see an error ("Must define at least 1 member type."). The error should resolve when you import the hub configuration. For full instructions, see the IBM Initiate Workbench User’s Guide.

Installing the Adobe Flash Builder plug-in

Installing the Adobe Flash Builder Plug-in within the IBM Initiate Workbench environment allows for a single Eclipse development environment.

About this task

The Adobe Flash Builder 4 download page includes downloads both for Flash Builder 4 and for Flash Builder 4 Eclipse Plug-in. Be sure to download the Flash Builder 4 Eclipse Plug-in, unless you want to develop Flex applications within an independent Eclipse environment. In that case, download and install Flash Builder 4. The IBM Initiate Master Data Service Composer documentation assumes that you have configured Workbench and Flash Builder within a single Eclipse environment.

Note: Adobe permits users to download free trials of Flash Builder 4 Eclipse Plug-in, effective for 60 days. Ultimately, be sure to use a fully licensed version of Flash Builder 4.
Downloading and installing Flash Builder 4

Flash Builder 4 is available for download from adobe.com.

About this task

Formerly Adobe Flex Builder, Adobe Flash Builder is the Eclipse-based IDE that allows IBM Initiate Master Data Service Composer developers to configure and test Composer applications.

To download and install Flash Builder 4:

Procedure

1. Navigate to the Adobe home page [adobe.com] and search for “Flash Builder.”
2. Choose Adobe Flash Builder > Try now and on the page that appears, from the download menu, choose English | Eclipse Plug-in Windows.
3. Click Download now.
4. When prompted, create an Adobe account or sign in.
5. When the download finishes, run the installer.
6. Accept the terms of the license agreement and the default installation directory.
7. Choose to Plug into another copy of Eclipse.
8. Select the Workbench installation folder. By default, the path is C:\Program Files\IBM\Initiate\WorkbenchX.X.0.
9. Click OK to close the browsing window and then click Next.
10. Choose to complete the installation.

Configuring the Flash Builder Plug-in as a product extension

By adding the Flash Builder Plug-in as an extension location, you can ensure that the full set of Flash Builder tools is available within IBM Initiate Workbench.

Procedure

1. Restart Workbench and navigate in the main menu to Window > Preferences > General > Capabilities.
2. In the list of capabilities, check the box for Classic Update and click OK.
3. Navigate in the main menu to Help > Software Updates > Manage Configuration. You might notice that the menu lists Software Updates twice. Choose the one that expands to display Manage Configuration.
4. In the window that appears, choose Add an Extension Location in the right pane.
5. Browse to the eclipse folder within the Flash Builder 4 Plug-in installation directory. By default, the path is: C:\Program Files\Adobe\Adobe Flash Builder 4 Plug-in\eclipse\
6. Click OK.
7. Restart Workbench as recommended.
8. Once Workbench has restarted, navigate to Window > Open Perspective > Other to verify that Flash, Flash Debug, and Flash Profile appear.
Installing the Apache Tomcat JDK to support Initiate Composer

Even if you will deploy your Composer application to a remote Apache Tomcat or IBM WebSphere Application Server, install Tomcat on the same machine where you install IBM Initiate Master Data Service Composer in order to enable the Composer development environment.

About this task

If you do not already have a local installation of Tomcat, download and install it as described in the subsequent topics.

Downloading the Java JDK

Tomcat requires that you have a compatible JDK (Java Development Kit) installed.

About this task

On Windows, to verify that the JDK is installed, open a command-line window and type java -version. If the JDK is installed, you see something like:

```
java version "1.6.0_17"
Java(TM) SE Runtime Environment (build 1.6.0_17-b04)
Java HotSpot(TM) Client VM (build 14.3-b01, mixed mode, sharing)
```

If you get the message java is not recognized as an internal or external command, the JDK is not installed. Download and configure it.

To download the Java JDK:

**Procedure**

2. Choose to download the Java Platform (JDK).
3. Make sure that the Platform is set to Windows and click Download.
4. When prompted to Log In for Download, choose to Skip this Step.
5. Choose to save the JDK installer.

Configuring the Java SDK

Once downloaded, set the paths for the Java SDK and verify that it is installed correctly.

**Procedure**

1. Run the JDK executable you have downloaded and accept the defaults presented by the installer.
3. Click Environment Variables.
4. For System variables, choose New, and set the new variable name to JAVA_HOME and the value to the path of the new JDK installation, for example: C:\Program Files\Java\jdk1.6.0_17
5. Click OK.
6. Still in System variables, choose to Edit the PATH variable.
7. At the end of the string, append a semicolon and include a pointer to the bin subdirectory of JAVA_HOME. You can specify either the new JAVA_HOME variable
followed by bin (for example, %JAVA_HOME%/bin) or the full path (for example, C:/Program Files/Java/jdk1.6.0_17/bin).

8. To test the JDK, open a command-line window and type java -version.

Results

The command returns something like this:

```
java version "1.6.0_17"
Java(TM) SE Runtime Environment (build 1.6.0_17-b04)
Java HotSpot(TM) Client VM (build 14.3-b01, mixed mode, sharing)
```

Installing Apache Tomcat

Even if you deploy your IBM Initiate Master Data Service Composer application to a remote Apache Tomcat, IBM WebSphere, or WebLogic Application Server, you must install Tomcat on the same machine where you install Composer in order to enable the Composer development environment.

About this task

If you do not already have a local installation of Tomcat, download and install it as described below.

Procedure

2. In the left column, click the link to download Apache 6.x. Refer to the IBM Initiate Master Data Service System Requirements for details on supported versions. The steps below use version 6.0.24.
3. From among the Binary Distributions, choose the version appropriate to your operating system.
4. Extract the file to C:/. Doing so creates the directory C:/apache-tomcat-6.0.xx.
6. Click Environment Variables.
7. For System variables, choose New, and set the new variable name to CATALINA_HOME and the value to the path of the new Tomcat installation. For example, C:/apache-tomcat-6.0.24
8. Click OK.
9. In a command-line window, navigate to C:/apache-tomcat-6.0.24/bin and type startup.bat. Tomcat starts and prints status messages.

Deploying the composer.war file on Tomcat

Deploying the composer.war file enables the IBM Initiate Master Data Service Composer services within Tomcat.

Procedure

1. Stop Tomcat if it is running. To do so, navigate at a command-line prompt to the bin/ directory within your Tomcat installation and type shutdown.bat.
2. Navigate in the file system to the ...untime\bin\ directory within the Composer plug-in in your Workbench installation. For example:
Copy the composer.war file.
4. Paste composer.war within the Tomcat webapps directory (or into a subdirectory of webapps). For example, paste the file within:
   C:\apache-tomcat-6.0.24\webapps\ 
5. Ensure that the IBM Initiate engine is running.
6. Restart Tomcat by navigating at a command-line prompt to the bin/ directory within your Tomcat installation and type startup.bat.
7. Navigate in your browser to verify that Tomcat is running again as expected. By default, the Tomcat home page is available at: http://hostname:8080/
8. To verify that the composer.war file has been deployed correctly, navigate to http://hostname:8080/composer. Depending on your configuration, the page either displays a login box or the default index.jsp with instructions to configure the redirect to your Flex wrapper. If you have specified a context other than composer, substitute your context in the URL as appropriate.

Setting Master Data Engine connection parameters for the local Tomcat installation

To enable communication with your local installation of Tomcat, configure connection parameters within catalina.properties and composersvcs.properties.

About this task

The following instructions apply to your local installation of Tomcat. See Chapter 6, "Deploying applications," on page 71.

In order for the IBM Initiate Master Data Service Composer applications that you generate to connect with the Master Data Engine, the application server needs the appropriate connection information. As described below, validate the settings within catalina.properties and create a composersvcs.properties file within a shared/classes/ directory on Tomcat.

Note: This topic does not discuss user authentication. For information about authentication, see "Managing authentication for Initiate Composer applications" on page 37.

Procedure

1. Ensure that the IBM Initiate engine is running.
2. Navigate in the file system to your Tomcat installation conf directory.
3. Open catalina.properties in a text editor.
4. Verify that the entry beginning with shared.loader= reads:
   shared.loader=${catalina.base}/shared/classes,${catalina.base}/shared/lib/*.*.jar
5. Save and close catalina.properties.
6. Within the Tomcat installation directory, create a directory named shared, for example: C:\apache-tomcat-6.0.24\shared
7. Within the new shared directory, create a classes directory.
8. Navigate in the file system to the ... \runtime\bin\ directory within the Composer plug-in in your Workbench installation. For example:
9. Copy the composervcs.properties.example file.
10. Paste composervcs.properties.example into the ..\shared\classes directory you created.
11. Rename composervcs.properties.example to composervcs.properties.
12. Open the composervcs.properties file.
13. Uncomment the following lines by removing the initial pound signs (#) and set the values to reflect a valid host name and port for your installation:

   ```
   contextFactory.hostName=
   contextFactory.hostPort=
   
   For example, if your configuration runs the Master Data Engine at localhost:16000, set the values as:
   contextFactory.hostName=localhost
   contextFactory.hostPort=16000
   ```
14. Save and close the composervcs.properties file.
15. Restart Tomcat.

**Configuring Flex settings for your Initiate project**

Enable Adobe Flash Builder by setting various parameters for the compiler, SDK, and more.

**Configuring Flash Builder settings within the Workbench**

To continue integrating Flash Builder into the Workbench installation, set values for the web root, root URL for the IBM Initiate Master Data Service Composer WAR file, and for the output folder.

**Procedure**

1. Within IBM Initiate Workbench, right-click your Initiate project and choose **Add/Change Project Type > Add Flex Project Type**.
2. Within the dialog window, for **Server technology**, set **Application server type** to **J2EE**.
3. Clear the checkmark for **Use remote object access service**.
4. Click **Next**.
5. For **Web root**, browse to and select the folder created when Tomcat unpacked the composer.war file. For example, C:\apache-tomcat-6.0.24\webapps\composer.
6. For **Root URL**, specify the URL where you deploy Composer applications. For example, http://hostname:8080/composer. The directory must be the parent directory of the WEB-INF folder.
7. Click to **Validate Configuration**. Look for a message indicating that **The web root folder and root URL are valid**.
8. For **Compiled Flex application location**, choose an **Output folder**. By default, the folder is the **projectname-debug** folder within the webapps directory or subdirectory. For example:

   ```
   C:\apache-tomcat-6.0.24\webapps\composer\projectname-debug
   ```
9. Click **Finish**.
10. When prompted to **Switch to the Flash perspective**, click **Yes**.
Setting the Flex compiler version

For IBM Initiate Master Data Service Composer components to function correctly, they must compile against the Flex 4.1 compiler.

Procedure
1. In the Flash Navigator, right-click the project and choose Properties.
2. Choose Flex Compiler.
3. In the section Flex SDK version, confirm that the project is using the Flex 4.1 SDK.
4. Click Apply.
5. Click OK to close the Properties window.

Creating the default Initiate Composer configuration

Once you have installed IBM Initiate Master Data Service Composer, the Flash Builder plug-in, and Tomcat, return to IBM Initiate Workbench to generate the default Composer configuration.

About this task

The process opens the Composer Configuration Editor and creates the `projectname/composer/config/composer.icc` configuration file, visible in the Workbench Navigator view.

As described in subsequent topics, you edit the `composer.icc` file using the Composer Configuration Editor to choose and configure the blueprints you want to serve as the basis of your applications.

Procedure
1. From the Composer menu, click Create Default Composer Configuration.
2. Select a project from the list and click OK. The Composer Configuration Editor opens.
3. Verify that the `composer.icc` file appears within `projectname/composer/config/` in the left-pane.

Configuring blueprints

IBM Initiate Master Data Service Composer includes blueprints included to demonstrate how Initiate Composer components can be assembled into applications.

About this task

Using blueprints is optional, though much of the information in the topics that follow applies even if you assemble components into applications from scratch. With the Configuration Editor for IBM Initiate Master Data Service Composer within the IBM Initiate Workbench, configure the names, labels, attribute sets, and composite views for your blueprints. Each blueprint you configure appears on a separate tab within the application you generate.

Configuring the Quick View blueprint

In order to configure blueprints, you specify settings in the Composer Configuration Editor, such as the name, label, member type, and attribute set.
About this task

The settings are saved within the composer.icc configuration file, which IBM Initiate Master Data Service Composer reads at run time.

In general, IBM Initiate Master Data Service Composer components parallel standard components in Flex (buttons, forms, and so on). By contrast blueprints are pre-wired arrangements of Composer components designed to enable common sets of tasks. The Quick View blueprint combines three different Composer components: Search Form, Search Results, and Details. Blueprints indicate the flow of events from one component to another. They can be used as-is within an application, or the relevant code can be copied and used within a larger application.

As part of the configuration process, you can choose the Search Type for the blueprint, either Entity or Record. Doing so configures the blueprint components as entity search form, results, and details versus record search form, results, and details. For details about the individual components including a description of available configuration options, see “Component reference” on page 81.

Procedure

1. If it is not already open, launch the Composer Configuration Editor by double-clicking the composer.icc file within the projectname/composer/config/directory in the left pane.
2. On the Blueprints view, click the green plus-sign. The right pane populates with a list of available blueprints.
3. Select Quick View and click Add. The blueprint appears in the list within the left pane of the Editor, and the right pane populates with configuration options.
4. For Name specify a value that reflects the role the blueprint plays in your application.
5. Set the Label for the blueprint. The label appears as the name of the blueprint in the UI tab. Using non-alphanumeric characters for Label causes an error within Flash Builder.
6. Package refers to the Java package generated for the blueprint. Only change the package name if you must have more than one instance of the same blueprint. The package name must be unique across your deployment environment.
7. Choose a Member Type from the drop-down menu. The values here are drawn from your underlying data model.
8. Choose a Search As option from the drop-down menu. Composer uses this entity type when conducting a search. As with Member Type, the available values are drawn from your data model.
9. Choose a Search Type option from the drop-down menu. Your choice determines whether the blueprint components are configured for entities or records.
10. Select an Attribute Set for each component. An attribute set is a container that administrators can create and configure in order to specify the set of attributes that display within a Composer component. Because an attribute set is required for each component, if no attribute sets appear in the drop-down menus for the components, you must configure at least one attribute set before you can save the blueprint configuration. For instructions about creating attribute sets, see “Creating and configuring an attribute set” on page 15.
Be sure to return to the **Blueprints** view to assign the attribute sets you create. The drop-down menus in the **Blueprints** view only offer attribute sets that match the member type you have chosen for the blueprint.

When you finish, save your changes. Changes are saved to the `composer.icc` file. Continue with the blueprint configuration using the **Model** view and **Attribute Sets** view.

### Configuring the Multi-Domain Entity Search blueprint

For the Multi-Domain Entity Search blueprint, configure a label, source, and attribute set for each entity type that the application manages.

#### About this task

At run time IBM Initiate Master Data Service Composer reads the `composer.icc` configuration file.

In general, Composer components parallel standard components in Flex (buttons, forms, and so on); by contrast blueprints are pre-wired arrangements of Composer components designed to enable common sets of tasks. The **Multi-Domain Entity Search** blueprint combines the following Composer components: **Search Form**, **Search Results**, **Add Form**, **Entity Details**, **Record Details**, and **Record List**. Embedded within the **Entity Details** view, the blueprint also includes the **Related Entities** component and **Relationship Viewer** component.

Blueprints indicate the flow of events from one component to another. They can be used as-is within an application or the relevant code can be copied and used from within a larger application. For details about the individual components including a description of available configuration options, see "Component reference" on page 81.

#### Procedure

1. If it is not already open, launch the Composer Configuration Editor by double-clicking the `composer.icc` within the `/projectname/composer/config/` directory in the Navigator.

2. On the **Blueprints** view, click the green plus-sign. The right pane populates with a list of available blueprints.

3. Select **Multi-Domain Entity Search** and click **Add**. The blueprint appears in the list within the left pane of the Editor, and the right pane populates with configuration options for the particular blueprint. Blueprints function as containers for components.

4. For **Name** specify a value that reflects the role the blueprint plays in your application.

5. Set the **Label** for the blueprint. The label appears as the name of the blueprint in the UI tab. Using non-alphanumeric characters for **Label** causes an error within Flash Builder.

6. **Package** refers to the Java package generated for the blueprint. Only change the package name if you must have more than one instance of the same blueprint. The package name must be unique across your deployment environment.

7. For each **Entity Type** in the drop-down menu, indicate whether it should be **Enabled**. Enabled entity types are available to the blueprint, meaning that the blueprint's components can search against the entity type, display details for the entity type, and so on. For the Multi-Domain Entity Search blueprint, each
entity type you enable appears as a tab within the application. The Relationship Viewer component and Related Entities component display relationships with entity types that are not enabled, but your users cannot view the details for those entity types. The following steps assume that the entity type is enabled.

8. Set the **Label** for the blueprint. The label appears as the name of the blueprint in the UI tab.

9. For **Add/Edit Source**, choose from the drop-down menu. The application created from this blueprint includes components for adding and editing records and entities. Choosing the source indicates to which repository the application writes the additions and changes. Typically it is best to choose a Hub-controlled source. A Hub-controlled source is one maintained by the IBM Initiate Master Data Service itself in order to aggregate additions and changes in one easily accessible location. Be certain to specify a source for every entity type that you have enabled. See "Configuring a source" and "Configuring authentication for editing entities and adding members" on page 38.

10. Select an **Attribute Set** for each component. You might need to expand the pane to see the full list of components. As explained in later sections of this guide, an attribute set is a container that administrators can create and configure in order to specify the set of attributes that display within a Composer component. Because an attribute set is required for each component, if no attribute sets appear in the drop-down menus for the components, you must configure at least one attribute set before you can save the blueprint configuration. For instructions about creating attribute sets, see “Creating and configuring an attribute set” on page 15. Be sure to return to the Blueprints view to assign the attribute sets you create. The drop-down menus in the Blueprints view only offer attribute sets that match the member type you have chosen for the blueprint.

11. When you finish, save your changes. Changes are saved to the composer.icc file. Continue with the blueprint configuration using the **Model** view and **Attribute Sets** view.

**Configuring a source**

For IBM Initiate Master Data Service Composer blueprints whose components add records or edit entities, you can create a dedicated Hub-controlled source and specify that source when configuring your blueprints.

**About this task**

Creating a dedicated Hub-controlled source allows Composer applications to maintain added records and edited entities in one easily accessible location. Once you have created the source, specify it as the **Add/Edit Source** within the Blueprints view of the Composer Configuration Editor for any blueprints that enable adding records or editing entities. Note that the Quick View blueprint does not include any components that add records or perform entity editing and so the **Add/Edit Source** drop-down does not appear.

Once the new Hub-controlled source is created and configured, when a user adds a record within the application, the application writes the new record to that source. Specifically, the application creates an entity which contains the single record you have just created. (Remember that entities are simply containers for one or more records.)
The application behavior for editing an existing entity is somewhat more complex. Suppose that an existing entity consists of three records maintained by three different sources. When a user edits the entity within your application, the changes are written to the record that is maintained in the source you specified as the Add/Edit Source. If you indicated a new Hub-controlled source (and if the entity has not previously been edited within your application), the application creates a record in the Hub-controlled source and links the record to the other three records in the other sources. If the entity has already been edited within your application, the application writes the updates into the existing record in the Hub-controlled source and links the record to the other sources. The record maintained by the Hub-controlled source typically becomes the most current record within an entity, unless edits are also made to records within other sources.

How the application links the record to the other records depends on your configuration. By default, the application "hard links" the records, meaning that the application issues a command that binds together the associated records for the entity using an identity rule. The identity rule indicates that two or more members have been assigned the same Member ID. (In the API, this is known as an entity rule, specifically EntRule.) This linking occurs each time a user edits an entity. As an alternative, you can configure the application to "soft link" the records so that when the user edits an entity, rather than linking the member records as occurs with hard linking, Composer reads the values from the other member records that comprise the entity — and writes those values into the member record within the selected source, where they appear as inactive values. See “Choosing how to link records when making edits to entities” on page 42.

Procedure

1. Within the projectname directory in the Navigator, double-click projectname.imm file.
2. In the window that opens in the right pane, click Member Types in the vertical list of options.
3. From the Member type drop-down menu at the top of the pane, choose the member type for which you want to create a Hub-controlled source.
4. Click to open the Sources tab.
5. Click the Add button. A new row appears in the table of sources.
6. Edit the fields in the table to assign a Source Name and Source Code. For example, assign a source name of Composer Add and Edit and a source code of COMPSR.
7. In the right column, ensure that all attributes are enabled for the selected source.
8. Repeat the process for all member types for which users can add new records or edit entities.
9. Save your changes. The new source is available within the Add/Edit Source drop-down menu on the Blueprints view of the Composer Configuration Editor.

Changing the default locale

Strictly speaking, the current release of IBM Initiate Master Data Service Composer supports only the English (United States) locale. However, users can rename the en_US folder to reflect their own locale and edit the composer.properties file to localize some of the application interface labels.
About this task

Renaming the en_US folder does not alter the Locale drop-down menu available within the Model view and Attribute Sets view in the Composer Configuration Editor. For the current release, the drop-down only displays English (United States) regardless of changes to the en_US folder or composer.properties file.

Procedure
1. If it is not already started, open the IBM Initiate Workbench.
2. In the left pane of the Workbench, navigate to the projectname/composer/config/en_US folder.
3. Right-click the en_US folder and choose Rename.
4. On the Rename Resource window, type a name that reflects your locale, for example en_GB or fr_CA.
5. Click OK.

   Note: For non-English locales, you can edit composer.properties to localize some of the interface labels. To do so, continue with the following steps.
6. Within the new folder, double-click composer.properties. The file opens in the right pane.
7. Edit the values to reflect your locale. Any Composer applications you create display the new values.
8. Save your changes.

