Contents

Chapter 1: Tutorial Overview
About the sample ................................................................. 1
Tutorial modules ................................................................. 2

Chapter 2: Getting Started
Download tutorial files ......................................................... 3
Install required software ....................................................... 3
Create a database ............................................................... 4
Create a web application ...................................................... 4
Create the Context ............................................................. 6
Start the server ................................................................. 7
Create a project in Flash Builder .......................................... 7
Connect to the RDS server .................................................. 8
Next steps ......................................................................... 9

Chapter 3: Create the Data Model
Setting up ........................................................................ 11
Review the application data ............................................... 11
Create the model ............................................................... 12
Create the database tables ................................................... 20
Next steps ........................................................................ 21

Chapter 4: Developing the Data-Centric Client Application
Setting up ........................................................................ 22
Generate ActionScript classes .............................................. 25
Populate the case list .......................................................... 26
Filter the case list .............................................................. 28
Extend CustCaseService to extract case metrics .................. 30
Generate a model-driven form .............................................. 31
Next Steps ........................................................................ 35

Chapter 5: Server-Side Configuration and Programming
Setting up ........................................................................ 36
Integrate with Spring security ............................................. 37
Customize the assembler classes ....................................... 45
Next steps ........................................................................ 49

Chapter 6: Packaging and Deploying the Application
Setting up ........................................................................ 50
Build and package the application ..................................... 52
Deploy the application ....................................................... 54
Next steps ........................................................................ 55

Last updated 6/29/2010
Chapter 1: Tutorial Overview

The Creating a LiveCycle Data Services ES2 Application tutorial guides you through the end-to-end process of developing a LiveCycle Data Services application.

The tutorial is a companion to the Engineering Support Center sample application. The tutorial steps you through the development of some areas of the sample application. You can explore the sample for more examples of Data Services feature implementations.

For more information about developing LiveCycle Data Services applications, see Using LiveCycle Data Services.

About the sample

The Engineering Support Center sample application is a Flash application with a LiveCycle Data Services server back end. The application is used by (the fictional company) GlobalCorp’s call center employees and customers.

The business workflow
Customers of GlobalCorp contact the corporate support center to log complaints about products that they have purchased. Support center agents perform the following tasks in response to the complaints:

• Retrieve the details of the complaint and the customer’s information (for example, their name and email address). This is typically done by telephone.
• Enter the information in an electronic form to create a case file.
• Email the file to the engineering team to formally begin the case resolution process.
• When the case is resolved, agents update the case file with the resolution details. They then contact the customer to provide the resolution and close the case.

Entering data into forms is time consuming and mistakes are common. During telephone conversations, customers tend to interject information that is not important to the case, which increases the length of the call. Also, it is difficult to extract information for efficiency metrics from the case files.

The requirements
GlobalCorp wants to improve the efficiency of this business process. They have the following requirements for the solution:

• Enable customers to use an online form to provide case details and contact information.
• When the customer submits the form, the case is automatically created and assigned to an agent.
• When the agent updates the case, the customer can automatically see the new information.
• Call center supervisors can monitor case activity to make sure they are closed in a reasonable time.
• All users need to log in to use the application.

The solution
The Engineering Support Center application uses Data Services, a back-end database, and Flash client applications to implement the solution. The customer, the call center agent, and the supervisor all use a different software client. Each client provides slightly different features because each type of user performs different tasks.
To download the sample, go to the LiveCycle Data Services Developer Center.

**Tutorial modules**

The Creating a Data Services Application tutorial is divided into several modules. The modules are designed so that you can choose to complete all or only some of them:

- Complete all modules in the order that they are presented to learn the end-to-end application development process.
- Complete a subset of the modules to focus on specific application development tasks.
- The Getting Started module is mandatory. Complete this module before continuing with any other module.

**Getting Started**

Install required software and set up your development environment. After you configure a web application and create a Flash Builder project, you are ready to continue with any other module. (See “Getting Started” on page 3.)

**Creating Entities and Database Tables**

Create and configure a data model to persist information required of GlobalCorp’s call center business workflow. Learn how to create entities, value expressions, and relationships. Deploy the model to create database tables. (See “Create the Data Model” on page 11.)

**Developing the Data-Centric Client Application**

Complete the development of the supervisor dashboard. Learn how to generate client ActionScript classes from the data model and bind data components to service operations. Customize the template for generating model-driven forms. (See “Developing the Data-Centric Client Application” on page 22.)