Configuring the type label
You can assign a user-readable label to the database names of member types and entity types using the Model view, which includes a Type Label column.

Procedure
1. If it is not already open, launch the IBM Initiate Master Data Service Composer Configuration Editor by double-clicking the composer.icc within the projectname/composer/config/ directory in the Navigator.
2. Navigate to the Model view.
3. Under the Type Label table heading click within the cell that corresponds to the member or entity you want to configure.
4. If a label is already present, you can edit it. If no label is present, you can add one.
5. Repeat as necessary to configure the entities and members.
6. Save your changes.

Configuring the composite view
You can indicate the composite view for a member type or entity type using the Model view, which includes a Composite View column.

About this task

For more information about composite views, see “The Composite View column” on page 22.
Note: If you have recently created new composite views, they might not immediately appear in the Composer Configuration Editor. If necessary, reopen the IBM Initiate Master Data Service Composer Configuration Editor.

Procedure
1. If it is not already open, launch the Composer Configuration Editor by double-clicking the composer.icc file within the projectname/composer/config/ directory in the Navigator.
2. Navigate to the Model view.
3. Under the Composite View table heading click within the cell that corresponds to the member or entity you want to configure.
4. Click the down arrow that appears at the right of the cell.
5. Choose from the composite views available.
6. Repeat as necessary to configure the entities and members.
7. Save your changes.

Assigning a default attribute set
Using the Model view, which includes a Default Attribute Set column, you can indicate which pre-configured set of attributes you want to display for the Related Entities component. Despite the name, the default attribute set does not function as a global setting.

About this task
To create an attribute set, use the Attribute Set view within the Composer Configuration Editor. See “Creating and configuring an attribute set” on page 15. For background information about attributes sets, see “Attribute Sets view” on page 23.

Procedure
1. If it is not already open, launch the Composer Configuration Editor by double-clicking the composer.icc file within the projectname/composer/config/ directory in the Navigator.
2. Navigate to the Model view.
3. Within the Member/Entity type table, click within the cell for the member or entity whose default attribute set you want to assign.
4. Click the down arrow that appears at the right of the cell and choose an attribute set. If no attribute sets have been configured, create one or more before proceeding. For instructions about creating attribute sets, see “Attribute Sets view” on page 23. Be sure to return to the Model view to assign the attribute sets you create.
5. When you finish, save your changes. Changes are saved to the composer.icc file.

Assigning icons for display on the Relationships tab
Using the Model view, which includes an Icon column, you can assign a small icon to represent a member type or entity type within your application. The icons you assign appear within the Relationships view for the Multi-Domain Entity Search blueprint.
About this task

The Relationships tab graphically displays a network of relationships for a selected member or entity.

Be sure to choose an icon of the appropriate size. The default icons included with IBM Initiate Master Data Service Composer are 48 x 48 pixels. For best results, use an image no larger than 64 x 64 pixels.

Procedure
1. If it is not already open, launch the Composer Configuration Editor by double-clicking the composer.icc within the projectname/composer/config/ directory in the Navigator.
2. Navigate to the Model view.
3. Under the Icon table heading click within the cell that corresponds to the member or entity you want to configure.
4. Click the ellipsis button at the right of the cell.
5. In the Open window that appears, navigate to the icon you want to use.
6. Click Open.
7. Repeat as necessary to configure the other entities and members.
8. Save your changes.

Configuring an instance label pattern

Use the Model view to set a label for instances of members and entities when they appear on interface tabs and within the Relationship Viewer component.

About this task

Instance label patterns allow you to combine attribute fields for display. Consider the display of a person's name. Rather than displaying title, first name, middle name, last name, and suffix all in separate fields, you can combine them into a single display, configured as you wish with the last name appearing first, the first name appearing first, or whatever suits your particular needs. The instance label pattern governs how fields are displayed on interface tabs as well as within the Relationship Viewer. A related tool called an attribute pattern allows you to combine fields for display within the Record List, Search Results, or Details components. See “Configuring an attribute pattern” on page 16.

Procedure
1. If it is not already open, launch the IBM Initiate Master Data Service Composer Configuration Editor by double-clicking the composer.icc file within the projectname/composer/config/ directory in the Navigator.
2. Navigate to the Model view.
3. In the Member/Entity type table, highlight the row for the member or entity whose label you want to configure. The display for Instance Label Pattern for... updates to reflect the member or entity you choose.
4. Choose an attribute from the Attribute drop-down menu.
5. Click the Insert Field button and choose from the list of available fields for the attribute.
6. Repeat until you have added all the fields you want to combine.
7. Cut and paste the field tags to arrange them as you want them to display.
8. Optionally, place additional text between the field tags. For example, to format an instance label using a **Name** attribute so that it appears as *Last Name, First Name*, insert a comma and space to configure the attribute pattern as `<lastname_attribute>, <firstname_attribute>`.

9. Optionally, you can configure a tag that indicates text to display only if the value for an attribute is present. For example, the record for a particular person might not include a middle name. If you are displaying names as *Last Name, First Name, Middle Name*, you can use an optional tag to display the comma and space after *First Name* only if the value for middle name is present. To do so, click the **Insert Optional Tag** button and arrange the tags as follows: `<field>lastname_attribute</field>, <field>firstname_attribute</field><optional>, <field>middlename_attribute</field><optional>.

10. Save your changes.

### Creating and configuring an attribute set

Create and configure an attribute set to indicate the specific attributes that you want to display within a IBM Initiate Master Data Service Composer component — or for a member type or entity type across all Composer components.

**Procedure**

1. If it is not already open, launch the Composer Configuration Editor by double-clicking the `composer.icc` file within the `projectname/composer/config/` directory in the Navigator.
2. Navigate to the **Attribute Sets** view.
3. Click the green plus-sign. A new attribute set appears in the first available table row.
4. In the **Name** column, click the new attribute set and assign a name that reflects how you intend to use the attribute set, for example `EntitySearchFormAttributes` or `EntitySearchDetailsAttributes`. Attribute set names must be unique across the application; two different member types cannot use an attribute set with the same name.
5. In the **Member Type** column, choose a member type from the drop-down list. The fields available in the list depend on the model for your underlying data. You can create multiple attribute sets for a single member type. As mentioned in the example, you might choose to use one attribute set for one component and a different attribute set for another component.
6. Choose which attributes and fields you want to include in the attribute set by moving those attributes from the **Currently Hidden** list to the **Currently Displayed** list.

**Note:** Attributes whose attribute type is **MEMATTR** include an **elemdesc** field. The **elemdesc** field is a read-only field whose value is assigned by the IBM Initiate Master Data Service Engine for attributes configured as Enumerated Data Types. The field is controlled exclusively by the Engine. To avoid confusion, do not include any **elemdesc** fields when configuring attribute sets.

7. Optionally, configure the **Label** setting for the attributes in the **Currently Displayed** list. The labels you configure appear as labels for the corresponding field within any component that uses this attribute set.

8. The **Attribute Pattern** column allows you to configure several fields to display for a single label. See "Configuring an attribute pattern" on page 16.

9. When you finish, save your changes. Changes are saved to the `composer.icc` file.
Related tasks

“Cloning an attribute set”
Rather than creating an attribute set from scratch, you can clone an existing attribute set and make any required edits.

Cloning an attribute set
Rather than creating an attribute set from scratch, you can clone an existing attribute set and make any required edits.

About this task
Depending on your data model, creating and configuring an attribute set may be a lengthy process. Rather than requiring you to repeat that process from beginning to end with each attribute set, the Composer Configuration Editor offers the ability to select and clone an existing attribute set.

Procedure
1. If it is not already open, launch the Composer Configuration Editor by double-clicking the composer.icc file within the projectname/composer/config/ directory in the Navigator.
2. Navigate to the Attribute Sets view.
3. Select a row in the Attribute Sets table and click the third icon in the column at the far right. (The icon shows two pages overlaid with a green arrow.) A new Copy of ... row appears in the table.
4. Edit the new attribute set as needed.
5. When you finish, save your changes. Changes are saved to the composer.icc file.

Configuring an attribute pattern
You can configure an attribute pattern in order to display several fields together as a unit.

About this task
Rather than each field in your data model displaying individually within the Record List, Search Results, or Details components, attribute patterns allow you to combine fields for display. Consider the display of a person’s name. Rather than displaying title, first name, middle name, last name, and suffix all in separate fields, you can combine them into a single display, configured as you wish with last name appearing first, first name appearing first, or whatever suits your particular needs. In order for the Record List or Search Results components to attend to the attribute patterns you create, set the showFields to false for the appropriate PM within the context file for the blueprint. See “Component reference” on page 81. By default, the showFields parameter is set to true for the Record List and Search Results components. The Details component uses attribute patterns within its display by default. A related tool called an instance label pattern allows you to combine fields for display on interface tabs and within the Relationship Viewer component. See “Configuring an instance label pattern” on page 14.
Procedure

1. If it is not already open, launch the IBM Initiate Master Data Service Composer Configuration Editor by double-clicking the composer.icc within the projectname/composer/config/ directory in the Navigator.
2. Navigate to the Attribute Sets view.
3. In the Attributes and Fields table, highlight a row that represents several attributes. For example, an attribute such as Legal Name might contain fields such as Title, First Name, Middle Name, and Last Name.
4. Within the Attribute Pattern column, click the ellipsis button at the right of the field. The Attribute Pattern dialog box window opens.
5. Click the Insert Field button and choose from the list of available fields for the attribute.
6. Repeat until you have added all the fields you want to combine.
7. Cut and paste the field tags to arrange them as you want them to display.
8. Optionally, place additional text between the field tags. For example, to format an instance label using a Name attribute so that it appears as Last Name, First Name, insert a comma and space to configure the attribute pattern as <lastname_attribute>, <firstname_attribute>.
9. Optionally, you can configure a tag that indicates text to display only if the value for an attribute is present. For example, the record for a particular person might not include a middle name. If you are displaying names as Last Name, First Name, Middle Name, you can use an optional tag to display the comma and space after First Name only if the value for middle name is present. To do so, click the Insert Optional Tag button and arrange the tags as follows: <field>lastname_attribute</field>, <field>firstname_attribute</field><optional>, <field>middleattribute</field></optional>.
10. Click OK to close the Attribute Pattern window.
11. Save your changes.

Displaying the score for search results

You can configure the Search Results component to include the matching score for each result displayed.

About this task

The Search Results component displays search results in descending order according to a matching score. Configure the component to display the score by editing the context file for the application.

By default, the Search Results component within the Quick View blueprint displays the score for search results.

Procedure

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to projectname/composer/src/blueprints/blueprintname/
2. Open the TabbedEntitySearchScreenContext.mxml context file for editing,
3. Locate the line that begins <components:SearchResultsPM ...
4. Place your cursor after the "PM" in SearchResultsPM and hit the space bar. A drop-down menu allows you to insert a parameter for the PM file.
5. Choose the showScore parameter and set the value to true. (The default setting is false.)

Grouping fields for display on a search form

You can configure the Search Form component to group together fields according to their attribute container.

About this task

As defined by the attribute set, individual fields belong to attributes. For example, the individual fields `stLine1`, `city`, `state`, and `zipCode` might belong to the Home Address attribute. By default, the Search Form component does not display the attribute name as a way of grouping the underlying fields. To change the default behavior, edit the instance of the Search Form component within context file for the application. The ordering of the field names in the application interface reflects the ordering of the field names within the attribute set.

See “Creating and configuring an attribute set” on page 15.

Procedure

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to `projectname/composer/src/blueprints/blueprintname/`
2. Open the TabbedEntitySearchScreenContext.mxml context file for editing. (If you are configuring a Quick View blueprint, edit the QuickViewBlueprintContext.mxml file rather than the TabbedEntitySearchScreenContext.mxml file.)
3. Locate the line that begins `<components:SearchFormPM ... >`
4. Place your cursor after the "PM" in SearchFormPM and hit the space bar. A drop-down menu allows you to insert a parameter for the PM file.
5. Choose the `groupAttributes` parameter and set the value to true. (The default setting is false.)
6. Save and close the context file.

Generating Initiate Composer blueprints

You can generate blueprint application code for a specified project.

About this task

To generate the IBM Initiate Master Data Service Composer blueprint application code, the generation process reads configuration information from the composer.icc configuration file, whose contents you set with the Composer Configuration Editor described in “Configuring blueprints” on page 7. If you configure and generate multiple blueprints, each one appears as a separate tab within DemoMainApplication.

After you have generated the blueprint source code as described below, you can further customize them within Flash Builder to adjust colors, fonts, layout, event processing, and so on.

Procedure

1. Within Workbench, from the Composer menu, click Generate Composer Blueprints.
2. In the window that appears, choose a project from the drop-down menu.
By default the following options are selected:

- **Clean the Composer client directory.** This option indicates that, before generating, the process deletes the contents of the client directory which contains UI component and blueprint code. Doing so is recommended when regenerating the source files and is helpful if you encounter errors and must start over.

- **Generate Composer Blueprint(s) source code.** This option indicates that the process generates the Flex UI code for the blueprint.

3. Click **Finish.**

The generation process creates the `libs/` folder and the `src/` folder within the `projectname/composer/` directory. The `libs/` folder contains the libraries that generate the code for components, commands, and services. The `src/` folder contains the `assets/` folder and `blueprints/` folder, as well as the `DemoMainApplication.mxml` and `DemoMainApplicationContext.mxml` files. The `blueprints/` folder includes the class and context files for the blueprint.

### Setting the source and library build paths

To enable Flash Builder to build your IBM Initiate Master Data Service Composer application, specify the source and library paths to point to the required files.

**Procedure**

1. In the Flash Navigator, right-click the project and choose **Properties.**
2. Choose **Flex Build Path** from the left pane.
3. If it is not already active, switch to the **Source path** tab.
4. For **Main source folder**, choose **Browse.**
5. Navigate to the `projectname/composer/src` directory.
6. Click **OK.**
7. Choose **Add Folder.**
8. On the dialog window that appears, choose **Browse.**
9. Navigate to the `projectname/composer/config` directory. For example, 
   ```
   C:\Documents and Settings\username\workspace\projectname\composer\config
   ```
10. Click **OK.**
11. Choose **Add Folder** again.
12. On the dialog window that appears, specify the locale directory for the project, but instead of your actual locale (for example, `en_US` or `fr_CA`) use `{locale}`. Doing so allows the locale to update dynamically. For example, specify `composer\config\{locale}`.
13. Click **OK.**
14. Switch to the **Library path** tab.
15. Choose **Add SWC Folder.**
16. On the dialog that appears, choose **Browse.**
17. Navigate to the `libs` directory within the Composer plug-in. For example:
   ```
   C:\Program Files\IBM\Initiate\WorkbenchX.X.0\plugins\com.initiatesystems.workbench.composer_X.X.0\runtime\lib\com.initiatesystems.workbench.composer_X.X.0\runtime\lib\
   ```
18. Click **OK** to exit the browse window.
19. Click **OK** to confirm the folder path.
20. Click **OK** to close the Properties window. Workbench updates the compiler settings.
Setting the default Initiate Composer application

By specifying DemoMainApplication.mxml as the default application, Flash Builder executes against that application when you choose the run and debug functions.

Procedure
1. In the Flash Navigator, right-click the project and choose Properties.
2. Choose Flex Applications.
3. Click Add.
4. In the Add Application Files window, choose DemoMainApplication.mxml and click OK.
5. Select DemoMainApplication.mxml and click the Set as Default button and OK. The Properties window closes as Workbench updates the compiler settings.

Testing an application in the Flash perspective

Before deploying your IBM Initiate Master Data Service Composer application, run it as a web application from within the Flash perspective of the IBM Initiate Workbench.

Procedure
1. Within the Flash perspective, double-click DemoMainApplication.mxml within the projectname/composer/src/(default package) directory.
2. From the main menu, choose Run > Run, or click the green run button.
3. If a Run As window appears, choose Web Application and click OK.
4. If a Save and Launch window appears, accept the defaults and click OK. Flash Builder launches a browser and displays the login panel for the application.
5. Enter your user name and password credentials for the Master Data Engine. The defaults are system and system. For details about managing login, see "Managing authentication for Initiate Composer applications" on page 37.

Results

If you have completed all of the steps above, you can now to use the application to search the data available within the Hub configuration you have imported.

Now that you have tested the application locally in the Flash perspective, you are ready to make the application available to users. See Chapter 6, “Deploying applications,” on page 71.
Chapter 2. Initiate Composer components and blueprints overview

A component is a GUI-based element that performs a function within a Flash application. A component can be as simple as a button or as sophisticated as a viewer for displaying complex relationships between entities. By extension, a IBM Initiate Master Data Service. A Composer blueprint is an assembly of several components wired together.

Blueprints manage the events flowing from one component to another, and are intended to answer certain common business needs.

The topics that follow describe the steps for configuring and extending individual Composer components and for configuring Composer blueprints, including steps for managing and customizing the login panel that appears by default for Composer applications.

Configuration Editor

Changes made within the three IBM Initiate Master Data Service Composer Configuration Editor views are written to the composer.icc file and to the locale-specific composer.properties file. The views are Blueprints, Model, and Attribute Sets.

Note that when you generate blueprints, the Composer Configuration Editor writes a second copy of the composer.icc file into the projectname/composer/WebContent/config directory. If you make additional changes to the composer.icc file within projectname/composer/config, Initiate Composer automatically propagates the changes to the copy of composer.icc file within projectname/composer/WebContent/config. However, any changes you make to the composer.icc file within projectname/composer/WebContent/config are not propagated to the composer.icc file within projectname/composer/config. In fact, any edits you make to the version of composer.icc within projectname/composer/WebContent/config will be overwritten by changes made (using the Composer Configuration Editor) to the version within projectname/composer/config.

The Blueprints view

The Blueprints view allows you to specify and configure the blueprints you generate as the basis of your IBM Initiate Master Data Service Composer applications.

The Blueprints view is the default view when you open the Composer Configuration Editor. Use it to add blueprints, name them, and specify their labels as well as to specify the source database and attribute sets the blueprints use.

The Model view

The Model view within the IBM Initiate Master Data Service Composer Configuration Editor allows administrators to configure how Master Data Engine data appears within Initiate Composer components.

The Model view allows administrators to:
• Configure the type label for members and entities. See “Configuring the type label” on page 12.

• Configure a composite view for a member or entity. See “The Composite View column” and “Configuring the composite view” on page 12.

• Assign a default attribute set for a member or entity. See “Assigning a default attribute set” on page 13.

• Set an icon for a member or entity. See “Assigning icons for display on the Relationships tab” on page 13.

• Configure an instance label pattern. See “Configuring an instance label pattern” on page 14.

The Composite View column
Using the Composite View column in the table on the Model, you can configure which value to display when multiple values are available for an individual member or entity attribute.

Consider the Composite View column only if you have created one or more composite views within your project's member model. By default, Composer components use the EMCA (Entity Most Current® Attribute) composite view for the display of entity information, and the MMCA (Member Most Current Attribute) composite view for the display of member information.

A composite view determines which value to display when multiple values are available for an individual member or entity attribute. For example, one of the member records associated with an entity may indicate that the person lives in Chicago while another indicates that the same person lives in Phoenix. Adhering to the default EMCA composite view for the entity, the Composer component displays whichever entity attribute value for city was most recently created or modified, regardless of which member record contains the value.

The same mechanism applies to the display of member attribute values. Adhering to the default MMCA composite view for the member, the Composer component displays whichever attribute value was most recently created or modified, again regardless of which member record contains the value.

By contrast, configuring your own composite view for the project alters the mechanism that determines what to display. For example, configuring a Trusted Source composite view causes Composer components to display whichever value for an attribute comes from the member record associated with the single trusted source — even if that value is not the most recent.

It is also possible to designate a Controlled composite view that displays a selection of attributes from multiple specified sources. For example, you can configure the composite view to display name and address information from Source A, phone and birth date from Source B, and Social Security number and address from Source C.

For complete details about creating and configuring composite views, see the IBM Initiate Workbench User’s Guide.

For the steps to configure a composite view in the Composer Configuration Editor, see “Configuring the composite view” on page 12.
**Attribute Sets view**

Using the **Attribute Sets** view within the IBM Initiate Master Data Service Composer Configuration Editor, you can specify the set of attributes that display within Composer components.

Attribute sets offer administrators a convenient way to reuse sets of attributes across multiple components rather than configuring the same individual attributes each time they create a component.

You must create and assign at least one attribute set.

For example, for a **Search Form** component, suppose you want to allow the user to search by first name, last name, zip code, or Social Security number. Also suppose that you would like the **Search Details** component to display not only the attributes just mentioned but also gender, street address, and telephone numbers. You might therefore create and configure two attribute sets, one called **SearchFormAttributes** to indicate the fields to display within the **Search Form** component, and a second called **SearchDetailsAttributes** to indicate the fields to display within the **Search Details** component.

By contrast, you might decide that all components should always display the same attributes regardless of the component, you can set the attribute set for the member type or entity type within the **Default Attribute Set** column of the Model view. The setting serves as a global setting for across components. See "Assigning a default attribute set" on page 13.
Chapter 3. Customizing your application

Customizing your application includes a broad range of tasks including how to manipulate styles, how to manage user authentication, how to configure communication between the application and JavaScript, and more.

Configuring your application interface

Configure your application interface to determine the appearance of labels, EDTs, icons, and CSS-governed aspects such as color and font size.

Customizing applications using CSS

By supplying your own CSS file, you can change the appearance of your application, including the background color, font size, and more.

About this task

Flex includes a global style sheet (defaults.css) inside the framework.swc file in the /frameworks/libs directory of the SDK. (For example, C:\Program Files\Adobe\Adobe Flash Builder 4 Plug-in\sdk\4.1.0\frameworks\libs.) Flex implicitly loads the defaults.css file and applies it to your Composer applications during compilation.

In addition to the global style sheet, a second defaults.css delivered within the ComposerLib.swc library applies styles to the Composer applications.

To override the settings in those CSS files, create a CSS file of your own. Once your new CSS file is available in the Flex build path, its settings are applied to your Composer applications during compilation. The settings from the custom CSS file you create override the settings within the ComposerLib defaults.css and the settings within the Flex SDK defaults.css. (In turn, the settings within the ComposerLib defaults.css override the settings within the Flex SDK defaults.css.) Any styles not specified by your own CSS file or by the ComposerLib defaults.css are determined by the settings within the Flex SDK defaults.css.

By default, at compilation time, Flex looks for a file named defaults.css in the Flex build path. For this reason, if possible, name your custom CSS file defaults.css. If you must name the CSS file something other than defaults.css or if you need to keep the CSS file in a location on the file system outside of the Flex build path, ignore the steps below and instead follow the steps in the topic "Using a custom CSS file whose name is not defaults.css" on page 28.

Note: As an alternative to creating and configuring a new CSS file, you can edit the appearance of the application using the Design View within Flash Builder. For example, if you generate a Multi-Domain Entity Search blueprint, the generation process creates a collection of .mxml files for the screens and tabs. Select the screen or tab you want to configure, choose Design from the right-pane toolbar in Flash Builder, and edit the style settings. Your changes are maintained directly in the .mxml file you are editing.
Procedure

1. In Workbench within the Flash Package Explorer, right-click `projectname` and choose Properties then Flex Build Path.
2. Choose the Source path tab and click the Add Folder button.
3. In the Add Folder window, specify `composer\src\assets\styles`.

   **Note:** In all likelihood, the `composer\src` folder is already designated as your Main source folder. If so, adding `composer\src\assets\styles` to your Flex build path is redundant, but does no harm.
4. Click OK to close the Add Folder window and OK to close the Properties window.
5. Right-click the `projectname/composer/src/assets/styles` directory and choose New > CSS File.
6. For File name, specify `defaults` and click Finish. The Workbench automatically appends the `.css` file extension and opens the file. In the navigation pane Workbench interposes a (default package) tier between the styles directory and the new `defaults.css` file. Despite that tier, the `defaults.css` file is located directly within the styles directory in the file system.
7. The new CSS file includes specifications for namespaces based on the location of the file. Flash Builder 4 requires that all CSS files specify namespaces. Edit the namespaces if you need the styles to apply beyond the current project.
8. Set any styles you want to customize for your application. To do so:
   a. In Workbench within the Flash Package Explorer, open the `DemoMainApplication.mxml` file.
   b. Locate a container or class whose style you want to configure.
   c. In your CSS file, begin typing the name of the container or class, and then type Ctrl+Space. The page displays a list of containers and classes that match the characters you have typed. Highlighting a selection populates a window with a description.
   d. Double-click to choose an item from the list.
   e. After the name, type a space and then an open curly-brace `{`.
   f. Type Ctrl+Space again. Workbench provides the closing curly-brace `}`).
   g. Type Ctrl+Space another time. The page displays a list of styles. Highlighting a selection populates a window with a description of the style including possible values.
   h. Double-click to choose a style from the list.
   i. Set the value for the style, for example, `color:#b0c4de;` or `font-family:"Times New Roman", Times, serif;`.
   j. Repeat to configure the styles you want to customize.
9. When you have finished, save and close `defaults.css`.
10. If you have deployed your application, redeploy it for the changes to take effect.

**CSS example: Re-skinning the AttributeInputField**
The example demonstrates how to use CSS to determine a Flex skin for a control regardless of where the control appears within your application.
About this task

By default, the appearance of a control within a component is hardcoded within the control. For example, the skinClass for AttributeInputField is hardcoded. By creating a new MXML skin and a reference within a CSS file, you can determine the appearance and behavior for the control across your application.

Procedure

1. Within your application source directory right-click and choose New > MXML Skin.
2. Leave blueprints as the Package and supply a unique Name.
4. Leave the Create as copy of option unchecked.
5. Click Finish.
6. Within the new .mxml file for the skin, set the following parts:
   - dateField
   - dropDownList
   - textInput
   For example:
   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
           xmlns:s="library://ns.adobe.com/flex/spark"
           xmlns:mx="library://ns.adobe.com/flex/mx">
       <!-- host component -->
       <fx:Metadata>
           [HostComponent("com.initiate.composer.controls.AttributeInputField")]
       </fx:Metadata>

       <!-- states -->
       <s:states>
           <s:State name="List" />
           <s:State name="Date" />
           <s:State name="String" />
       </s:states>

       <!-- SkinParts -->
       name=dateField, type=mx.controls.DateField,
           required=false
       name=dropDownList, type=mx.controls.ComboBox,
           required=false
       name=textInput, type=spark.components.TextInput,
           required=false

       <!-- SkinParts -->
       <mx:DateField id="dateField" includeIn="Date" yearNaNavigationEnabled="true" />
       <s:TextInput id="textInput" includeIn="String" />
       <mx:ComboBox id="dropDownList" includeIn="List" />
   </s:Skin>
   
   Note: In the code above for SkinParts, required is set to false for each part because no one setting is used every time. Setting required=false and selecting a default (text) in the component itself indicates to the compiler that the setting is determined at runtime.

7. For the sake of the example, the code surrounds each field in a horizontal box with a label indicating that it is a customized skin:
8. Within the ..\assets\styles\ directory, create defaults.css. (The example assumes that you have not previously defined a style sheet.)

9. Within defaults.css, define the default style to use for all occurrences of AttributeInputFields within the application.

```plaintext
@namespace mx "library://ns.adobe.com/flex/mx";
@namespace s "library://ns.adobe.com/flex/spark";
@namespace code "http://code.google.com/p/flexlib/";
@namespace spark "spark.skins.spark.*";
@namespace controls "com.initiate.composer.controls.*";

controls|AttributeInputField
{
    skinClass: ClassReference("blueprints.CustomButtonSkin");
}
```

10. Instruct the compiler to use your new style sheet. First right-click your project and select Properties > Flex Compiler.

11. For Additional compiler arguments use the -defaults-css-url to specify the relative path to the new CSS file. For example:

```
-locale en_US -defaults-css-url assets/styles/defaults.css
```

12. Click OK.

13. Run the application to verify the style change.

**Using a custom CSS file whose name is not defaults.css**

Additional steps are required to configure a custom CSS file that is not named defaults.css.

**About this task**

By default, at compilation time, Flex looks for a file named defaults.css in the Flex build path. For this reason, it is ideal to create a file named defaults.css which you maintain in the Flex build path. To do so, ignore the steps below and follow the steps in the topic "Customizing applications using CSS" on page 25. However, if you must name the CSS file something other than defaults.css or if you need to keep the CSS file in a location on the file system outside of the Flex build path, follow the steps below.

**Note:** As an alternative to creating and configuring a new CSS file, you can edit the appearance of the application using the Design View within Flash Builder. For example, if you generate a Multi-Domain Entity Search blueprint, the generation process creates a collection of .mxml files for the screens and tabs. Select the screen or tab you want to configure, choose Design from the right-pane toolbar in Flash Builder, and edit the style settings. Your changes are maintained in the .mxml file you are editing.
Procedure
1. Navigate to the `projectname/composer/src/assets/styles/directory`.
2. Right-click the directory and choose New > CSS File.

   **Note:** If you need to maintain your CSS file outside of the `projectname` directory structure, for example if the CSS file must be shared by multiple applications, you must configure a security domain to bypass the default Flex security settings. Contact your system administrator.

3. For File name, specify the file name for your .css file. Do not include the .css extension. The Workbench automatically appends the .css file extension to the name you provide, and opens the file. In the navigation pane Workbench interposes a (default package) tier between the styles directory and the new CSS file.

4. The new CSS file includes specifications for namespaces based on the location of the file. Flash Builder 4 requires that all CSS files specify namespaces. Edit the namespaces if you need the styles to apply beyond the current project.

5. Set any styles you want to customize for your application:
   a. In Workbench within the Flash Package Explorer, open the `DemoMainApplication.mxml` file.
   b. Locate a container or class whose style you want to configure.
   c. In your CSS file, begin typing the name of the container or class, and then type Ctrl+Space. The page displays a list of containers and classes that match the characters you have typed. Highlighting a selection populates a window with a description.
   d. Double-click to choose an item from the list.
   e. After the name, type a space and then an open curly-brace ({).
   f. Type Ctrl+Space again. Workbench provides the closing curly-brace (}).
   g. Type Ctrl+Space another time. The page displays a list of styles. Highlighting a selection populates a window with a description of the style including possible values.
   h. Double-click to choose a style from the list.
   i. Set the value for the style, for example, color:#b0c4de; or font-family:"Times New Roman", Times, serif;
   j. Repeat to configure the styles you want to customize.

6. When you have finished, save and close the CSS file.
7. In Workbench within the Flash Package Explorer, right-click `projectname` and choose Properties then Flex Build Path.
8. Choose the Source path tab and click the Add Folder button.
9. In the Add Folder window, specify the folder where you created the CSS file. For example, `composer\src\assets\styles`.

   **Note:** In all likelihood, the `composer\src` folder is already designated as your Main source folder. If so, adding `composer\src\assets\styles` to your Flex build path is redundant, but does no harm.
10. Click OK to close the Add Folder window and OK to close the Properties window.
11. Once again, right-click `projectname` and choose Properties then Flex Compiler.
12. In the Additional compiler arguments text box specify `--defaults-css-url` and the path to your CSS file name. For example, the whole argument line...
might read: -locale en_US -defaults-css-url src/assets/styles/
filename.css. If the CSS file is not maintained within the
projectname/composer/src/assets/styles/ directory, specify a relative path to
the file. In such a case, the whole argument line might read: -locale en_US
-graphic css-url ../../../common/styles/healthapps.css.

13. Click OK to close the Properties window.
14. If you have deployed your application, redeploy it for the changes to take
effect.

**Configuring EDTs (enumerated data types)**

Enumerated Data Types (EDTs) are lists of possible values (true/false, fifty states,
marital status) for a particular attribute. To configure an EDT, first create the list of
values and then associate the list with a particular attribute, such as STATE or
MARITAL STATUS.

**About this task**

An EDT (enumerated data type) is one in which incoming data has a finite, known
standard expression. For example: Sex - M/F; marital status - married, single; or
state - AZ, CA, FL, and so on. EDT elements are the standard values that an EDT
recognizes. Composer applications use EDTs to provide users selection values
within drop-down lists.

Once you have created an EDT and associated it with a member attribute, your
Composer application looks for any field definitions with associated EDTs. It
automatically presents the drop-down list of values to the user wherever the
application allows the user to edit or choose a value for that attribute.

In particular, EDTs appear within:

- **SearchForm** component
- **FlexibleSearchForm** component (available for Dojo development only)
- **AddRecordForm** component
- **EntityDetailTable** component (if editing is enabled)
- **RecordDetailTable** component (if editing is enabled)

Note that for the **SearchForm** component, the EDT appears as a combo box,
meaning that users can choose one of the EDT values or type in a value of their
own. This allows users to search for values that may not be listed within the EDT,
but which may exist in the database. By contrast, the other components only allow
the user to choose from the EDT values offered. This limitation prevents unwanted
values from being written to the database.

**Related reference**

[“Component reference” on page 81](#)

Per-component topics provide an easy reference to the main IBM Initiate Master
Data Service Composer components including a short description, configuration
options, and a summary of events.

**Adding an EDT to your project**

If it is not already configured, create the enumerated data type that you want to
make available to the users of your IBM Initiate Master Data Service Composer
application.
Procedure
1. Within the `projectname/composer/config/` directory in the Navigator, double-click `projectname.imm` file.
2. In the window that opens in the right pane, click Enumerated Data Types in the vertical list of options.
3. Next to the Data type drop-down menu at the top of the pane, click the Add button. The Workbench adds a new data type in the form `EnumTypeX`, where X represents a unique number. You next add the possible values for the EDT you have created.
4. To the right of the Data type values table, click the other Add button for each value in your EDT. Workbench adds default values for each table row. The
5. Edit the Value and Description content to reflect the EDT you want to create. The setting for Description will appear in the drop-down menu in the application. For example, if you are creating an EDT for eye color, add rows and set the Description content to amber, blue, brown, gray, green, hazel, and other. Create the rows in the order you want the options to appear in the drop-down menu.
6. Repeat the process for all EDTs you want to create.
7. Save your changes. The new EDTs will be available for association to a member attribute.

Associating an EDT with a member attribute
Once you have created an EDT, associate it with a member attribute to make the values available within the applications you create.

Procedure
1. If it is not already open, open the `projectname.imm` file within the `projectname/composer/config/` directory in the Navigator.
2. In the window that opens in the right pane, click Member Types in the vertical list of options.
3. From the Member type drop down menu, choose the member type you want to configure.
4. On the Attributes tab, select an attribute whose type is MEMATTR or MEMIDENT. No other member types support EDTs.

Note: Technically speaking, EDTs can function only for attributes with a single field. For example, the MEMATTR attribute type has a single field: attrval. By contrast, MEMIDENT is composed of several fields, however with MEMIDENT the idissuer field is automatically hard-wired as the single field considered for EDTs.
5. In the lower left corner of the Workbench, select the Properties tab. A table shows the properties and values for the attribute you have selected.
6. Highlight the row for Enumerated data type code. Although this row appears for all attributes, as mentioned above, EDTs are only available for MEMATTR and MEMIDENT member types. For any other member type, any setting for Enumerated data type code is ignored.
7. In the cell in the Value column click the button at the right to see a drop-down list of EDTs.
8. Select an EDT and save your changes.
9. In the main menu, choose Initiate and then Deploy Hub Configuration.
10. Ensure that Core configuration is selected.
11. Click Finish. Your changes are saved to the Hub.
12. If you have deployed your application redeploy it for the changes to take effect.

Results

When your application initializes, field definitions with associated EDTs render as drop-down lists showing the values you configured.

Customizing the icons that appear on tabs

You can configure the tabs within your application to display custom icons representing member and entity records.

Procedure

Begin by creating an icon library to contain the images you want to display.

1. Create a Multi-domain Blueprint in Workbench. In the example, the blueprint is configured to support ID and HH searches. Configure your blueprint according to own data. See “Configuring the Multi-Domain Entity Search blueprint” on page 9.

2. Define the icons you want to use for each of your entities. See “Assigning icons for display on the Relationships tab” on page 13.

3. Generate the blueprint. See “Generating Initiate Composer blueprints” on page 18.

4. To create the icon library, first navigate in the Package Explorer to the images that you defined in the Composer configuration file.

5. Create a new ActionScript file that references each of the images that you want to use. For simplicity, the example creates an IconLibrary.as file in the source directory (default path):

   ```actionscript
   package {
   import flash.utils.Dictionary;

   [Bindable]
   public class IconLibrary{
      [Embed (source="assets/images/id_card2.png")]
      public static var ID_CARD2_PNG:Class;

      [Embed (source="assets/images/id.png")]
      public static var ID_PNG:Class;

      [Embed (source="assets/images/home.png")]
      public static var HOME_PNG:Class;

      [Embed (source="assets/images/cube_blue.png")]
      public static var CUBE_BLUE_PNG:Class;

      [Embed (source="assets/images/hh.png")]
      public static var HH_PNG:Class;

      [Embed (source="assets/images/record.png")]
      public static var DEFAULT_RECORD:Class;

      [Embed (source="assets/images/entity.png")]
      public static var DEFAULT_ENTITY:Class;

      private static var _isInitialized:Boolean;
      private static var _dictionary:Dictionary;

      public static function init():void {
```
if (_isInitialized) {
    return;
}
_dict = new Dictionary();
_dict["assets/images/id_card2.png"] = ID_CARD2_PNG;
_dict["assets/images/id.png"] = ID_PNG;
_dict["assets/images/home.png"] = HOME_PNG;
_dict["assets/images/cube_blue.png"] = CUBE_BLUE_PNG;
_dict["assets/images/hh.png"] = HH_PNG;
_isInitialized = true;
}

public static function getIconClass(iconPath:String, defaultIcon:Class):Class
{
    _init();
    var clz:Class = _dict[iconPath] != null ? _dict[iconPath] : defaultIcon;
    return clz;
}

6. Update TabbedEntitySearchScreen.mxml to refer to the library. For example, change
entityDetailTab.icon=entityImg; to
entityDetailTab.icon=IconLibrary.getIconClass(entConfig.iconPath, entityImg); and edit the handleRecordListRecordSelected function to specify
the icon library you created:
if ( recordDetailTab == null )
{
    recordDetailTab = new RecordDetailTab();
    recordDetailTab.record = record;
    recordDetailTab.detailsConfigName = recordDetailsConfigName;
    var memConfig:MemberTypeConfig = configService.getMemberTypeConfig(searchMemtype);
    recordDetailTab.label = memConfig.applyPattern(record);
    recordDetailTab.icon=IconLibrary.getIconClass(memConfig.iconPath, recordImg);
    tabNavigator.addChild(recordDetailTab);
    tabMap[getRecordKey(record)] = recordDetailTab;
}

7. Also within TabbedEntitySearchScreen.mxml, if necessary specify the height of
the tab to accommodate your icon:
    <flexlib:SuperTabNavigator id="tabNavigator" width="100%" height="100%" tabHeight="32">
8. Run the application.

**Editing the application title as it appears in the browser**

Edit a setting within the DemoMainApplication.mxml file to change the display of
the application name in the user’s browser tab and title bar.

**Procedure**

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to
   projectname/composer/src/(default package)/
2. Open the DemoMainApplication.mxml file for editing.
3. Find the following code:
   ```
   private function init():void {
     // To set the application title, change the value below.
     BrowserManager.getInstance().setTitle(resourceManager.getString('composerLib','ApplicationTitle'));
   }
   ```
4. Clear the default setting for setTitle:
   ```
   resourceManager.getString('composerLib','ApplicationTitle')
   ```
5. Specify your application title. Include the value within quotation marks, for example:
   ```
   setTitle("My Application Title");
   ```
7. If you have deployed your application, redeploy it for the changes to take effect.

### Editing the application masthead images

Edit the DemoMainApplication.mxml file to change the two images that display in the application masthead.

#### About this task

By default, the IBM Initiate Master Data Service Composer applications you create from the blueprints display IBM images in the application masthead. One is the IBM logo. The other is the product name: IBM Initiate Composer. The following steps describe replacing those two images with images relevant to your own application. For information about setting the spacing around the images you provide, see "Editing the application masthead image spacing."

#### Procedure

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to `projectname/composer/src/(default package)/`
2. Open the DemoMainApplication.mxml file for editing.
3. Find the `<controls:ComposerMasthead ... >` tag:
   ```
   <controls:ComposerMasthead leftBannerImage="assets/images/title_composer.png" rightBannerImage="assets/images/ibm-logo-black.png" ... />
   ```
   The `title_composer.png` image is 154 pixels wide by 15 pixels high. The `ibm-logo-black.png` image is 41 pixels wide by 15 pixels high.
4. Edit the code to point to your own image files.
5. Copy your images files into `projectname/composer/src/assets/images/`
6. Save and close DemoMainApplication.mxml.
7. If you have deployed your application, redeploy it for the changes to take effect.

### Editing the application masthead image spacing

Edit the DemoMainApplication.mxml file to change the spacing around the images that you display in the application masthead.
About this task

The following steps assume that you have already replaced both of the images in the application masthead according to the instructions “Editing the application masthead images” on page 34.

Depending on the dimensions of the images you used to replace the default images, you can change the positions of the images by editing the spacing around them.

Procedure

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to projectname/composer/src/(default package)/
2. Open the DemoMainApplication.mxml file for editing.
3. Find the <controls:ComposerMasthead ... > tag:
   <controls:ComposerMasthead leftBannerImage="assets/images/title_composer.png" rightBannerImage="assets/images/ibm-logo-black.gif"
   ... />
4. Within the <controls:ComposerMasthead ... > tag, add settings for leftImageSpacing and rightImageSpacing. For example:
   <controls:ComposerMasthead leftBannerImage="assets/images/title_composer.png" rightBannerImage="assets/images/ibm-logo-black.gif"
   leftImageSpacing="{myLeftImageSpacing}"
   rightImageSpacing="{myRightImageSpacing}" ...
   />
5. In order to specify the values for the spacing, add new objects to DemoMainApplication.mxml. You have two options:
   • Declare spacing values within a bindable object variable. For example:
     [Bindable]
     public var myLeftImageSpacing:Object = {top:2, bottom:2, right:10, left:10};
   • Declare a Flex object within the <fx:Declarations> tag. For example:
     <fx:Object id="myImageSpacing" top="3" right="3" bottom="12" left="12"/>
    Whichever method you choose, be certain to specify a value for all four parameters.
6. When you have finished, save and close DemoMainApplication.mxml.
7. If you have deployed your application, redeploy it for the changes to take effect.

Editing the links in the application header

Edit the DemoMainApplication.mxml file to add or remove links to the upper right navigation within your IBM Initiate Master Data Service Composer application.