**Server-side Programming and Configuration**

Customize assembler classes and deploy them to the server. Integrate Data Services with Spring security to implement authentication and authorization. (See “Server-Side Configuration and Programming” on page 36.)

**Packaging and Deploying**

Compile, package, and deploy the application to the Tomcat application server. (See “Packaging and Deploying the Application” on page 50.)
Chapter 2: Getting Started

In this module of the Creating a LiveCycle Data Services ES2 Application, you learn how to prepare your environment for developing applications. While preparing your environment, you learn how to accomplish the following tasks:

- Create a web application to use as a starting point for developing a new application.
- Configure the application to connect to a back-end database.
- Configure the application to enable the application modeler to connect to the database.
- Use Flash Builder to create a project, and to see a view of the database.

Note: You must complete this module before starting any other module of this tutorial.

Download tutorial files

Several files are provided for you to work with in each tutorial module. Download the tutorial files and save them to your hard drive.

Install required software

To perform the exercises of this tutorial, you must have the following software installed:

- Adobe Flash Builder 4.0
- Adobe LiveCycle Data Services 3.1, including the application modeling plugin for Flash Builder.

Install Flash Builder

The procedures in this tutorial assume you are using the standalone version of Flash Builder 4.0.

To obtain Flash Builder 4.0, visit the Adobe web site. You can try Flash Builder for a period of time without purchasing a license.

Install LiveCycle Data Services

The procedures in this tutorial assume you are using LiveCycle Data Services ES2 3.1 with integrated Apache Tomcat application server.

To obtain LiveCycle Data Services 3.1, visit the Adobe web site. During the installation, select the LiveCycle Data Services With Tomcat option.

When you install, the application modeler plugin for Flash Builder is copied to your hard drive. Install the plugin file to your Flash Builder 4.0 installation.

For installation instructions, see Installing Adobe LiveCycle Data Services ES2 Version 3.1.

Note: This tutorial assumes you install the Data Services server on the same computer where Flash Builder is installed. If you install the server on a different computer, note the server’s IP address.
Create a database

When you installed LiveCycle Data Services, an instance of the HSQLDB database server was also installed for use with Data Services samples and for development purposes. This tutorial also uses the database server:

- The model creates database tables
- Data services persist application data

Edit the database properties file and start the database server to create a new database. The database uses the default values for the user name (sa) and password (no password). The database provides access on port 9002.

Create the tutorial database:
1. Use a text editor to open the server.properties file in the \[lcds_root\]/sampledb directory.
2. Above the line `server.port=9002`, insert the following text:
   ```
   server.database.x=file:tutorialdb/tutorialdb
   server.dbname.x=tutorialdb
   ```
3. In the text that you just inserted, change the two instances of x to the next integer in the sequence of existing database definitions. For example, if the highest integer is 4 (such as `server.dbname.4=callcenterarchived`), then change x to 5:
   ```
   server.database.5=file:tutorialdb/tutorialdb
   server.dbname.5=tutorialdb
   ```
4. Save and close the server.properties file.
5. Start the database server using one of the following scripts, depending on your operating system:
   - (Windows) \[lcds_root\]/sampledb/startdb.bat
   - (Unix-based) \[lcds_root\]/sampledb/startdb.sh

On Windows, you can click Start > All Programs > Adobe > LiveCycle Data Services ES2 Version 3.1 > Start Samples Database

The sampledb directory now contains a directory named tutorialdb where the application data will be stored. Now, modify the stopdb script to include the tutorialdb database.

Add tutorialdb to the stopdb script:
1. Open the stopdb script in a text editor:
   - (Windows) \[lcds_root\]/sampledb/stopdb.bat
   - (Unix-based) \[lcds_root\]/sampledb/stopdb.sh
2. Add the following line of code:
   ```java
   ```
3. Save and close the stopdb.bat or stopdb.sh file.

Create a web application

Create a web application on the Tomcat application server to use for application development throughout the tutorial. LiveCycle Data Services provides a simple web application that you use as the starting point.
After you copy the application, change the port numbers for NIO-based end points to avoid port conflicts.

**Copy the starter web application:**
1. Create a directory named tutorialApp under the following directory: \[lcds_root\]/tomcat/webapps.
2. Copy the content of the following directory to the tutorialApp directory: \[lcds_root\]/tomcat/webapps/lcds.
   