About this task

By default, the Composer application you create from the blueprints display a Help link in the upper right of the application interface, which opens an alert that encourages the user to customize the links. To change that link or add others, edit the <fx:ArrayCollection ... > tag within DemoMainApplication.mxml.
**Procedure**

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to `projectname/composer/src/(default package)/`
2. Open the DemoMainApplication.mxml file for editing.
3. Find the `</s:ArrayCollection ...>` tag within the `<fx:Declarations>` tag:
   ```xml
   <s:ArrayCollection id="links">
      <fx:Object id="helpLink" label="Help" linkAction="HELP"/>
   </s:ArrayCollection>
   ```
4. Using the line `<fx:Object id="helpLink" label="Help" linkAction="HELP"/>` as a model, add `<fx:Object ...>` tags for any links you want to add. For example, you might add links for About and Contact:
   ```xml
   <s:ArrayCollection id="links">
      <fx:Object id="aboutLink" label="About" linkAction="ABOUT"/>
      <fx:Object id="contactLink" label="Contact" linkAction="CONTACT"/>
      <fx:Object id="helpLink" label="Help" linkAction="HELP"/>
   </s:ArrayCollection>
   ```
   The links appear in the upper right corner of the application arranged from left to right in the order you list them within the `<s:ArrayCollection ...>` tag.
5. Elsewhere within the DemoMainApplication.mxml file, find the myLinkActions function, which determines which actions result from clicking the links you define. For example:
   ```actionscript
   public function myLinkActions(event:ItemClickEvent):void
   {
      var item:Object = event.item;
      switch(item.linkAction)
      {
         case "ABOUT":
            // Clicking the link launches the MyAboutWindow popup component. Note that for a popup component to work, your DemoMainApplication.mxml file must import the necessary package. Within the CDATA import statements include:
            // "import mx.managers.PopUpManager;"
            var win:MyAboutWindow = new MyAboutWindow();
            PopUpManager.addPopUp(win, this, true);
            PopUpManager.centerPopUp(win);
            break;
         case "CONTACT":
            // Clicking the link opens the indicated URL.
            navigateToURL( new URLRequest("http://www.example.com/contact.html" ), "_self" );
            break;
         case "HELP":
            // Clicking the link displays a simple alert popup window with the indicated text.
            Alert.show("You can add any number of links to this menu and define the associated actions to perform", "Link Action Performed");
            break;
      }
   }
   ```
6. Save and close DemoMainApplication.mxml.
7. To open a popup component when the user clicks a link, as indicated for the About link above:
   a. In the Flash Package Explorer, right-click the `projectname` folder and choose **New > MXML Component**.
   b. In the New MXML Component window, assign a **Name**, for example, MyAboutWindow.mxml.
c. For Based on, type “title” into the text box.
d. From the list that appears double-click TitleWindow - spark.components. Doing so populates the text box with spark.components.TitleWindow.
e. Click the Finish button.
f. Edit the default code in the new file to correspond with the code you added to DemoMainApplication.mxml. For example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<s:TitleWindow xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark" close="close()"
    xmlns:mx="library://ns.adobe.com/flex/mx" width="400" height="300">
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import mx.managers.PopUpManager;
            public function close():void
            {
                PopUpManager.removePopUp(this);
            }
        ]]>  
    </fx:Script>
    <s:Button x="43" y="236" label="Button"/>
</s:TitleWindow>
```
g. Save your changes.
h. Still within Flash Builder, switch to the Design View by clicking the Design button.
i. Add any text, images, or other formatting to the component.
j. Save the new component.

8. If you have deployed your application, redeploy it for the changes to take effect.

### Managing authentication for Initiate Composer applications

By default, the IBM Initiate Master Data Service Composer applications derived from the included blueprints prompt users for user name and password credentials at startup using Flex-based authentication. You can configure your application to use JSP authentication, which is more secure, or turn off authentication altogether. In addition, you can set the session timeout and configure the image to display on the login panel.

#### About this task

To log in to your Composer application, users supply the same credential information they would use to log in to the Master Data Engine itself. The defaults for administrative privileges are system and system.

The default Flex-based authentication relies on settings in two locations:
- The beans-security.xml file within the WEB-INF directory for the application on the application server.
- The `<security:FormAuthenticationProvider/>` setting within the DemoMainApplicationContext.mxml context file.

When a user logs in, Composer sets a session cookie in memory that remains valid for as long as the user is actively using the application. By default the session ends when the user clicks Logout in the upper right within the application or after 20
minutes of inactivity. In either case the user must then log in again. At logout, Composer clears the data visible within the application components.

By default, Composer applications derived from the blueprints check for required authorization and disable or hide the features for users who are not authorized to use them. For example, users who are not permitted to write to the database do not see the Add Member button available to users with write permissions.

### Configuring authentication for editing entities and adding members

IBM Initiate Master Data Service Composer components that enable users to edit entities or add members require a higher level of authentication than is required for read-only functions.

#### About this task

Because Composer authentication uses the Master Data Engine authentication framework, configure authentication permissions for Composer using the Workbench User Management tool.

#### Procedure

1. Within IBM Initiate Workbench, navigate to Window > Open Perspective > Other > User Management.
2. On the User Management tab that opens, if you are not already connected to a Directory Server, click the icon in the upper right of the perspective to Connect to a Directory Server.
3. Create or edit a user.
4. In the Group Membership pane in the lower right, assign the user as appropriate. Users assigned to All Segments Read Write can add members and edit entities. Users assigned to All Segments Read Only cannot.

### Turning off the login panel for your application

To turn off the login panel, embed a set of default credentials in the DemoMainApplication.mxml file. In order to perform authentication and authorization for all operations, the application submits the credentials you provide.

#### About this task

The following instructions assume that you have configured Flex-based authentication. (By default, Initiate Composer uses JSP-based authentication.) See "Configuring Flex-based authentication" on page 39.

By default, the DemoMainApplication.mxml file includes the showLogin function which is called to display the login panel. You can comment out the view stack selectedChild assignment and instead provide a user name and password directly to the loginManager.

The following steps simply describe the simplest way of disabling the login panel. Depending on your configuration and security needs, you can obtain the credentials from a separate configuration file or configure single sign-on (SSO) using the Spring Framework.
Regardless of how you decide to manage authentication and authorization, consider making your application available only over HTTPS. Running the application over HTTP can allow credentials to be accessed in plain text by other computers on the network.

**Procedure**

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to `projectname/composer/src/(default package)/`
2. Open `DemoMainApplication.mxml` for editing.
3. Find the `showLogin` function:
   ```
   public function showLogin():void {
     // change the view stack to the login form
     appStack.selectedChild = loginView;
   }
   ```
4. Comment out `appStack.selectedChild = loginView;` and add code to submit credentials directly to the `loginManager`. For example, to submit user name `system` and password `system`, edit `showLogin` as follows:
   ```
   public function showLogin():void {
     // change the view stack to the login form
     //appStack.selectedChild = loginView;
     loginManager.login("system","system");
   }
   ```
5. Save and close the `DemoMainApplication.mxml` file.
6. Test the change by running the application within the Flash Development environment.

**Configuring Flex-based authentication**

As an alternative to the default JSP-based authentication, you can configure Flex-based authentication.

**About this task**

Flex-based applications send credentials to be validated and then when the credentials are approved, the appropriate view is displayed. Switching from the default JSP-based authentication to Flex-based authentication requires you to edit the security settings in multiple files. The instructions that follow assume that you have deployed the `composer.war` file as described in "Deploying the composer.war file on Tomcat" on page 4.

The Flex-based blueprints are configured by default for Flex-based authentication. You should not need to edit the blueprint files to enable Flex-based authentication.

**Procedure**

1. Navigate to your application server WEB-INF directory for your Composer application. For example: `C:\apache-tomcat-6.0.24\webapps\composer\WEB-INF`
2. Open the `beans-security.xml` Spring configuration file for editing. By default, the Flex configuration is commented out.
3. Activate the Flex configuration by editing the markup between the `<!-- Start Flex based authentication, no interception -->` tag and the `<!-- End Flex based authentication, no interception -->` tag. The markup is currently commented out using a false closing to the `<!-- Start Flex based authentication, no interception -->` tag. You can uncomment the markup by adding an additional hyphen to the end of the tag: `-->.`
4. Similarly, add an additional hyphen to the tag before End Flex based authentication, no interception in order to change it from <!- to <!--.
5. De-activate the JSP configuration by commenting out the markup between the <!-- Start JSP Login Form configuration --> and the <!-- End JSP Login Form configuration --> tag.
6. Save and close the beans-security.xml file.
7. Restart the web application or application server as necessary for the web application to be reloaded.

Configuring JSP-based authentication

Edit your blueprint files to fully enable JSP-based authentication.

About this task

By default, configuration files within composer.war are set to use JSP-based authentication. However, the Flex-based blueprints are configured with Flex-based authentication as the default. Edit the blueprint files to enable JSP-based authentication.

Procedure

1. Open the DemoMainApplicationContext.mxml context file.
2. Comment out the setting <security:FormAuthenticationProvider/>.
3. Uncomment the setting <security:JSPFormAuthenticationProvider/>.
4. Save and close the DemoMainApplicationContext.mxml file.
5. Restart the web application or application server as necessary for the web application to be reloaded.

Configuring the session timeout

To configure the timeout, edit the session-timeout parameter within the web.xml file for your application server. The default is 20 minutes.

Customizing the login panel (Flex-based authentication)

If you are using the default Flex-based authentication, you can change the appearance of the login panel by creating a custom login form component.

About this task

By default, the DemoMainApplication.mxml includes the LoginForm component to display a login panel that manages user login. You can customize the login panel by creating your own component and editing theDemoMainApplication.mxml file so that it includes the component you create. Ensure that the login form component you create invokes the LoginManager's login method with the user credentials. Also ensure that the component listens for the event that the LoginManager throws when a login fails: SecurityEvent.LOGIN_FAILED. (The event type is login_failed.) As an alternative to creating a login component, you can include new login markup directly within the DemoMainApplication.mxml file.

Procedure

1. Within IBM Initiate Workbench in the Flash Package Explorer, navigate to projectname/composer/src/(default package)/.
2. Open the DemoMainApplication.mxml file.
3. Locate the following markup:
4. Edit the markup to point to the login form component you have created. For example:

```xml
<s:NavigatorContent id="loginView">
  <components:LoginForm id="loginForm"/>
</s:NavigatorContent>
```

Here `myComponents:myLoginForm` represents your component's namespace and name. As an alternative to creating a component, you can include your custom login markup directly within the `DemoMainApplication.mxml` file after the `<s:NavigatorContent id="loginView">` tag.

5. Save and close the `DemoMainApplication.mxml` file.

**Customizing the Flex-based login panel: an example**

Sample code for the Flex-based login component.

**Purpose**

By default, the `DemoMainApplication.mxml` application file includes the `LoginForm` component to display a login panel that manages user login. Using the sample code here as a guide, you can customize the login panel by creating your own component. Edit the `DemoMainApplication.mxml` file so that it includes the component you create.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!--
A sample login form that can be customized for a customer's environment.
-->  
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"  
  xmlns:s="library://ns.adobe.com/flex/spark"  
  xmlns:parsley="http://www.spicefactory.org/parsley"  
  xmlns:mx="library://ns.adobe.com/flex/mx" width="100%" height="100%">  
  <fx:Declarations>  
    <parsley:Configure/>  
  </fx:Declarations>  
  <fx:Script>  
    <![CDATA[  
      import com.initiate.composer.security.LoginManager;  
      import com.initiate.composer.security.SecurityEvent;  
      import mx.controls.Alert;  
    ]]>
    /** Injected LoginManager */
    public var loginManager:LoginManager;
    
    /** Initializes the event listeners once Parsley has completed injecting. */
    public function init():void {
      username.text = "";
      password.text = "";
    }
    
    /** Called to clear the form elements. */
    public function clearForm():void {  
      loginForm.addEventListener(KeyboardEvent.KEY_DOWN, formCheckKey);  
      loginManager.addEventListener(SecurityEvent.LOGIN_FAILED, loginFailed);  
    }
    
    /** Event handler to cause a login when enter is pressed. */
  </fx:Script>
</s:Group>
```
private function formCheckKey(event:KeyboardEvent):void {
    if(event.charCode == Keyboard.ENTER) {
        doLogin();
    }
}

/** Invoked to login using the credentials in the form. */
private function doLogin():void {
    loginManager.login(username.text, password.text);
}

/** Simple alert showing login failure. */
private function loginFailed(event:SecurityEvent):void {
    Alert.show("Login failed.");
}


Customizing the login panel (JSP-based authentication)

If you are using the JSP-based authentication, you can change the appearance of
the login panel by editing or replacing the login.jsp file.

About this task

If you have configured JSP-based authentication instead of the default Flex-based
authentication, the appearance of the login panel is governed by the login.jsp file.
For additional information, see "Configuring JSP-based authentication" on page 40.

Procedure

1. In the file system for your application server, navigate to the webapps\composer
directory for your application. For example: C:\apache-tomcat-6.0.24\webapps\composer
2. Open the login.jsp, login.css, and login.js files for editing.
3. Edit the files to reflect any customization you want to make.
4. Save and close the files.

Choosing how to link records when making edits to entities

As part of configuring a IBM Initiate Master Data Service Composer application
that allows users to edit entities, decide how the application links records when
adding or changing attribute values.

About this task

Because an entity is composed of one or more member records, editing an entity
means that the user is actually editing one of the member records that comprises
the entity. Specifically, the application writes the user's edits into the record maintained within the source specified during configuration of the blueprint.

As described elsewhere, it is recommended that you configure a Hub-controlled source to serve as a repository for the edits that Composer applications make to entities. See "Configuring a source" on page 10.

By default, when a user clicks Update to submit edits, the application writes the new values to the specified source and then issues a command that binds together the associated records for the entity using an identity rule. The identity rule indicates that two or more member records have been assigned the same Member ID. (In the API, this identity rule is known as an entity rule, specifically EntRule.) This linking occurs each time a user edits an entity. If you have created and configured a new Hub-controlled source, the first time the user edits an existing entity, the application creates the record in the Hub-controlled source, and links the new record to any other records according to the identity rule.

As described in the steps below, you can change the default "hard link" behavior so that when the user edits an entity, the application writes the edited values into a member record within the selected source (as with hard linking) but rather than linking the member records, Composer reads the values from the other member records that comprise the entity — and writes those values into the member record within the selected source, where they appear as inactive values. If there are multiple values for an attribute across the other source records, all values are written to the specified member record as inactive values.

Choosing this "soft link" behavior might be important for installations that also use the IBM Initiate Entity Manager, especially if the algorithms within the Entity Manager use inactive values to compare and score member attribute similarities and differences.

The steps that follow assume that you have configured a blueprint that allows users to edit entities. (With the current release, only the Multi-Domain Entity Search blueprint includes the ability to edit entities.)

**Procedure**

1. In IBM Initiate Workbench within the Flash Package Explorer, navigate to `projectname/composer/src/blueprints/blueprintname/`
2. Open the `EntityDetailTabContext.mxml` context file for editing.
3. Locate the line that begins `<components:EntityDetailTablePM ...>`
4. Place your cursor after the "PM" in `EntityDetailTablePM` and hit the space bar. A drop-down menu allows you to insert a parameter for the PM file.
5. Choose the `updateEntityType` parameter and set the value to `ModifySourceExplicit`. (The default setting is `ModifySourceHardLink`.)
6. Save and close the `EntityDetailTabContext.mxml` file.

---

**Enabling communication between Initiate Composer applications and JavaScript**

You can enable communication between your IBM Initiate Master Data Service Composer applications and HTML/JavaScript using the Adobe Flash Builder `ExternalInterface` class.
About this task

The use of the ExternalInterface class within your Composer ActionScript code allows you to manage both directions of communication:

- ActionScript within your Composer application can call JavaScript functions in the parent HTML/JavaScript web page.
- JavaScript can call ActionScript functions within your Composer application that have been exposed.


Note: The ExternalInterface class requires the user's web browser to support either ActiveX or the NPRuntime API that is exposed by some browsers for plug-in scripting. See [http://www.mozilla.org/projects/plugins/npruntime.html](http://www.mozilla.org/projects/plugins/npruntime.html).

Using the ExternalInterface class, ActionScript can call any JavaScript function. The class allows ActionScript to pass arguments and receive return values from the JavaScript function. By the same token, ExternalInterface allows JavaScript on an HTML page to call an ActionScript function, including passing arguments to the ActionScript and receiving values.

Calling an ActionScript function from JavaScript code: an example

An example of JavaScript (JS) code that calls an ActionScript (AS) function within the IBM Initiate Master Data Service Composer application.

About this task

Consider an HTML page that includes radio button options for switching between three separate tabs within a Composer web application. Suppose that all three tabs are search forms, but one form allows the user to search for patients, another for providers, and a third for organizations.

The HTML page includes a JS function (radioClick) to be called when a user selects one of the radio buttons:

```javascript
function radioClick()
{
    if(document.getElementById('txtType1').checked) {
        getFlexApp('${application}').htmlRadioClick(0);
    } else if(document.getElementById('txtType2').checked) {
        getFlexApp('${application}').htmlRadioClick(1);
    } else {
        getFlexApp('${application}').htmlRadioClick(2);
    }
}
```

For the Composer application to respond to the JS call, the DemoMainApplication.mxml file must include code that makes an AS function (htmlRadioClick) available to JS. It does so with the ExternalInterface.addCallback method:

```xml
<mx:Script>
<![CDATA[
    protected function application1_initializeHandler(event:FlexEvent):void {

```
// expose an ActionScript function to the container
ExternalInterface.addCallback("htmlRadioClick", htmlRadioClick);
}
</mx:Script>

For security reasons, the parent HTML/JS application can only call AS functions that have been granted permission with the addCallback method.

Elsewhere in the DemoMainApplication.mxml file, htmlRadioClick is defined as a public function, while AS code specifies which search form tab to display based on the user's selection.

Creating an Initiate Composer object factory

To create your own Composer façade objects, implement a Composer factory for generating entities, records, attributes, or fields.

About this task

Create ActionScript files for implementing the factory and the new object, then inject the factory into your context.

Procedure

1. Create an ActionScript file for the factory. For example, create a PersonFactory.as file that implements EntityFactory:

   ```actionscript
   package model {
   import com.initiate.composer.model.core.Entity;
   import com.initiate.composer.model.utils.EntityFactory;

   public class PersonFactory implements EntityFactory {
   public function PersonFactory() {
   }
   public function createInstance(enttype:String):Entity {
   if (enttype == "id") {
   return new Person();
   } else {
   return null;
   }
   }
   }
   }
   ```

   You can also implement factories for records (RecordFactory), fields (FieldFactory), and attributes (AttributeFactory). Edit the sample code to reflect the interface for the object. For example, for the implementation of RecordFactory, change public function
creatInstance(enttype:String):Entity to public function
creatInstance(memtype:String):Record. See the ASDoc for specifics:

2. Create an ActionScript file for the object itself. For example, create a Person.as file with the following contents:
package model
{
    import mx.collections.ArrayCollection;
    import com.initiate.composer.model.core.*;

    public class Person extends EntityImpl{
    
        public function get firstName():String{
            return String(view.getField("LGLNAME","onmfirst"));
        }

        public function set firstName(firstName:String):void{
            view.setField("LGLNAME", "onmfirst", firstName);
        }
    }
}

Here again, edit the sample code to reflect the object type.

3. Finally, add code to your context file to inject the factory into the context:

```xml
<fx:Declarations>
<!-- This gets injected into the ComposerObjectFactory to have Composer create Person entity objects instead of generic Entity objects -->
    <facadeModel:PersonFactory/>
...
</fx:Declarations>
```
Chapter 4. Application infrastructure

An understanding of the IBM Initiate Master Data Service Composer application infrastructure provides the foundation required for you to create custom Composer components and applications.

Once you have absorbed the information in the topics that follow, you can create custom Flex components and insert them into Composer applications to fulfill specific needs. For example, you might create a component to perform a security check before returning detailed information about a person in your source. Perhaps you need the component to be launched in response to an event in the Search Results component. If so, the new component must respond to the SearchResultSelectedEvent event that the Search Results component generates when the user selects a search result. In addition, you would configure event handling to listen for and process authorization approval events generated by your custom component.

The information in the topics that follow are relevant if you intend to create custom Flex components. If instead you must enable communication between Composer and JavaScript code, see “Enabling communication between Initiate Composer applications and JavaScript” on page 43.

Initiate Composer and Parsley

Built on an IOC container and messaging framework, Parsley is an application framework for Flex and Flash Applications that can be used to create highly de-coupled architectures.

Because the handling of events is separated from the presentation layer (managed by the components), aspects of the application such as layout and styles can be changed independently of the event communication infrastructure. Parsley allows you to configure objects to be managed by the container with Metadata, MXML, XML, or ActionScript. Parsley’s IOC container supports dependency injection using the [Inject] AS3 metadata tag, which Composer blueprint code uses extensively to inject PMs into components, to inject properties by ID, as well as to inject the object factory, metadata manager, and configuration service.

The Parsley documentation is available at [http://www.spicefactory.org/parsley/](http://www.spicefactory.org/parsley/).

Initiate Composer components and PMs

IBM Initiate Master Data Service Composer components use the [Inject] tag within component files to inject the presentation model (PM) into the view.

Each Composer component has its own presentation model (PM) that maintains variables and functions to manage the state and behavior required to present data to users. PMs also process user input (events) flowing from the components.

The PM does not directly refer to the component. Rather, the component observes the PM. The component and its corresponding PM are wired together using binding expressions and event handlers in the form of message handlers and command results.
If you want to adapt a Composer component, create your own .mxml UI class and inject the existing PM into the new class. For example, MyComponent.mxml injects MyComponentPM into the view:

```mxml
[Inject]
[Bindable]
public var model:MyComponentPM;
```

Here model is the backing model used with this component.

Also be certain to include custom components within your context. See "Component reference" on page 81.

---

### Events

Flex applications like those created with IBM Initiate Master Data Service Composer are driven by events, which might include external/network events, internal application events, and user actions.

Because the Composer components are delivered within a Flex SWC library, the individual component code is not exposed. Nevertheless, the documentation within the "Component reference" on page 81 describes the individual outgoing events that each component generates. Any application you create must include handlers to manage the events generated by the Composer components you include.

In the context of Composer, all events result from user actions intended to:

- Perform a search for an entity or member based on user-provided values.
- Retrieve details for a specific entity or member returned by a search.
- For a specific entity, display the constituent member records.
- Add a new member to a source.
- Edit an entity and update the record in the source.
- Graphically display the relationships between entities.
- Display a table of entities related to a selected entity.
- Clear search parameters from a search form.

Consider a search event. As implemented in the Composer blueprints, clicking **Search** within a Search Form component prompts the display of results within a Search Results component. Because of the need to pass events between components, Composer blueprints take advantage of the Parsley framework, which supplies a system for listening for and responding to events.


Most Composer events require some communication with the IBM Initiate Master Data Service. For example, when a user clicks **Search** on a Search Form component in a blueprint, the PM dispatches the event which Parsley dispatches in turn to any listeners that are registered to listen to the particular event. In this case, the listener is a command that queries the IBM Initiate Master Data Service (through a REST call) for a list of matches. With `[CommandResult]` code, a presenter listens for the results of the command returned from the IBM Initiate Master Data Service Engine.
and determines how the event and its result payload are routed. In this case, the presenter would pass the result to a searchResultsPM for display within a Search Results component.

A single component (PM) can be both the dispatcher and the recipient of an event. For example, in the Multi-Domain Entity Search blueprint, EntityRecordListPM dispatches an event to a retrieve a record list. The event is captured by a command, routed to the IBM Initiate Master Data Service Engine, and the results are routed directly back to the EntityRecordListPM for display within EntityRecordList. The EntityRecordListPM includes its own [CommandResult] handler; in other words, in this case no presenter is involved.

With Parsley, commands get registered into contexts. As you create your custom application, ensure that the context handles all events generated by the components you include. For details about editing a context file to enable a component, see the individual component topics within the “Component reference” on page 81.

By contrast with commands, some Composer events do not require communication with the IBM Initiate Master Data Service. These events can simply be routed between the components. Consider the event launched when a user clicks to view the details of a particular search result. Because the original search has already returned a list of matches with full attribute information for each item in the list, viewing the details only requires Composer to expose attributes that it has already fetched. With [MessageHandler] code, a presenter listens for the event, which comes directly from the PM, and routes the event to whatever PM has been specified in the [MessageHandler] code. In this case, the presenter would pass a SearchResultSelectedEvent event (with the selected record as its payload) to a recordDetailPM for display.

Event names within the [MessageHandler] and [CommandResult] tags indicate which events a presenter is listening for. Some [MessageHandler] and [CommandResult] tags also include a selector specification and scope specification which further refine the events the presenter attends to. Event classes such as AddEvent and RecordItemSelectedEvent define events and any selectors as constants. After initialization, Parsley injects the values for the events (and selectors) into the PMs. The values define the events that the PMs dispatch.

Within Composer blueprints, Composer components and their supporting elements work with minimal configuration and wiring. Knowing clearly which aspects of your application to customize - and which aspects to leave alone - allows you to focus your efforts effectively. Proceeding with care is especially important when customizing the communication infrastructure that ties the components together. If you must expand or customize a blueprint or assemble an application from scratch, first understand the details of component communication.

**PMs and events**

With the Multi-Domain Search Entity blueprint, PMs dispatch events on behalf of components.

Although the source code for the PM files is not exposed, as you consider creating custom components, it might be of use to understand how the Multi-Domain Search Entity blueprint PMs manage events. Within the blueprint PMs, [Event]
tags declare the events that the class dispatches while the [ManagedEvents] tag tells Parsley which of those declared events to "manage" so that they are available for presenters.

Consider the following code from a search form component:

```xml
[Event(name="entitySearch", type="com.initiate.composer.events.SearchEvent")]
[Event(name="recordSearch", type="com.initiate.composer.events.SearchEvent")]
[ManagedEvents(names="entitySearch, recordSearch")]
public class SearchFormPM extends EventDispatcher
[...]
public function doSearch():Boolean
{
  var event:SearchEvent = new SearchEvent(searchType, record, enttype);
  return dispatchEvent(event);
}
```

The dispatched SearchEvent includes parameters for searchType, record, and enttype. For information about configuring those parameters, see the "Component reference" on page 81.

You can configure the [MessageHandler] tag within presenters to specify interest in events based on the parameters on the method. Because IBM Initiate Master Data Service Composer applications route several different message types of the same event class, message handlers within presenters use selectors to specify the particular events they attend to.

PMs dispatch various events, but presenters attend only to those managed by Parsley (that is, those events that appear within the PM’s [ManagedEvents] tag).

**Presenters and events**

Presenters within the Multi-Domain Entity Search blueprint manage events using either [MessageHandler] or [CommandResult] tags depending on whether the event requires communication with the Master Data Service engine.

Based on the event (class) and optionally based on selectors, the code within presenters determines which events to attend to. Unlike typical event handlers, Parsley events are not string-based. This inversion of control aids in the development of large applications in particular because there is no need for unique event type constants. In fact, messaging can be configured to occur within particular scopes, locally or globally as your application requires.

The sample code here includes a message handler configured to listen for SearchResultSelectedEvent events of the event type recordSearchResultSelected. The handleRecordSearchResultSelection method is invoked whenever a message of event type SearchResultSelectedEvent and selector recordSearchResultSelected gets dispatched. When the presenter detects the event, it notifies the RecordDetailTablePM.

The sample code also includes a [CommandResult] tag configured to listen for any event matching the SearchEvent event type that also has a result event. (Result events typically occur with asynchronous processes.) The event triggers the handleSearchResults method.

```xml
public class SearchScreenPresenter extends EventDispatcher
{
[...]
[MessageHandler(selector="recordSearchResultSelected")]
public function handleRecordSearchResultSelection(event:SearchRes
MVC framework

In keeping with the principles of model-view-controller (MVC) and inversion of control (IOC), IBM Initiate Master Data Service Composer separates the presentation of the application (for example, the layout, styles, labels, and banner artwork) from the underlying communication and data processing.

Specifically, the model is the underlying model for the data maintained by the Master Data Service Engine. The view is managed by components whose layout and style you can configure. The controller is the architecture and communication framework that ties the pieces of the application together; it consists of contexts, screens, presenters, and per-component PMs. See the Parsley documentation for more detail: [http://www.spicefactory.org/parsley/](http://www.spicefactory.org/parsley/)

Error handling

Code within IBM Initiate Master Data Service Composer blueprints manages errors. Adapt the code to handle errors according to your needs.

By default, Composer blueprints display errors in a dialog window (ComposerAlert) within the application. You can configure additional error handling, for example, to manage failures on your login page.

The following code appears within QuickViewBlueprint.mxml (for the Quick View blueprint) and within TabbedEntitySearchScreen.mxml (for the Multi-Domain Entity Search blueprint):

```mxml
[CommandError(scope="local")]public function handleSearchFault(fault:FaultEvent, trigger:SearchEvent):void
{
    searchResultsPM.searchResults = null;
    ComposerAlert.displayFault(fault, this);
}
```

The scope for CommandError must always be set to local.

For any application you create, insert error handling code within the main page for a context. Because commands run in the local scope, errors are specific to a context. You cannot manage errors at a global level.

Chapter 5. Extending Initiate Composer

You can extend the base functionality of Composer by overriding a component, creating a custom PM, or extending the Composer services.

Overriding a component

Developers can modify Composer components to satisfy particular business requirements.

About this task

The Composer Flex SDK was created using an Inversion of Control (IOC) architecture model. This model allows a developer to quickly and easily modify only those components that need to be changed to satisfy their business requirements. In this section we will walk through the steps necessary create a custom component that extends the functionality of an existing SearchForm.

The topics in this section describe an example in which the SearchForm component is modified to allow the user to set the minimum score required for a match.

Setup

To enable the example, first create and configure a QuickView blueprint.

About this task

Create and configure the QuickView blueprint as described below. For detailed steps, see “Configuring blueprints” on page 7.

1. Create a QuickView blueprint. The example uses configures the blueprint for an entity search on id.
2. If it isn’t already set, in QuickViewBlueprintContext.mxml set the showScore property to true in the SearchResults. See “Displaying the score for search results” on page 17.

Creating the custom component

Create a new MXML component that extends the generic sparks control group.

About this task

Because the minimum score is a publicly available property that can be statically set in the context file, simply modify the UI layer to add this functionality.

Adding a component to the original SearchForm does not require you to extend the class. Simply wrap the new component around the existing one.

1. Within Flash Builder, choose File > New MXML Component.
2. In the New MXML Component window, set the Package to blueprints.quickview.custom.search.
3. Specify a unique Name, for example CustomSearchForm.
4. Verify that Based on is spark.components.Group.
5. Set the Width and Height in pixels, for example 400 by 300.
6. Click **Finish**.

First, satisfy the contract for the search form on this control by exposing all the public properties that the user can set on the original component and by adding the actual SearchForm to this new control.

```xml
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:composer="http://www.ibm.com/initiate/composer">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:String id="configName"/>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import com.initiate.composer.events.ClearEvent;
            public function clear(clearEvent:ClearEvent= null) {
                searchForm.clear(clearEvent);
            }
        ]]>
    </fx:Script>
    <composer:SearchForm id="searchForm" configName="(configName)"/>
</s:VGroup>
```

Next apply the changes to the form to fit your needs. In this case, add a control to allow the user to input the value and then bind it back to the model in the searchForm. By adding the valueCommit event, only when the value is persisted on the stepper will it update the model in the backend.

```xml
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:composer="http://www.ibm.com/initiate/composer">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
    <fx:Declarations>
        <fx:String id="configName"/>
    </fx:Declarations>
    <fx:Script>
        <![CDATA[
            import com.initiate.composer.events.ClearEvent;
            import mx.events.FlexEvent;
            public function clear(clearEvent:ClearEvent= null) {
                searchForm.clear(clearEvent);
            }
            public function updateValue(event:FlexEvent) {
                searchForm.model.minScore = stepper.value;
            }
        ]]>
    </fx:Script>
    <mx:HBox>
        <s:Label text="Minimum Score to Search On"/>
        <s:NumericStepper minimum="0" maximum="100" value="(searchForm.model.minScore)" id="stepper" valueCommit="updateValue(event)"/>
    </mx:HBox>
</s:Group>
```
The changes are complete. Next modify the QuickViewBlueprint.mxml to use the new form. Comment out the default component and replace it with the custom one you have just created.

<!--
This is the main layout of the blueprint. It adds and positions the various components within the application.
The "configName" property on the components is the name of the Composer configuration Attribute Set to use within the component. This will adapt the component to a user-defined model at runtime based on the configuration.
-->

<mx:HDividedBox width="100%" height="100%">
  <s:Panel title="Search Form" width="30%" height="100%">
    <search:CustomSearchForm id="SearchForm" configName="Person Search"/>
    <!--
    <components:SearchForm id="searchForm" configName="Person Search"/>
    -->
  </s:Panel>
  <mx:VDividedBox width="70%" height="100%">
    <s:Panel title="Search Results" width="100%" height="50%">
      <components:SearchResults id="searchResults" configName="Person Search"/>
    </s:Panel>
    <s:Panel title="Details" width="100%" height="50%">
      <components:EntityDetailTable id="detailsView" configName="Person Search"/>
    </s:Panel>
  </mx:VDividedBox>
</mx:HDividedBox>

Run the application. Test it by searching for a known entity without setting the min score. Look at the scores and set the min score above at least one of the scores. Rerun the search and confirm that the results that fall below the minimum score are not returned.

**Overriding a default PM**

To satisfy particular business requirements, developers can modify Composer Presentation Managers (PMs).

**About this task**

The Composer Flex SDK was created using an Inversion of Control (IOC) architecture model. This model allows a developer to quickly and easily modify only those components that need to be changed to satisfy their business requirements. In this section we will walk through the steps necessary to override a default Presentation Manager (PM).

For the example offered in this section, you modify the SearchFormPM to only search on the name "Patty Countryman." Although not a practical use case, the example demonstrates the steps necessary to implement a custom model. A more
practical implementation might involve adding additional components to the UI layer and then setting them on the model as additional search parameters.

Setup

To enable the example, first create and configure a QuickView blueprint.

About this task

Create and configure the QuickView blueprint as described below. For detailed steps, see “Configuring blueprints” on page 7.

1. Create a QuickView blueprint. The example uses configures the blueprint for an entity search on id.
2. If it isn’t already set, in QuickViewBlueprintContext.mxml set the showScore property to true in the SearchResults. See “Displaying the score for search results” on page 17.

Creating the custom PM

Because the new PM only needs to override the search behavior, create a new ActionScript Class that extends the existing component.

About this task

1. Within Flash Builder, choose File > New ActionScript Class.
2. In the New ActionScript Class window, set the Package to blueprints.quickview.custom.search.
3. Specify a unique Name, for example CustomSearchFormPM.
4. Set the Modifiers to public.
5. Leave dynamic and final unchecked.
6. Set Superclass to com.initiate.composer.components.SearchFormPM.
7. For Code generation options, check Generate constructor from superclass.
8. Click Finish.

When using Parsley, you must re-register all of the events that the class will throw, and notify Parsley of the events it must manage. Although the example does not change any events, you must nevertheless register them on the class.

package blueprints.quickview.custom.search
{import com.initiate.composer.components.SearchFormPM;

[Event(name="entitySearch", type="com.initiate.composer.events.SearchEvent")]
[Event(name="recordSearch", type="com.initiate.composer.events.SearchEvent")]
[ManagedEvents(names="entitySearch, recordSearch")]
public class CustomSearchFormPM extends SearchFormPM
{
  public function CustomSearchFormPM()
  {
    super();
  }
}
}

Next apply the changes to the PM to fit your needs. In this case the doSearch() is called from the UI and so we must override that method.
package blueprints.quickview.custom.search
{
  import com.initiate.composer.components.SearchFormPM;
  import com.initiate.composer.events.SearchEvent;

  [Event(name="entitySearch", type="com.initiate.composer.
  events.SearchEvent")]
  [Event(name="recordSearch", type="com.initiate.composer.
  events.SearchEvent")]

  [ManagedEvents(names="entitySearch, recordSearch")]
  public class CustomSearchFormPM extends SearchFormPM
  {
    public function CustomSearchFormPM()
    {
      super();
    }

    override public function doSearch():Boolean {
      record.setField("LGLNAME", "onmfirst", "Patty");
      record.setField("LGLNAME", "onmlast", "Countryman");
      var event:SearchEvent = new SearchEvent(searchType,
          record, enttype, _cvwname, _minScore, _maxResults);

      return dispatchEvent(event);
    }
  }
}

Notice that the matches the event types declared in the MetaData, and that it has
been added to the import list. To include other properties in the payload, set them
at this point before dispatching the event.

Note: in the overridden function doSearch, you may need to change the
parameters for the record.setField instance object method call to match the
attributes and fields defined in your Workbench project. For example, the demo
project Person's name attribute code is “NAME” not “LGLNAME”. The new code
would be:
record.setField("NAME", "onmfirst", "Patty");
record.setField("NAME", "onmlast", "Countryman");

As a final step, update your configuration files to notify Parsley to use the custom
model instead of the default model. Open the file
QuickViewBlueprintContext.mxml file and comment out the line that refers to the
SearchFormPM, then add the reference to the CustomSearchFormPM that you
created.

<fx:Script>
<![CDATA[
  import com.initiate.composer.events.SearchEvent;
  import com.initiate.composer.events.AddEvent;
  ]]>
</fx:Script>

<fx:Declarations>
<!--
Components generate events like SearchEvent, AddEvent,
etc. These events however need to be handled by a Command.
Composer provides defualt commands to handle these events,
but to use these (or custom alternates) you must add them to
the context where they are needed (where the events are dispatched
from).
-->
<commands:SearchCommand/>

<!--
For the most part, each Composer component (UI/mxml class) has a
corresponding presentation model (ActionScript class).
The presentation model is responsible for holding the data the
composer binds to and creating/handling/dispatching any events
the component is involved in.

Some of these components require defaults to be set, like memtype,
etype, searchType, etc.

In most cases, these presentation model classes have other properties
with sane defaults that may be overridden here (like maxResults, minScore,
etc.).
-->
<commands:CustomSearchFormPM memtype="PERSON" enttype="id" searchType="entitySearch"/>

<!--
<commands:SearchFormPM memtype="PERSON" enttype="id" searchType="entitySearch"/>

<components:SearchResultsPM showScore="true"/>
<components:EntityDetailTablePM/>

Now when you run the application, the search results return only Patty
Countryman.

---

### Extending services

To extend Composer services, author new REST services and co-deploy them with
the existing Composer services.

Extending Composer to include new services can be done using several methods.
One method is to use REST services that are already defined somewhere and
connect to them directly using Flex. This usually requires some form of proxy or
codeployment to access the service. In this case new Flex classes may need to be
authored to access these services directly or indirectly through a proxy.

Another method is to author new REST services and co-deploy them with the
existing Composer services. This requires the authoring and deployment of these
new services. Once deployed, these new services can be exposed on the client by
adding them to the existing IBM Initiate Master Data Service interface and
implementations through normal Flex extension methods. The following topics
describe this second method of extending Composer.

#### Using Spring

The Composer services are deployed in the composer.war file using Spring and
CXF.

Some knowledge of Spring and its configuration files is required to add new REST
services to the composer.war file. The topics in this section provide enough
information to configure the example.

#### Creating a REST service

For the purposes of this example we add a new search method that integrates with
an outside data provider to identify records that are ultimately returned from the
IBM Initiate Master Data Service.
The external data provider in this example might be a name recognition service or some form of search appliance that returns record identifiers that can be used to retrieve records from the IBM Initiate Master Data Service.

**Verifying dependencies**
The example requires the use of three libraries located within the composer.war file.

Confirm the presence of the following files within the composer.war file WEB-INF/lib/ directory:

- madapi.jar: This library is part of the core Java API for the IBM Initiate Master Data Service.
- jsr311-api-1.0.jar: This library contains the JAX-RS API used for RESTful Web Services.
- madapi2.jar: This library is part of the newer ESOA and Generic IBM Initiate Master Data Service API.

Any additional dependency required for the outside data provider should also be included in the project used to build these extensions.

**Defining the service interface**
We begin by defining an interface. The search method defined by the interface simply accepts a query to be passed to the external data search index.

One of the primary tenets of Spring is that you should program using interfaces instead of classes directly. The approach taken with the IBM Initiate Master Data Service Composer REST services is no different. The set of services in Initiate Composer is comprised of multiple service interfaces that provide the relevant features deployed to the REST server. Each service interface in use normally groups methods based on the type of functionality they provide. This service interface defines how the method or methods will operate independent of REST. This interface can be tested independent of REST and looks like a normal Java interface.

In our example, we provide a new search method that interfaces with an external data search index. The search method defined by the interface simply accepts a query to be passed to the external data search index. It returns a list of GenericRecord objects up to the maxResults specified.

```java
package com.initiate.composer.extension.service;

import java.util.List;
import com.initiate.client.generic.GenericRecord;
import com.initiate.client.internal.MasterDataServiceException;

/**
 * This interface defines the service interface for our search extension to Initiate Composer.
 */
public interface SearchExtension {
    /**
     * Searches for records using an external data index.
     * @param query the query to pass to the external data index
     * @param maxResults the maximum number of records to return
     * @return a list of GenericRecord objects that matched the
     */
    List<GenericRecord> search(String query, int maxResults);
}
```

Chapter 5. Extending Initiate Composer  59
Defining the service implementation

As mentioned previously separating the implementation from the interface is good programming practice preferred by Spring. The implementation of our service is responsible for performing the search and then returning the list of GenericRecord objects.

The following source code shows the bulk of the implementation class with the exception of the imports and method body which have been omitted for brevity. The method body is shown in a subsequent code block. The important thing to notice about the following source code is the injection of a GenericMasterDataService and a ServiceMetaData object. These two objects are part of the generic IBM Initiate Master Data Service API that we'll need in order to get the records from the IBM Initiate Master Data Service once we have the record identifiers back from the external data search index.

```java
public class SearchExtensionImpl implements SearchExtension {
    private GenericMasterDataService mds;
    private ServiceMetaData metadata;

    protected GenericMasterDataService getMds() {
        return mds;
    }

    public void setMds(GenericMasterDataService mds) {
        this.mds = mds;
    }

    protected ServiceMetaData getMetadata() {
        return metadata;
    }

    /*
     * query
     * @throws MasterDataServiceException if there's a problem
     * performing the search
     */
    List<GenericRecord> search(String query, int maxResults)
        throws MasterDataServiceException;
}
```
Any search method implementation would be specific to the external index. That part of the code is left as an exercise for the reader since it cannot be coded until the external data search index is selected. The method implementation that follows is currently not coded to access an external search index. It’s presently creating new record identifiers using hardcoded values. The comments in the method body should clearly explain what portions of the code remain to be implemented. If you haven’t yet selected an external data search index, be sure to use record identifiers appropriate for your deployment.

```java
public List<GenericRecord> search(String query, int maxResults) throws MasterDataServiceException {
    // method body omitted for brevity
}
```

Once the record identifiers have been obtained from the external data search interface it is fairly straightforward to obtain the list of records from the IBM Initiate Master Data Service using the getRecordList method of the GenericMasterDataService. If support for other parameters like the composite view name or attribute filtering are desired, they can be added to the method signature in the interface and implementation classes. The code makes some simplifying assumptions, for example that all record identifiers are of the same member type. If they are not, adjust the code accordingly.
Defining the REST interface

The REST interface is responsible for identifying how our service is to be exposed as a REST service.

The method signatures between the service interface and the REST interface are similar but differ primarily in their return value. The REST interface is where we place our JAX-RS annotations that describe how our service is deployed and how the HTTP requests are mapped to our interface. The following source code demonstrates how our search extension is deployed.

```java
package com.initiate.composer.extension.service.rest;

import java.util.List;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import com.initiate.client.generic.GenericRecord;
import com.initiate.client.internal.MasterDataServiceException;

@Consumes({MediaType.APPLICATION_JSON, "text/json",
            MediaType.APPLICATION_XML, MediaType.TEXT_XML})
@Produces({MediaType.APPLICATION_JSON, "text/json",
            MediaType.APPLICATION_XML, MediaType.TEXT_XML})
@Path("/search")
public interface SearchRestExtension {

  /**
   * Search for records using the supplied query string.
   * @param query the search query string
   * @param maxResults [OPTIONAL] the max results to return
   * @return JAX-RS response containing the record search results
   * @throws MasterDataServiceException
   */
  @GET
  Response search(
      @QueryParam("text") String query,
      @QueryParam("maxResults") int maxResults)
      throws MasterDataServiceException;

  /**
   * @httpStatus 200 Ok.
   */
  public static final List<GenericRecord> search = null;
}
```

The Consumes and Produces annotations tell the JAX-RS server what types this interface can accept or create. The Path annotation determines where our service will be deployed in the JAX-RS server. The important thing to note about this path is that it needs to be unique for any given JAX-RS server. Trying to deploy two
services to the same URL will not work. In this case any method sent to /search will be applied to this interface. The GET annotation on the search method indicates that when the HTTP method is GET that this method should be invoked. Finally the QueryParam annotations identify which query parameters are applied to which method parameters. Note that the static variable search is not required by JAX-RS and is included for use by our automated REST documentation tools to document the return value of the REST method.

**Defining the REST implementation**

The implementation of our REST service in most cases will be a simple delegation to our previously developed service. The exception would be those cases where parameters may need to be converted into different values.

The following source code shows the implementation class. A subsequent code block shows the method body.

```java
/**
 * Copyright (c) 1998-2011 Initiate Systems, Inc. All Rights Reserved.
 */
package com.initiate.composer.extension.service.rest;

import java.util.List;
import javax.ws.rs.core.Response;
import com.initiate.client.internal.MasterDataServiceException;
import com.initiate.client.generic.GenericRecord;
import com.initiate.composer.extension.service.SearchExtension;

public class SearchRestExtensionImpl implements SearchRestExtension {
    private SearchExtension searchExtension;

    protected SearchExtension getSearchExtension() {
        return searchExtension;
    }

    public void setSearchExtension(SearchExtension searchExtension) {
        this.searchExtension = searchExtension;
    }

    /**
     * REST implementation of search
     *
     * @param query the search query string
     * @param maxResults [OPTIONAL] the max results to return
     * @return JAX-RS response containing the record search results
     * @throws MasterDataServiceException
     */
    @Override
    public Response search(String query, int maxResults)
    {
        // Implementation logic here
    }
}
```
Like our prior service implementation, this implementation also has an injected object. In this case we’re injecting an instance of our service interface that will be used by our REST implementation.

```java
public Response search(String query, int maxResults) throws MasterDataServiceException {
    // get the records from the service
    List<GenericRecord> listResult =
        getSearchExtension().search(query, maxResults);
    // wrap the result in a JAX-RS entity for inclusion in the response
    javax.ws.rs.core.GenericEntity<List<GenericRecord>> entity =
        new javax.ws.rs.core.GenericEntity<List<GenericRecord>>(listResult) {};
    // build the response and return it
    return Response.ok(entity).build();
}
```

The search method first delegates to our service interface to get the list of records. The next two steps are responsible for wrapping the result and then returning it as the response to the original request. Note that the fully qualified class name for the JAX-RS `GenericEntity` is used since the Generic IBM Initiate Master Data Service API also has a `GenericEntity` class.

## Defining the service deployment

Deploying the service with Composer requires a few simple steps. The service classes must be compiled and packaged in the `composer.war` file. You can deploy them as a JAR file in `WEB-INF/lib` or copy the classes into `WEB-INF/classes`. Using a separate JAR file should be the preferred option for non-test environments.

Once the service classes are packaged into the WAR file, add them to the Spring configuration to expose them as REST service. This is done in two files in `WEB-INF`: `beans-composer.xml` and `beans-rest-common.xml`.

The `beans-composer.xml` file is the main context configuration file for the application. It imports other files and also defines services unique to the application. We’ll define our search extension as another bean near the end of `beans-composer.xml`.

```xml
<beans>
    <!-- existing portions omitted for brevity -->
    <bean id="searchExtension"
        class="com.initiate.composer.extension.service.SearchExtensionImpl">
        <property name="mds" ref="mds"/>
        <property name="metadata" ref="serviceMetaData"/>
    </bean>
</beans>
```

This defines our search service as `searchExtension` and provides it with the two injected dependencies, the `GenericMasterDataService (mds)` and `ServiceMetaData (serviceMetaData)`, which are defined elsewhere in the configuration.

The `beans-rest-common.xml` file is imported into `beans-composer.xml` indirectly through `beans-rest.xml`. It contains definitions for other generic application
services as well as the configuration for CXF which serves as our JAX-RS provider. The second modification is made at the end of the jaxrs:serviceBeans element:

```xml
<beans>
  <!-- sections omitted for brevity -->
  <jaxrs:server>
    <jaxrs:serviceBeans>
      <!-- sections omitted for brevity -->
      <bean id="searchRestExtension" class="com.initiate.composer.extension.service.rest.SearchRestExtensionImpl">
        <property name="searchExtension" ref="searchExtension"/>
      </bean>
    </jaxrs:serviceBeans>
  </jaxrs:server>
</beans>
```

Here we define the searchRestExtension bean as a JAX-RS service and inject it with its dependency searchExtension that we added in the earlier change.

Once you've packaged the classes and made the Spring configuration changes the last step is to redeploy or restart your web application to the container you’re using. At the completion of this step you should be able to make a request like the following to see your service in action:

```
http://<host>:<port>/composer/services/search?query=query+string
```

**Note:** Depending on your configuration you may or may not be required to log in. If you receive an error message for an invalid password instead of the expected result then you are not logged in. You can log in by accessing the login JSP page here:

```
http://<host>:<port>/composer/login.jsp
```

Simply enter your user name and password and click **Login**. Once you are logged in you can re-send the request to check the result.

The result should be records that matched the query data sent to the external search index or in the case of our sample code, the records that match the hard coded identifiers.

**Extending the Flex MasterDataService**

Once you've got your new REST service packaged and deployed you'll want to access it from the client.

Because your service is located in the same JAX-RS CXF container as the other MDS REST services you can simplify the amount of code you have to write by adding your new REST service calls to the existing IBM Initiate Master Data Service.

**Adding search method to the MasterDataService interface (MyMasterDataService)**

For our example we expose the new search method that we added by adding it to the existing MasterDataService interface in the client.

As you can see this is simply an extension of the existing interface and we're adding the new search method. Any time you need to use this method you'll have to use this interface instead of the normal MasterDataService interface.
package com.initiate.composer.extension.service.mds
{
    import com.initiate.composer.services.mds.MasterDataService;
    import mx.rpc.AsyncToken;

    /**
     * An extension of the original MDS interface to add an
     * additional search.
     */
    public interface MyMasterDataService extends MasterDataService
    {
        function search(text:String, maxResults:int = -1):AsyncToken;
    }
}

Extending the implementation (MyMasterDataServiceImpl)

Now that we've extended the interface to define our new search method we need to extend the implementation to add the necessary code to invoke this method.

The following listing shows the overall structure of the file. Like before, the omitted areas will be covered in more detail.

package com.initiate.composer.extension.service.mds
{
    // imports omitted for brevity

    public class MyMasterDataServiceImpl
    extends MasterDataServiceImpl implements MyMasterDataService
    {
        private static var _log:ILogger =
            Log.getLogger("com.initiate.composer"+
            "+extension.service.mds.MyMasterDataServiceImpl");

        public function MyMasterDataServiceImpl()
        {
            // let the MDS get set up
            super();
        }

        override protected function buildOperations() : Array
        {
            // omitted for brevity
        }

        public function search(text:String, maxResults:int) : AsyncToken
        {
            // omitted for brevity
        }

        protected function search_result(result:Object,
            token:Object = null) : void
        {
            // omitted for brevity
        }

        protected function search_fault(error:FaultEvent,
            token:Object = null) : void
        {
            // omitted for brevity
        }
    }
}

The implementation class extends the MasterDataService and implements our new interface that adds the new search method. It contains a logger that can be useful for debugging code. See the Adobe documentation on how to enable logging for
more information. The constructor for this example is simple. It only invokes the parent class' constructor. While Flex would generate this constructor and super() call for us, it's best if we include it. The MasterDataService constructor will be invoking the buildOperations function so we cover that next.

```actionscript
override protected function buildOperations():Array {
    _log.info("Adding my operations.");

    // get the operations from our super class
    var operations:Array = super.buildOperations();

    // search
    var operation:Operation = new Operation(null, "search");
    operation.url = "search/";
    operation.headers["Accept"] = "application/xml";
    operation.headers["X-HTTP-Method-Override"] = "GET";
    operation.method = "POST";
    operation.resultFormat = "e4x";
    operation.serializationFilter = restSerializer;
    var argsArray:Array = new Array("text","maxResults");
    operation.argumentNames = argsArray;
    operation.properties = new Object();
    operation.properties[RESTSerializationFilter.PARAMETERS] = {
        text: { type:"query", optional:true },
        maxResults: { type:"query", optional:true }
    };
    operations.push(operation);

    // return operations array with our operations
    return operations;
}
```

This creates the configuration that describes our REST service. The important thing to note is that we first call the super class' buildOperations method to obtain the operations array from the parent. Then we construct our operation, add it to the array, and finally return it back to the caller. This allows extending classes to add their operations before they are set on the backing HTTPMultiService defined in MasterDataServiceImpl.

Most of the operation's properties are documented in the Adobe documentation. The url property is relative to the CXF servlet so the operation only includes the portion of the URL that appears after /composer/services/. Because we're not using a proxy and a limitation placed on the player you'll notice the use of the POST method combined with the X-HTTP-Method-Override header. This is required to be able to send headers on the request. The result format "e4x" tells the HTTPMultiService that the result will be XML and that we'll be using e4x syntax to work with the result.

The RESTSerializationFilter defined by the static variable restSerializer is responsible for building the URL and the body of the request. The argumentNames array and properties object are used by the RESTSerializationFilter to determine where the named arguments are to be used. There are three parameter types: entity, path, and query. The entity parameter is the body of the request and there's at most one per operation. Path parameters are part of the path and are defined inside braces. Query parameters are included in the query string if defined.

The following listing shows the search function implementation.

```actionscript
public function search(text:String, maxResults:int = -1) : AsyncToken {
    _log.info("search({0}, {1})", text, maxResults);
    // search
    var operation:Operation = new Operation(null, "search");
    operation.url = "search/";
    operation.headers["Accept"] = "application/xml";
    operation.headers["X-HTTP-Method-Override"] = "GET";
    operation.method = "POST";
    operation.resultFormat = "e4x";
    operation.serializationFilter = restSerializer;
    var argsArray:Array = new Array("text","maxResults");
    operation.argumentNames = argsArray;
    operation.properties = new Object();
    operation.properties[RESTSerializationFilter.PARAMETERS] = {
        text: { type:"query", optional:true },
        maxResults: { type:"query", optional:true }
    };
    operations.push(operation);

    // return operations array with our operations
    return operations;
}
```
var sMaxResults:String = null;
if (maxResults > -1) {
    sMaxResults = maxResults.toString();
}

var myToken:AsyncToken = new AsyncToken();

var operation:Operation =
    Operation(httpMultiService.getOperation("search"));
var httpToken:AsyncToken = operation.send(text, sMaxResults);
httpToken.addResponder(new AsyncResponder(search_result, search_fault, myToken));
resetTimer();
return myToken;
}

protected function search_result(result:Object, token:Object = null):void {
    var recordList:XML = result.result as XML;
    _log.debug("search_result: {0}", recordList);
    var array:ArrayCollection =
        XmlObjectFactory.toRecordListObject(recordList);
    var resultEvent:ResultEvent =
        ResultEvent.createEvent(array, token as AsyncToken);
    new AsyncDispatcher(dispatchRpcEvent, [resultEvent], 50);
}

protected function search_fault(error:FaultEvent, token:Object = null):void {
    logFault("search_fault", error);
    genericFaultHandler(error,token);
}

The search function prepares the parameters to be passed to the operation and creates the AsyncToken used to perform the asynchronous communication with the caller. Since maxResults is optional we only need to include it if the caller gave us a valid value. After obtaining our operation from the service we need to invoke the send method passing the arguments in the same order they were defined in the arguments array. We add our result and fault handlers to the AsyncToken returned by send along with the token we're returning to the caller. We also invoke resetTimer() to refresh the session timeout timer. Finally we return the token we created to the caller.

The result and fault handlers are called when the data from the REST service is returned or a fault occurs. In both cases the original AsyncToken we created in the search function is available as the second argument to the handlers. The result handler casts the result as an XML object and then invokes the XmlObjectFactory method to convert the result into the appropriate return value for our method. In this case it's converting the XML into an ArrayCollection of Record objects. The last two lines prepare a ResultEvent and dispatch that to the original caller using the dispatchRpcEvent method and the AsyncToken we created originally in the search function. The fault handler uses methods from the parent class to log and handle the error returned from the server. The fault handler is where you can trap specific HTTP status codes if they have meaning in your service. If you're not looking for a specific status code or the error is something other than your expected status code then it's best to pass the fault on and let the genericFaultHandler deal with it. It takes care of cases where the client is no longer authenticated and creates appropriate Flex errors using the data provided by the server.
Update the context configuration to use the new service

Using your new service implementation instead of the existing MasterDataServiceImpl is simply a matter of changing your context configuration.

Edits to the code below appear in bold.

```xml
<?xml version="1.0" encoding="utf-8"?>
xmlns:mds="com.initiate.composer.extension.service.mds.*"
xmlns:metadata="com.initiate.composer.services.metadata.*"
xmlns:security="com.initiate.composer.security.impl.*"
xmlns:config="com.initiate.composer.services.config.*">
<fx:Declarations>
<config:ConfigurationServiceImpl configUrl="composer.icc"/>
<mds:MyMasterDataServiceImpl/>
<!-- Session -->
<security:SessionManagerImpl/>
<security:SessionTimeoutHandlerImpl/>
<!-- Authentication -->
<security:LoginManagerImpl/>
<!-- use FormAuthenticationProvider to login in Flex -->
<security:FormAuthenticationProvider/>
<!-- use JSPFormAuthenticationProvider to login using JSP -->
</fx:Declarations>
</fx:Object>
```

Update the mds namespace location and change the tag to reference your service as in the example above.
Chapter 6. Deploying applications

When you have generated the IBM Initiate Master Data Service Composer application and verified that it functions, you are ready to deploy the application to an application server where it can be available to users.

About this task

For the application to run properly on the application server, set the connection parameters, then package and deploy the composer.war file as described in the topics.

The topics describe the deployment process for the Apache Tomcat, IBM WebSphere, and WebLogic Application Servers.

Setting Master Data Engine connection parameters for Tomcat, WebSphere, and WebLogic

In order for the IBM Initiate Master Data Service Composer applications that you generate to connect with the Master Data Engine, the application server needs the appropriate connection information.

About this task

The following instructions apply to the remote installation of Apache Tomcat, IBM WebSphere, or Oracle WebLogic. These instructions parallel the configuration you did for the local Tomcat installation: “Setting Master Data Engine connection parameters for the local Tomcat installation” on page 5.

As described in the topics that follow, edit the composersvcs.properties file, and save it in the appropriate location on your application server.

Setting Master Data Engine connection parameters for Tomcat

In order to enable the communication between your Tomcat installation and the IBM Initiate Master Data Service, configure connection parameters within catalina.properties and composersvcs.properties.

About this task

Note: The following instructions assume that Tomcat is not running on the same machine as the Master Data Engine. If it is, see “Setting Master Data Engine connection parameters for the local Tomcat installation” on page 5.

Note: This topic does not discuss user authentication. For information about authentication, see “Managing authentication for Initiate Composer applications” on page 37.

Procedure

1. Ensure that the IBM Initiate engine is running.
2. On the machine where Tomcat is running, navigate in the file system to the Tomcat installation’s conf directory.
3. Open catalina.properties in a text editor.
4. Verify that the entry beginning with shared.loader= reads:
   
   shared.loader=${catalina.base}/shared/classes,${catalina.base}/shared/lib/*.jar

5. Save and close catalina.properties.
6. Within the Tomcat installation directory, create a directory named shared, for example: C:\apache-tomcat-6.0.24\shared
7. Within the new shared directory, create a classes directory.
8. On the machine where IBM Initiate Master Data Service Composer is installed, navigate in the file system to the ...untime\bin\ directory within the Composer plug-in in your Workbench installation. For example:
   C:\Program Files\IBM\Initiate\WorkbenchX.X.0\plugins\com.initiatesystems.workbench.composer_X.X.0\runtime\bin\ 
9. Copy the composervcs.properties.example file.
10. Paste composervcs.properties.example into the ..\shared\classes directory you created.
11. Rename composervcs.properties.example to composervcs.properties.
12. Copy the composervcs.properties file.
13. Transfer the composervcs.properties file to the machine running Tomcat and paste it into the shared/classes/ directory you created on that machine.
14. Open composervcs.properties.
15. Uncomment the following lines by removing the initial pound signs (#) and set the values to reflect a valid host name and port for your installation:
   
   contextFactory.hostName=
   contextFactory.hostPort=

   For example, if your configuration runs the Master Data Engine at myMasterDataServer:16000, set the values as:
   contextFactory.hostName=myMasterDataServer
   contextFactory.hostPort=16000
16. Save and close the composervcs.properties file.
17. Restart Tomcat.

**Setting Master Data Engine connection parameters for WebSphere**

In order to enable the communication between your WebSphere installation and the IBM Initiate Master Data Service, set the connection parameters within the composervcs.properties configuration file. The following instructions assume that WebSphere is not running on the same machine as the IBM Initiate Master Data Service.

**About this task**

**Note:** This topic does not discuss user authentication. For information about authentication, see "Managing authentication for Initiate Composer applications" on page 37.

**Procedure**

1. Ensure that the IBM Initiate engine is running.
2. On the machine where IBM Initiate Master Data Service Composer is installed, navigate in the file system to the ...untime\bin\ directory within the Composer plug-in in your Workbench installation. For example:
   C:\Program Files\IBM\Initiate\Workbench\X.X.0\plugins\com.initiatesystems.workbench.composer\X.X.0\runtime\bin\composersvcs.properties.example
3. Create a copy of the composersvcs.properties.example file.
4. Rename the copy to composersvcs.properties.
5. Copy the composersvcs.properties file.
6. Connect to the machine where WebSphere is running.
7. Navigate on that machine to the properties directory for the appropriate server and profile. For example,
   \machinename\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\properties\composersvcs.properties
8. Paste composersvcs.properties into the properties directory.
9. Open composersvcs.properties for editing.
10. Uncomment the following lines by removing the initial pound signs (#) and set the values to reflect a valid host name and port for your installation:
   
   contextFactory.hostName=
   contextFactory.hostPort=

   For example, if your configuration and runs the Master Data Engine at myMasterDataServer:16000, set the values as:
   contextFactory.hostName=myMasterDataServer
   contextFactory.hostPort=16000
11. Save and close the composersvcs.properties file.
12. Restart the Composer web application.

**Setting Master Data Engine connection parameters for WebLogic**

In order to enable the communication between your WebLogic installation and the Master Data Engine, set the connection parameters within the composersvcs.properties configuration file. The following instructions assume that WebLogic is not running on the same machine as the Master Data Engine.

**About this task**

*Note:* This topic does not discuss user authentication. For information about authentication, see "Managing authentication for Initiate Composer applications" on page 37.

**Procedure**

1. Ensure that the IBM Initiate engine is running.
2. On the machine where IBM Initiate Master Data Service Composer is installed, navigate in the file system to the ...untime\bin\ directory within the Composer plug-in in your Workbench installation. For example:
   C:\Program Files\IBM\Initiate\Workbench\X.X.0\plugins\com.initiatesystems.workbench.composer\X.X.0\runtime\bin\composersvcs.properties.example
3. Create a copy of the composersvcs.properties.example file.
4. Rename the copy to composersvcs.properties.
5. Open composersvcs.properties for editing.
6. Uncomment the following lines by removing the initial pound signs (#) and set the values to reflect a valid host name and port for your installation:

```java
contextFactory.hostName=
contextFactory.hostPort=
```

For example, if your configuration runs the Master Data Engine at myMasterDataServer:16000, set the values as:

```java
contextFactory.hostName=myMasterDataServer
contextFactory.hostPort=16000
```

7. Save and close the `composersvcs.properties` file.

8. Connect to the machine where WebLogic is running.

9. Navigate on that machine to the `.../server/lib` directory for your WebLogic installation. For example,

   `/local/compusr1/Oracle/Middleware/wlserver_10.3/server/lib/`

10. If the `.../server/lib` directory does not already contain the `weblogic_sp.jar` configuration file, you must create it:

    a. Copy and paste the `composersvcs.properties` file into a temporary directory.
    
    b. Compress the contents of the temporary directory.

    Note: Do not compress the temporary directory itself. Include only the contents of the directory.

    c. Rename the compressed file to `weblogic_sp.jar`.
    
    d. Copy `weblogic_sp.jar` back to the `.../server/lib` directory.
    
    e. Start the Composer application.

11. If the directory does already contain the `weblogic_sp.jar` configuration file, you must extract the contents of the file, insert `composersvcs.properties`, and re-package the file:

    a. Create a copy of the `weblogic_sp.jar` file and rename the copy to `weblogic_sp.zip`.
    
    b. Using WinZip or similar extraction tool, extract `weblogic_sp.zip` into a temporary directory.
    
    c. Copy and paste the `composersvcs.properties` file into the temporary directory.
    
    d. Compress the contents of the temporary directory.

    Note: Do not compress the temporary directory itself. Include only the contents of the directory.
    
    e. Rename the compressed file to `weblogic_sp.jar`.
    
    f. Copy `weblogic_sp.jar` back to the `.../server/lib` directory.
    
    g. Start the Composer application.

---

**Setting Master Data Engine SSL connection parameters**

By default, communication with the Master Data Engine does not use SSL. You can enable SSL communication by editing the `composersvcs.properties` configuration file.
About this task

Note: The following instructions assume that you have created and configured the composersvcs.properties configuration file according to the instructions for Tomcat ("Setting Master Data Engine connection parameters for Tomcat" on page 71) or WebSphere ("Setting Master Data Engine connection parameters for WebSphere" on page 72).

Procedure

1. Ensure that the IBM Initiate engine is running.
2. Within the Tomcat installation directory, navigate to the \shared\classes directory, for example: C:\apache-tomcat-6.0.24\shared\classes
3. Open composersvcs.properties.
4. Uncomment the line `contextFactory.useSSL=false` by removing the initial pound sign (#) and set the value to `true`.
5. If your installation uses TLSv1 SSL rather than SSLv3, uncomment the line `contextFactory.sslVersion=SSLv3` and set the value to `TLSv1`.
6. By default, the application server does not verify the SSL certificate presented by the Master Data Engine. The certificate is used for encryption only. If you prefer that the application server verify the SSL certificate, uncomment the line `contextFactory.sslCertVerify=false` and set the value to `true`.
7. Save and close the composersvcs.properties file.
8. Restart the application server.

Configuring the default start page for your application

Once your application is ready for deployment and once you know the intended start page, update the `<c:redirect>` tag within the `index.jsp` file.

About this task

The `<c:redirect>` tag redirects the user's browser to an alternate URL. Specifying the `<c:redirect>` tag within `index.jsp` allows you to indicate where to direct the user once he or she has successfully logged in.

Procedure

1. Navigate to the webapps\composer directory for your application. For example: `C:\apache-tomcat-6.0.24\webapps\composer`
2. Open the `index.jsp` file for editing.
3. Uncomment the line `<%-- <c:redirect url="/YourProjectDir/YourWrapper.html"/> --%>`.
4. Edit the URL to point to your application. For example: `<c:redirect url="/DemoMainApplication.jsp"/>`.
5. Save and close the `index.jsp` file.

Manually packaging the composer.war file for deployment

Once you have successfully generated your IBM Initiate Master Data Service Composer application and tested it within the Flash perspective, you can manually package the application output within the composer.war file to prepare for deployment.
About this task

The instructions assume that you have successfully generated and tested your Composer application in the Flash perspective. See Chapter 1, “Getting started,” on page 1. As part of that process, Workbench outputs your application files to the local Tomcat ..\webapps\composer\projectname-debug folder. By unpacking the composer.war file, embedding the application files, and repackaging composer.war, you create a single file for deployment on your application server.

Procedure

1. On your local machine, create a temporary directory in the file system.
2. Navigate in the file system to the ..\runtime\bin\ directory within the Composer plug-in in your Workbench installation. For example:
   C:\Program Files\IBM\Initiate\WorkbenchX.X.0\plugins\com.initiatesystems.workbench.composer_X.X.0\runtime\bin\
3. Create a copy of the composer.war file and rename the copy to composer.zip.
4. Using WinZip or similar extraction tool, extract composer.zip into the temporary directory you created.
5. Within Workbench, change to the Flash perspective.
6. From the main menu, choose Project > Export Release Build.
7. In the Export Release Build window, choose Browse and set the value of Export to folder to the temporary directory where you extracted composer.zip.
8. Click Finish.
9. Confirm that the temporary directory now contains the projectname folder containing DemoMainApplication.html and other supporting files and directories.
10. Compress the contents of the temporary directory.

   Note: Do not compress the temporary directory itself. Include only the contents of the directory.
11. Rename the compressed file to composer.war.

Results

The composer.war file now contains both the Composer libraries and your application files in a single WAR file. To deploy composer.war to Tomcat, see “Deploying a Composer application to a J2EE Tomcat environment.” To deploy to WebSphere, see “Deploying an application to WebSphere” on page 79.

Deploying a Composer application to a J2EE Tomcat environment

Tomcat offers two options for deploying the composer.war file.

About this task

To deploy the composer.war file on Tomcat, you can:

- Copy composer.war directly to the Tomcat installation's webapps directory and restart Tomcat.
- Use the Tomcat Deployment Manager to install the composer.war.
Deploying via manual file copy and Tomcat restart

You can deploy the IBM Initiate Master Data Service Composer application by copying the composer.war file to the Tomcat webapps directory and restarting the application server.

**Procedure**
1. Navigate to the temporary directory where you created the composer.war file.
2. Copy the file into the Tomcat webapps directory. For example, `C:\apache-tomcat-6.0.24\webapps`
3. Stop and restart Tomcat. To do so, navigate at a command-line prompt to the `bin/` directory within your Tomcat installation and type `shutdown.bat`, and then `startup.bat`.

**Results**

*Note:* If you encounter any errors during Tomcat startup, verify that the Master Data Engine is running by browsing to the scripts/ directory within the Master Data Engine installation and running `madconfig ping_instance`. If the engine is running, next verify the accuracy of the property settings within the `composersvcs.properties` file. See [“Setting Master Data Engine connection parameters for the local Tomcat installation” on page 5](#). Finally, review the latest localhost and catalina logs in the Tomcat `log/` directory for errors or review the output from the Tomcat console.

Deploying using Tomcat Manager

You can use the Tomcat Manager to find and deploy the IBM Initiate Master Data Service Composer application.

**Procedure**
1. If it is not already started, start Tomcat. To do so, navigate at a command-line prompt to the `bin/` directory within your Tomcat installation and type `startup.bat`.
2. Log in to the Tomcat Manager. Typically, the Tomcat Manager runs at the URL: `http://localhost:8080/manager/html`

   *Note:* If you have not done so already, enable a Tomcat user with the manager role. To do so, edit `tomcat-users.xml` to add the manager role to an existing user as described in the Manager App HOWTO.
3. Scroll down to the **Deploy** section.
4. Within the WAR file to deploy section, click **Browse**.
5. Navigate to the temporary directory where you created the new composer.war file.
6. Select the file and click **OK**.

**Results**

*Note:* If you encounter any errors during Tomcat startup, verify that the Master Data Engine is running by browsing to the scripts/ directory within the Master Data Engine installation and running `madconfig ping_instance`. If the engine is running, next verify the accuracy of the property settings within the `composersvcs.properties` file. See [“Setting Master Data Engine connection parameters for the local Tomcat installation” on page 5](#).
Finally, review the latest localhost and catalina logs in the Tomcat log/ directory for errors or review the output from the Tomcat console.

### Testing the Tomcat deployment

After deploying the composer.war file to Tomcat, navigate to the Tomcat HTTP URL, followed by the composer context and DemoMainApplication.html.

**Procedure**

1. Navigate to DemoMainApplication.html. For example: http://localhost:8080/composer/myproject-debug/DemoMainApplication.html
2. Log in.
3. Verify that you see the generated web page.

### Debugging Flex

Setting up IBM Initiate Workbench for Flash debugging allows you to set breakpoints and step through Flex sources for IBM Initiate Master Data Service Composer.

**Procedure**

1. Ensure that Tomcat is running.
2. Open the Flash Debug perspective: **Window > Open Perspective > Other > Flash Debug.**
3. Open a blueprint .mxml file within one of the `projectname/composer/src/blueprints/blueprintname/` folders.
4. Set a breakpoint within the ActionScript section of the file by clicking within a line of ActionScript code and navigating in the main menu to Run > **Toggle Breakpoint**. (As shown in the menu, you can also use Ctrl+Shift+B to toggle breakpoints.)
5. Switch back to the Configurations perspective: **Window > Open Perspective > Other > Configuration.**
6. Within the `projectname/composer/src/` folder, right-click `DemoMainApplication.mxml` and choose **Debug As > Debug Configurations.**
7. On the Debug Configurations window, verify the settings and click the **Debug** button.

If you see a dialog message that indicates your version of the Flash Player is not a debugger, follow the link to the Adobe Flash Player downloads and download the appropriate Flash Player. Depending on your browser of choice, download either the **Windows Flash Player 10 Active X control content debugger** (for IE) or the **Windows Flash Player 10 Plugin content debugger** (for Netscape-compatible browsers). Install the new player and resume the steps above.

If you encounter errors when attempting to launch, you must resolve them before proceeding. Errors can occur if the .mxml files contain illegal characters such as parentheses or if the file names and attribute names are not unique across the project.

When you have successfully launched the application, the Flash Debug perspective opens a browser and displays the application.
Deploying an application to WebSphere

Use the WebSphere Integrated Solutions Console to create and install a new application.

Procedure

1. Open a browser and navigate to the WebSphere Integrated Solutions Console web application, for example: https://mymachine:9043/ibm/console/logon.jsp
2. Log in and navigate to Applications > New Application.
3. Choose New Enterprise Application.
4. Choose Local file system, and browse in the file system to select the composer.war that you created in “Manually packaging the composer.war file for deployment” on page 75, for example, C:\Program Files\IBM\Initiate\WorkbenchX.X.0\plugins\com.initiatesystems.workbench.composer_X.X.0\runtime\bin
5. Click Next.
6. Choose Fast Path and click Next.
7. In the Select installation options window, accept the defaults and click Next.
8. In the Map modules to servers window, set a checkmark next to your module, and click Next.
9. In the Map virtual hosts for Web modules window, set a checkmark next to your web module, and click Next.
10. In the Map context roots for Web modules window, provide the Context root (for example, /composer), and click Next.
11. Verify the settings on the Summary window and click Finish. WebSphere installs the application. Look for the message Application composer_war installed successfully.
12. If you are satisfied with the installation, click Save to save your changes directly to the master configuration.
13. Still within the WebSphere Integrated Solutions Console, navigate to Applications > Application Types > WebSphere enterprise applications.
14. Set a checkmark next to composer_war.

Note: Before you start the Composer application in the next step, verify that you have configured your composersvcs.properties file and have started the IBM Initiate Master Data Service Engine. See “Setting Master Data Engine connection parameters for WebSphere” on page 72.
15. Click Start. When WebSphere has processed the command, you should see the message: Application composer_war on server servername and node nodename started successfully.
16. Browse to the application to verify that it is running, for example: http://mymachine:9080/composer/myproject-debug/DemoMainApplication.html

Results

Note: If you encounter any errors during WebSphere startup, verify that the Master Data Engine is running by browsing to the scripts directory within the Master Data Engine installation and running madconfig ping_instance. If the engine is running, next verify the accuracy of the property settings within the...
Deploying an application to WebLogic

Use the WebLogic Administration Console to create and install a new application.

**Procedure**

1. Open a browser and navigate to the WebLogic Administration Console, for example: http://mymachine:7001/console/login/LoginForm.jsp
2. Log in and navigate to **Configure applications**.
3. Click the **Install** button.
4. Choose the link to **upload your file(s)**.
5. For the **Deployment Archive**, browse to the directory containing the composer.war file that you created in "Manually packaging the composer.war file for deployment" on page 75, for example, C:\Program Files\IBM\Initiate\WorkbenchX.X.X.0\plugins\com.initiatesystems.workbench.composer_X.X.0\runtime\bin.
6. Click **Next**.
7. Confirm that the composer.war file has been uploaded successfully to the correct upload directory.
8. Click **Next**.
9. On the next page, set the radio button to **Install this deployment as an application**.
10. Click **Next**.
11. Accept the defaults on the next page and click **Next**.
12. Again, accept the defaults on the next page and click **Finish**.
13. On the **Configuration** tab, specify composer as the **Context Root** and click **Save**.
14. Click **OK** to save the deployment plan.
15. Confirm that all changes have been activated and that no restarts are necessary.
16. Browse to the application to verify that it is running, for example: http://mymachine:9080/composer/myproject-debug/DemoMainApplication.html

**Results**

If you encounter any errors during WebLogic startup, verify that the Master Data Engine is running by browsing to the scripts directory within the Master Data Engine installation and running madconfig ping_instance. If the engine is running, next verify the accuracy of the property settings within the **composersvcs.properties** file. See "Setting Master Data Engine connection parameters for WebLogic" on page 73. Finally, review the latest logs in the WebLogic log directory.
Chapter 7. Reference

These topics offer quick reference and configuration information about IBM Initiate Master Data Service Composer components, the ActionScript API, common Composer terms, and more.

For information about the concepts of components, blueprints, and so on, see "Introduction" on page v.

Accessing the ActionScript API documentation (ASDoc)

IBM Initiate Master Data Service Composer includes HTML-based documentation for commands, components, and so on using ASDoc.

About this task

After you have extracted the composerdocs.zip file as described below, users can view the ASDoc documentation using a browser.

Procedure

1. Navigate in the file system to the ...untime\doc\ directory within the Composer plug-in in your IBM Initiate Workbench installation. For example:
   C:\Program Files\IBM\Initiate\Workbench9.7.0\plugins\com.initiatesystems.workbench.composer_9.7.0\runtime\doc\  
2. Copy the composerdocs.zip file.
3. Extract composerdocs.zip to a location of your choosing.
4. In the new folder, navigate to the asdoc directory and open index.html in browser.
5. To view configuration options (Public Properties) for the Composer components, click to open the com.initiate.composer.components package.

Component reference

Per-component topics provide an easy reference to the main IBM Initiate Master Data Service Composer components including a short description, configuration options, and a summary of events.

Because the Composer components are delivered within a Flex SWC library, the component code is not exposed. Instead you manipulate the behavior of components by setting parameter values for a component’s corresponding PM file. Within application context files, configure PM (presentation model) parameters to alter the behavior of the Composer components. To do so, navigate to a context file within your application and set your cursor after the "PM" in the <components:*PM/> tag. Hitting the spacebar opens the list of configuration options.

In addition to the components, Composer includes a set of pre-configured blueprints which assemble several components together to fulfill a particular function. Blueprints establish how the components are wired together to manage the events flowing from one component to another. See "Initiate Composer blueprints" on page ix.
Search Form component

The component is a simple form allowing end users to conduct searches based on member type or entity type attributes you specify in the IBM Initiate Master Data Service Composer Configuration Editor.

By default, the form includes Search and Clear buttons. The Composer Configuration Editor allows you to choose whether the form searches for members or entities.

Behavior

Clicking the Search button triggers SearchEvent with a selector of entitySearch or recordSearch. SearchCommand captures the event and routes it to the IBM Initiate Master Data Service Engine. The results are passed via the SearchScreenPresenter to the SearchResultsPM.

Associated PM

SearchFormPM

Configuration options

The Composer ASDoc offers detailed descriptions of the Search Form component configuration options. See "Accessing the ActionScript API documentation (ASDoc)" on page ix.

Outbound events

- SearchEvent.SEARCH_TYPE_ENTITY where SearchEvent is the event class and SEARCH_TYPE_ENTITY is the event type, also called the selector.
- SearchEvent.SEARCH_TYPE_RECORD where SearchEvent is the event class and SEARCH_TYPE_RECORD is the event type, also called the selector.

Ensure that any application you create includes handlers to manage all outbound events.

Context requirements

Inject instances of the following into the context for any application you create that uses the Search Form component:

- com.initiate.composer.commands.SearchCommand
- com.initiate.composer.components.SearchFormPM

To use a custom command or component inject the appropriate reference to that command in the context file.

Blueprint implementation

The Search Form component appears within the following blueprints:

- Quick View
- Multi-Domain Entity Search (within TabbedEntitySearchScreenContext.mxml)

Search Results component

The component is a tabular display of search results in descending order according to a matching score.

The table columns are based on the member type attributes that you specify in the IBM Initiate Master Data Service Composer Configuration Editor.

You can configure the Search Results component to include the matching score for each result displayed. See "Displaying the score for search results" on page 17.
Behavior
Clicking a record or entity triggers SearchResultSelectedEvent with a selector of entitySearchResultSelected or recordSearchResultSelected.

Associated PM
SearchResultsPM

Configuration options
The Composer ASDoc offers detailed descriptions of the Search Results component configuration options. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

Outbound events
- SearchResultSelectedEvent.RECORD_SEARCH_RESULT_SELECTED where SearchResultSelectedEvent is the event class and RECORD_SEARCH_RESULT_SELECTED is the event type, also called the selector.
- SearchResultSelectedEvent.ENTITY_SEARCH_RESULT_SELECTED where SearchResultSelectedEvent is the event class and ENTITY_SEARCH_RESULT_SELECTED is the event type, also called the selector.

Ensure that any application you create includes handlers to manage all outbound events.

Context requirements
Inject instances of the following into the context for any application you create that uses the Search Results component:
com.initiate.composer.components.SearchResultsPM

To use a custom command or component inject the appropriate reference to that command in the context file.

Blueprint implementation
The Search Results component appears within the following blueprints:
- Quick View
- Multi-Domain Entity Search (within TabbedEntitySearchScreenContext.mxml)

Details component
For a member (also often called a record) or an entity that you select from a Search Results list, a display of all the fields you specify in the IBM Initiate Master Data Service Composer Configuration Editor.

RecordDetailTablePM and EntityDetailTablePM inherit properties and methods from DetailTablePM. However, events are defined on RecordDetailTablePM and EntityDetailTablePM.

If the user is authorized to do so and if the editable property is set to true, the user can edit entities. If editing is enabled, you must also configure the source property, which determines to which database source Composer writes any edits. See “Configuring authentication for editing entities and adding members” on page 38 and “Configuring a source” on page 10.
Note: With the current release, when editing an entity or a record attribute that is a MEMATTR attribute, Composer generates an error message if a user clears an existing value. In the engine error log, the message indicates that it cannot insert the value NULL into the column.

The Design View within Flash Builder displays an empty panel for Details View components. The panel populates at run time.

Behavior
Displays the details for a record or entity based on events flowing from the Search Results component. If the component has been enabled for editing and if the user has the required authorization, completing an edit to an entity triggers either an UpdateRecordEvent or an UpdateEntityEvent (depending on whether the component is configured to display record details or entity details). The UpdateRecordCommand / UpdateEntityCommand captures the event and routes it to the IBM Initiate Master Data Service Engine. The results are passed via a presenter back to the RecordDetailTablePM / EntityDetailTablePM.

Associated PM
RecordDetailTablePM or EntityDetailTablePM

Configuration options
The Composer ASDoc offers detailed descriptions of the Details component configuration options. See “Accessing the ActionScript API documentation” on page ix.

Outbound events
- UpdateEntityEvent.UPDATE_ENTITY with UpdateEntityEvent is the event class and UPDATE_ENTITY is the event type, also called the selector.
- A Flash event (flash.events.Event) with an event type (selector) of entityChanged.

Ensure that any application you create includes handlers to manage all outbound events.

Context requirements
Inject instances of the following into the context for any application you create that uses the Details component:
com.initiate.composer.components.EntityDetailTablePM

To use a custom command or component inject the appropriate reference to that command in the context file.

Blueprint implementation
The Details component appears within the following blueprints:
- for RecordDetailTablePM:
  - Multi-Domain Entity Search (within RecordDetailTabContext.mxml)
- for EntityDetailTablePM:
  - Quick View
  - Multi-Domain Entity Search (within EntityDetailTabContext.mxml)

Add Form component
The component allows users to add a member to a source specified within the IBM Initiate Master Data Service Composer Configuration Editor.
Users authorized to add members see a link to **Add Member Record** at the top right of the application. The link does appear for users who are not authorized. For more information, see “Configuring authentication for editing entities and adding members” on page 38.

When a user adds a record, the application creates an entity which contains the single record the user has created. (Entities are simply containers for one or more records.) For more information about, adding records, see “Configuring a source” on page 10.

**Behavior**

Clicking the **Add** button triggers **AddEvent** with a selector of **addRecord**. **AddCommand** captures the event and routes it to the IBM Initiate Master Data Service Engine. The confirmation of the addition is passed via a presenter to the **AddRecordFormPM**.

**Associated PM**

**AddRecordFormPM**

**Configuration options**

The Composer ASDoc offers detailed descriptions of the Add Form component configuration options. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

**Outbound events**

- **AddEvent.ADD_TYPE_RECORD** where **AddEvent** is the event class and **ADD_TYPE_RECORD** is the event type, also called the selector.

Ensure that any application you create includes handlers to manage all outbound events.

**Context requirements**

Inject instances of the following into the context for any application you create that uses the Add Form component:

- `com.initiate.composer.commands.AddCommand`
- `com.initiate.composer.components.AddRecordFormPM`

To use a custom command or component inject the appropriate reference to that command in the context file.

**Blueprint implementation**

The Add Form component appears within the following blueprint:

- Multi-Domain Entity Search (within **TabbedEntitySearchScreenContext.mxml**)

In addition to the `<components:AddRecordFormPM ...>` tag in the context file, the following code within the **TabbedEntitySearchScreen.mxml** file displays a popup that either confirms the add request or displays a fault if the request fails:

```xml
[CommandResult(scope="local")]
public function handleAddResult(result:ResultEvent, trigger:AddEvent):void
{
    Alert.show("Record successfully added.", "Request Completed");
}

[CommandError(scope="local")]
public function handleAddFault(fault:FaultEvent,
```
Relationship Viewer component

The component is a graphical display of connections between entities.

By default, the Relationship Viewer component simply displays the connections between the entities. The component also includes a control panel with which users can choose levels of zoom and scale, and by which they can choose how many degrees of separation to display. The control panel is hidden by default. To show the control panel, set the `showControlPanel` property to true.

See also: "Assigning icons for display on the Relationships tab" on page 13.

Behavior

For the Multi-Domain Entity Search blueprint, selecting an entity in the Search Results component populates a network display with the selected entity at the center and related entities connected to that entity by lines of relationship. You can populate the Relationship Viewer by setting the root entity submitted to the PM. Doing so initializes the graph, centered on the entity.

If at runtime no relationships have been defined for the specified entity, the Relationship Viewer does not render. Additionally, when the default Composer configuration was created, if the Hub configuration contains no defined relationships for the entity, the Relationship Viewer will not be available in the generated application. Ensure that your Hub configuration is complete before creating the default Composer configuration. See "Creating the default Initiate Composer configuration" on page 7.

Associated PM

RelationshipViewerPM

Configuration options

The Composer ASDoc offers detailed descriptions of the Relationship Viewer component configuration options. See "Accessing the ActionScript API documentation (ASDoc)" on page ix.

Outbound events

- Right-clicking an entity in the Relationship Viewer and choosing View Entity fires the `RelatedEntitiesEvent.ENTITY_SELECTED` event, where `RelatedEntitiesEvent` is the event class and `ENTITY_SELECTED` is the event type, also called the selector. The event populates a separate Details view for the selected entity.
- Clicking an entity in the Relationship Viewer fires the `RelatedEntitiesEvent.GET_RELATED_ENTITIES` event, where `RelatedEntitiesEvent` is the event class and `GET_RELATED_ENTITIES` is the event type, also called the selector.

Ensure that any application you create includes handlers to manage all outbound events.

Context requirements

Inject instances of the following into the context for any application you create that uses the Relationship Viewer component:

- `com.initiate.composer.commands.RelatedEntitiesCommand`
- `com.initiate.composer.components.relViewer.RelationshipViewerPM`
To use a custom command or component inject the appropriate reference to that command in the context file.

**Blueprint implementation**

The Relationship Viewer component appears within the following blueprint:
- Multi-Domain Entity Search (within EntityDetailTabContext.mxml)

**Related Entities component**

The component is a read-only tabular display of entities related to the entity the user has selected in a Search Results component.

**Behavior**

For the Multi-Domain Entity Search blueprint, selecting an entity in the Search Results component launches a Details tab, which includes a pane of details for the entity on the left, and on the right a set of tabs, each of which contains the Related Entities component. Each tab in the display represents a relationship type such as **boss** or **friend**. The relationship types available depend on your data model. To configure the attributes displayed in each tab, define a default attribute set for each entity type. See "Assigning a default attribute set" on page 13.

If the user clicks an entity within a tab, an additional Details tab opens for that selected entity.

If there are no defined relationships found at runtime for the specified entity, the Related Entities component does not render. Additionally, when the default Composer configuration was created, if no defined relationships for the entity are found in the Hub configuration, the Related Entities component will not be available in the generated application. Ensure that your Hub configuration is complete before creating the default Composer configuration. See "Creating the default Initiate Composer configuration" on page 7.

**Associated PM**

RelatedEntitiesListPM

**Configuration options**

The Composer ASDoc offers detailed descriptions of the Related Entities component configuration options. See "Accessing the ActionScript API documentation (ASDoc)" on page ix.

**Outbound events**

- **RelatedEntitiesEvent.GET_RELATED_ENTITIES**
  where **RelatedEntitiesEvent** is the event class and **getRelatedEntities** is the event type, also called the selector. A RelatedEntitiesEvent event with the **GET_RELATED_ENTITIES** selector fires automatically when the component is called in order to populate the component.

- **RelatedEntitiesEvent.ENTITY_SELECTED**
  where **RelatedEntitiesEvent** is the event class and **entitySelected** is the event type, also called the selector. A RelatedEntitiesEvent event with the **ENTITY_SELECTED** selector fires when the user clicks an entity within a Related Entities component. For the Multi-Domain Entity Search blueprint, the event launches a new Details tab.

Ensure that any application you create includes handlers to manage all outbound events.
Context requirements
Inject instances of the following into the context for any application you create that uses the Related Entities component:
com.initiate.composer.commands.RelatedEntitiesCommand
com.initiate.composer.components.RelatedEntitiesListPM

To use a custom command or component inject the appropriate reference to that command in the context file.

Blueprint implementation
The Related Entities component appears within the following blueprint:
• Multi-Domain Entity Search (within RelatedEntityTabContext.mxml)

Record List component
The component is a display of the individual records associated with a particular entity.

Behavior
This component is embedded within the Search Results component.

Associated PM
EntityRecordListPM

Configuration options
The Composer ASDoc offers detailed descriptions of the Record List component configuration options. See "Accessing the ActionScript API documentation (ASDoc)" on page ix.

Outbound events
• RetrieveItemSelectedEvent.RECORD_SELECTED where
  RetrieveItemSelectedEvent is the event class and RECORD_SELECTED is the event type, also called the selector. Retrieves details for the item when the user clicks a row in the list.
• RetrieveRecordListEvent.RETRIEVE_RECORD_LIST where
  RetrieveRecordListEvent is the event class and RETRIEVE_RECORD_LIST is the event type, also called the selector. The event fires automatically when rendering the component in order to populate the list of members based on the entity.

Ensure that any application you create includes handlers to manage all outbound events.

Context requirements
Inject instances of the following into the context for any application you create that uses the Related Entities component:
com.initiate.composer.commands.RecordListCommand
com.initiate.composer.components.EntityRecordListPM

To use a custom command or component inject the appropriate reference to that command in the context file.

Blueprint implementation
The Record List component appears within the following blueprint:
• Multi-Domain Entity Search (within EntityDetailTabContext.mxml)
Command reference

IBM Initiate Master Data Service Composer commands handle events that need to execute against the Master Data Engine. Results are returned asynchronously.

Commands have method signatures that associate them with particular events and selectors. By injecting a command into a context from which an event is dispatched, the command is automatically registered as a handler for the event.

Parsley handles the results of a command and calls all matching [CommandResult] and [CommandFault] handlers.

AddCommand
A Parsley command for adding records in the service layer.

The Add Form component uses AddCommand to add records. For property, constructor, and method details, refer to the IBM Initiate Master Data Service Composer ActionScript documentation. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

For detail about the Add Form component, see “Add Form component” on page 84.

RecordListCommand
A Parsley command for retrieving a list of records in the service layer.

The Record List component uses RecordListCommand to retrieve the record list. For property, constructor, and method details, refer to the IBM Initiate Master Data Service Composer ActionScript documentation. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

For detail about the Record List component, see “Record List component” on page 88.

RelatedEntitiesCommand
A Parsley command for retrieving a list of related entities in the service layer.

The Relationship Viewer component and the Related Entities component each use RelatedEntitiesCommand to retrieve the list of related entities. For property, constructor, and method details, refer to the IBM Initiate Master Data Service Composer ActionScript documentation. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

For detail about the Relationship Viewer component, see “Relationship Viewer component” on page 86. For detail about the Related Entities component, see “Related Entities component” on page 87.

SearchCommand
A Parsley command for performing searches against the service layer.

The Search Form component uses SearchCommand to initiate searches. For property, constructor, and method details, refer to the IBM Initiate Master Data Service Composer ActionScript documentation. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.
For detail about the Search Form component, see “Search Form component” on page 82.

**UpdateEntityCommand**

A Parsley command for updating an entity in the service layer.

The Details component uses UpdateEntityCommand to update entities. For property, constructor, and method details, refer to the IBM Initiate Master Data Service Composer ActionScript documentation. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

For detail about the Details component, see “Details component” on page 83.

**UpdateRecordCommand**

A Parsley command for updating a record in the service layer.

The Details component uses UpdateRecordCommand to update records. For property, constructor, and method details, refer to the IBM Initiate Master Data Service Composer ActionScript documentation. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

For detail about the Details component, see “Details component” on page 83.

---

**REST API reference**

Composer uses a subset of the RESTful service calls available with the IBM Initiate Master Data Service.

The Composer ActionScript API provides a façade for interacting with the IBM Initiate Master Data Service Engine. That API should answer most of your needs. See “Accessing the ActionScript API documentation (ASDoc)” on page ix.

The topic lists the REST services in use by Composer. The specific calls made available are intended to give developers the tools necessary to contact the IBM Initiate Master Data Service Engine using the Composer components and to populate the controls. Specifically, the calls allow Composer components to:

- Search for entities or records that match particular criteria
- Create, retrieve, update, and delete records
- Retrieve entity objects by the entity’s own ID or by the ID of any of its member records
- Retrieve (as an XML object) the application’s core system metadata
- Manage application view metadata for source and attribute filtering
- Retrieve the list of all member records associated with a particular entity ID
- Edit and save the parameters for an entity

All calls return an AsyncToken, which provides a place to set additional or token-level data for asynchronous remote procedure calls. This is the primary object used to communicate results and faults to the caller for asynchronous RPC operations.

To verify that the REST service is running correctly for Composer browse to http://host:port/composer/services/metadata. The host and port should correspond to the application server host and port where the composer.war file is
deployed. If you are using JSP-based authentication, the web page prompts you to log in. If you have deployed Composer to a context other than composer, substitute your context in the URL as appropriate.

Similarly, you can verify that the core REST service is running on the engine by browsing to http://host:7378/madappsvcs/rest/metadata. The host should correspond to the server host where the IBM Initiate Master Data Service Engine is deployed. (Note that the Composer-specific URL substitutes services for rest in the path.) Composer deploys another instance of the same services provided by the core REST services. You do not need to deploy the core REST services on the engine to use Composer.

To access the RESTdoc for the core MDS REST services, go to the following URL: http://host:7378/madappsvcs/, for example: http://localhost:7378/madappsvcs/. Note that the documentation available in the ASDoc supplements the RESTdoc with information specific to Composer.

See "Configuring JSP-based authentication" on page 40 and "Customizing the login panel (Flex-based authentication)" on page 40.

createRecord

REST service call for creating a record.

The operation fails if a record with the identical record ID exists.

Parameters

- Query parameters:
  - updateMode: (Optional) style of attribute data specified on the input record
  - entPriority: (Optional) entity management priority value
- Entity parameter: record: the record to be created
- Returns: AsyncToken

deleteRecord

REST service call for deleting a record.

Parameters

- recordId: Identifier for the record to be deleted
- drop: (Optional) Indicates whether to drop or delete the record. Default is false, indicating delete.
- Returns: AsyncToken

deleteViewMetaData

REST service call for deleting the specified metaData view.

Parameters

- viewName: the metaData view to delete
- Returns: AsyncToken

See "Source and Attribute Filtering with the REST API" on page 95.
**getAppMetaData**
REST service call for retrieving core system metadata and application-specific metadata.

If the application name is specified, the returned *metaData* is populated with the application-specific *metaData* (labels, defaults, and so on). The *metaData* is returned as an XML object.

**Parameters**
- Query parameters:
  - appName: (Optional)
  - locale: (Optional)
- Returns: AsyncToken

**getContext**
REST service call for getting the user context for the currently logged in user.

**Parameters**
The method has no parameters.
- Returns: AsyncToken

**getEntityByEntityId**
REST service call for retrieving an entity using the Entity ID.

**Parameters**
- entityId: the entity identifier to retrieve
- cvwName: (Optional) composite view name, null to ignore
- viewName: (Optional) view name for attribute filter, null to ignore
- memStat: (Optional) member status filter, null to ignore
- recStat: (Optional) record status filter, null to ignore
- Returns: An entity value asynchronously through the AsyncToken.

**getEntityByRecordId**
REST service call for retrieving an entity using one of the entity's member records

**Parameters**
- recordId: the record identifier for the entity to retrieve
- entType: the entity type to return
- cvwName: (Optional) composite view name, null to ignore
- viewName: (Optional) view name for attribute filter, null to ignore
- memStat: (Optional) member status filter, null to ignore
- recStat: (Optional) record status filter, null to ignore
- Returns: An entity value asynchronously through the AsyncToken.

**getRecord**
REST service call for retrieving a record using its record identifier.

**Parameters**
- recordId: the record identifier for the record to retrieve
- cvwName: (Optional) composite view name, null to ignore
- viewName: (Optional) view name for attribute filter, null to ignore
- memStat: (Optional) member status filter, null to ignore
- recStat: (Optional) record status filter, null to ignore
- Returns: A record value asynchronously through the AsyncToken.

**getRecordListByEntityId**

REST service call for retrieving a list of records using an entity identifier.

**Parameters**

- entityId: the entity identifier to retrieve
- cvwName: (Optional) composite view name, null to ignore
- viewName: (Optional) view name for attribute filter, null to ignore
- memStat: (Optional) member status filter, null to ignore
- recStat: (Optional) record status filter, null to ignore
- Returns: The list asynchronously through the AsyncToken.

**getRelatedEntityList**

REST service call for getting the related entity records for the specified entity ID.

**Parameters**

- entityId: the EntityId
- relType: gives relationshipType name
- relDirection: (Optional) gives relationshipDirection
- cvwName: (Optional) composite view name
- viewName: (Optional) view name for attribute filter
- memStat: (Optional) member status filter
- recStat: (Optional) record status filter
- Returns: The ArrayCollection of Entity asynchronously through the AsyncToken.

**getViewMetaData**

REST service call for getting the metaData for the given viewName.

**Parameters**

- viewName: the metaData view to get
- Returns: An XML representation of the view metaData is returned asynchronously through the AsyncToken.

See ["Source and Attribute Filtering with the REST API" on page 95](#).

**listViewNames**

REST service call for getting the list of available view names for the given member type. If the member type filter is null, returns all views.

**Parameters**

- memType: (Optional) the member type
- Returns: An ArrayCollection of Strings asynchronously through the AsyncToken.
See “Source and Attribute Filtering with the REST API” on page 95.

**putViewMetaData**

REST service call for persisting a metaData view. If a view is already persisted with the given viewName, it will be overwritten.

**Parameters**

- **viewName**: the view name for this metaData object
- **metaData**: the metaData object to persist, in XML

**Returns**: AsyncToken

See “Source and Attribute Filtering with the REST API” on page 95.

**searchForEntityList**

REST service call for searching for entities that match the criteria in a given record.

**Parameters**

- **criteria**: the record that contains the search criteria. (This is a generic record in the Flex API.)
- **entType**: the entity type to return
- **minScore**: (Optional) the minimum score for the search
- **maxResults**: (Optional) the maximum number of results to return with the query
- **cvwName**: (Optional) composite view name, null to ignore
- **viewName**: (Optional) view name for attribute filter, null to ignore
- **memStat**: (Optional) member status filter, null to ignore
- **recStat**: (Optional) record status filter, null to ignore

**Returns**: A list of EntitySearchResult objects through the AsyncToken.

**searchForRecordList**

REST service call for searching for records that match the criteria in a given record.

**Parameters**

- **criteria**: the record that contains the search criteria. (This is a generic record in the Flex API.)
- **entType**: the entity type to return
- **minScore**: (Optional) the minimum score for the search
- **maxResults**: (Optional) the maximum number of results to return
- **cvwName**: (Optional) the composite view name, null to ignore
- **viewName**: (Optional) view name for attribute filter, null to ignore
- **memStat**: (Optional) member status filter, null to ignore
- **recStat**: (Optional) record status filter, null to ignore

**Returns**: A list of RecordSearchResult objects through the AsyncToken.

**updateEntity**

REST service call for updating an entity.

For more information about update strategies, see “Choosing how to link records when making edits to entities” on page 42.
Parameters

- entity: the Entity to be updated or modified
- modifySource: the source for the application to modify
- updateEntity: (Optional) the entity update strategy. Default is ModifySourceHardLink. Alternative value is ModifySourceExplicit.
- updateMode: (Optional) the update mode. The default is AttrComp.
- Returns: AsyncToken

updateRecord

REST service call for updating a record.

The operation fails if a record with the identical record ID does not exist.

Parameters

- record: the record to be updated or modified.
- updateMode: (Optional) style of attribute data specified on the input record.
- entPriority: (Optional) entity management priority value
- Returns: AsyncToken

Source and Attribute Filtering with the REST API

The introduction of the REST API brings with it a new paradigm in performing filtering by source and/or attribute.

Older API methods allowed the caller to include an array of source codes or attribute codes to be included in the resulting method results. To reduce the number of parameters passed in the REST API calls the concept of a view metadata was introduced. The view metadata contains the attribute codes and/or source codes used to filter return values. It also allows the client to define a view once and have it applied to multiple method calls.

It is important to note that view metadata and composite views have some similarities but are in fact two different mechanisms to control the results returned from the MDS. Composite views and view metadata can be used together or independently.

The REST API and Composer IBM Initiate Master Data Service provide four methods to manage view metadata. These methods provide the ability to list the defined view names as well as perform normal CRUD operations on the view metadata. The four methods are: "listViewNames" on page 93, "deleteViewMetaData" on page 91, "getViewMetaData" on page 93, and "putViewMetaData" on page 94.

The view metadata objects used by these methods are structured objects (XML in ActionScript and JSON in JavaScript) and are identical in structure to the application metadata returned from the getAppMetaData method. However, the view metadata only uses a subset of the structure. To help developers build view metadata objects a convenience method, buildViewMetaData, was added to the MetaData Service API. This method accepts an array of source codes, an array of attribute codes, or both and returns the structured object ready for use with the MDS view metadata methods.

As an example, if you wanted to filter record or entity search results to a subset of sources you could build a view that restricts the results to only the sources you
want to see. You would do that by passing the array of source codes to include to the MetaData Service's buildViewMetaData method. It will return the view metadata object for you to use with putViewMetaData. When you call putViewMetaData, you need to provide a view name along with the view metadata. If view metadata already exists with that name, the existing view metadata will be replaced. Finally the view name will be included as the viewName parameter to the searchForRecordList or searchForEntityList methods.

Of course this filtering could be done statically at design time with predefined views being constructed ahead of time by one or more components or it could be defined dynamically at runtime. If this view filtering is done dynamically at runtime, don't forget that you will need to wait for the view to be created on the server before using it. Also remember to include the view name with other method calls like getRecordListByEntityId if you want to keep propagating the source code filter. This may require you to extend existing components and events to include the viewName property so that it can be passed on to downstream components.

It is important to note that by default view metadata is stored in the user's session and is discarded at the end of the user's session. This means that when the user logs out or their session times out, any view metadata stored in the session is lost and must be reconstructed before it is used again. It also means that the names used to identify the view metadata only need to be unique within the context of the user's session.
Notices and trademarks

This information was developed for products and services offered in the U.S.A.

Notices

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web
sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to
IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**Trademarks**

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Other product and service names might be trademarks of IBM or other companies.


The following terms are trademarks or registered trademarks of other companies:

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Contacting IBM

You can contact IBM for customer support, software services, product information, and general information. You also can provide feedback to IBM about products and documentation.

The following table lists resources for customer support, software services, training, and product and solutions information.

Table 1. IBM resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description and location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Support Portal</td>
<td>You can customize support information by choosing the products and the topics that interest you at [<a href="http://www.ibm.com/support/">www.ibm.com/support/</a>] entry/portal/Overview/Software/Information_Management/IBMInitiate_Master_Data_Service</td>
</tr>
<tr>
<td>Software services</td>
<td>You can find information about software, IT, and business consulting services, on the solutions site at [<a href="http://www.ibm.com/businesssolutions/">www.ibm.com/businesssolutions/</a>]</td>
</tr>
<tr>
<td>My IBM</td>
<td>You can manage links to IBM web sites and information that meet your specific technical support needs by creating an account on the My IBM site at [<a href="http://www.ibm.com/account/">www.ibm.com/account/</a>]</td>
</tr>
<tr>
<td>Training and certification</td>
<td>You can learn about technical training and education services designed for individuals, companies, and public organizations to acquire, maintain, and optimize their IT skills at [<a href="http://www.ibm.com/software/sw-training/">http://www.ibm.com/software/sw-training/</a>]</td>
</tr>
<tr>
<td>IBM representatives</td>
<td>You can contact an IBM representative to learn about solutions at [<a href="http://www.ibm.com/connect/ibm/us/en/">www.ibm.com/connect/ibm/us/en/</a>]</td>
</tr>
</tbody>
</table>

Providing feedback

The following table describes how to provide feedback to IBM about products and product documentation.

Table 2. Providing feedback to IBM

<table>
<thead>
<tr>
<th>Type of feedback</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product feedback</td>
<td>You can provide general product feedback through the Consumability Survey at [<a href="http://www.ibm.com/software/data/info/consumability-survey">www.ibm.com/software/data/info/consumability-survey</a>]</td>
</tr>
</tbody>
</table>
Table 2. Providing feedback to IBM (continued)

<table>
<thead>
<tr>
<th>Type of feedback</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation feedback</td>
<td>To comment on the information center, click the Feedback link on the top right side of any topic in the information center. You can also send comments about PDF file books, the information center, or any other documentation in the following ways:</td>
</tr>
<tr>
<td></td>
<td>• Online reader comment form: <a href="http://www.ibm.com/software/data/rcf/">www.ibm.com/software/data/rcf/</a></td>
</tr>
<tr>
<td></td>
<td>• E-mail: <a href="mailto:comments@us.ibm.com">comments@us.ibm.com</a></td>
</tr>
</tbody>
</table>