   *Note:* The context root of your application is /tutorialApp.
3. Open the \[lcds_root\]/tomcat/webapps/tutorialApp/WEB-INF/flex/services-config.xml file in a text editor.
4. In the channels element, locate the <!-- NIO based endpoints --> comment (near line 94).
5. Change the port number used for the endpoint that is defined for each NIO-based endpoint. The following table identifies the channel definition and port numbers to use.

<table>
<thead>
<tr>
<th>Channel definition id</th>
<th>Original port number</th>
<th>New port number</th>
</tr>
</thead>
<tbody>
<tr>
<td>my-rtmp</td>
<td>2038</td>
<td>2048</td>
</tr>
<tr>
<td>my-nio-amf</td>
<td>2880</td>
<td>2890</td>
</tr>
<tr>
<td>my-nio-amf-poll</td>
<td>2880</td>
<td>2890</td>
</tr>
<tr>
<td>my-nio-http</td>
<td>2880</td>
<td>2890</td>
</tr>
<tr>
<td>secure-nio-amf</td>
<td>2443</td>
<td>2453</td>
</tr>
<tr>
<td>secure-nio-http</td>
<td>2443</td>
<td>2453</td>
</tr>
</tbody>
</table>

For example, the following XML defines the my-rtmp channel after the port is changed:

```
<channel-definition id="my-rtmp" class="mx.messaging.channels.RTMPChannel">
  <endpoint url="rtmp://{server.name}:2048" class="flex.messaging.endpoints.RTMPEndpoint"/>
  <properties>
    <idle-timeout-minutes>20</idle-timeout-minutes>
    <!-- for deployment on WebSphere, must be mapped to a WorkManager available in the web application's jndi context. -->
    <websphere-workmanager-jndi-name java:comp/env/wm/MessagingWorkManager />
  </properties>
</channel-definition>
```

6. Save and close the services-config.xml file.

**Enable the RDS servlet**

The web application that you copied includes a preconfigured remote development services (RDS) servlet. The application modeler accesses the database through the RDS servlet.

By default, the RDS servlet is disabled. Enable the RDS servlet so that he modeler can access the tutorialdb database. In addition, disable security to ease the development process.

**Configure the RDS servlet**
1. Use a text editor to open the web.xml file from the \[lcds_root\]/tomcat/webapps/tutorialApp/WEB-INF directory.
2. Locate the line with the text <!-- begin rds and append --> to the end of the line.
3. Locate the line with the text `end rds -->` and insert `<!--` at the start of the line:

```xml
<!-- begin rds -->
<servlet>
    <servlet-name>RDSDispatchServlet</servlet-name>
    <display-name>RDSDispatchServlet</display-name>
    <servlet-class>flex.rds.server.servlet.FrontEndServlet</servlet-class>
    <init-param>
        <param-name>useAppserverSecurity</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>10</load-on-startup>
</servlet>
<servlet-mapping id="RDS_DISPATCH_MAPPING">
    <servlet-name>RDSDispatchServlet</servlet-name>
    <url-pattern>/CFIDE/main/ide.cfm</url-pattern>
</servlet-mapping>
<!-- end rds -->
```

4. In the `init-param` element inside the `servlet` element, change the value for `useAppserverSecurity` to `false`:

```xml
<param-name>useAppserverSecurity</param-name>
<param-value>false</param-value>
```

5. Save and close the `web.xml` file.

---

## Create the Context

Create the context descriptor on the Tomcat application server to represent the tutorialApp application. The Context element in your context descriptor defines the data source for the tutorialdb database. It also indicates the transaction factory to use. (The Tomcat instance that is installed with Data Services includes the Java Open Transaction Manager (JOTM) libraries.)

The following text is used for your Context Descriptor:

```xml
<Context privileged="true"
    reloadable="true">
    <!-- JOTM -->
    <Transaction factory="org.objectweb.jotm.UserTransactionFactory"
        jotm.timeout="60"/>
    <Resource name="jdbc/tutorialdb" type="javax.sql.DataSource"
        driverClassName="org.hsqldb.jdbcDriver"
        maxIdle="2" maxWait="5000"
        url="jdbc:hsqldb:hsq1://localhost:9002/tutorialdb"
        username="sa" password="" maxActive="4" />
</Context>
```

Notice that the value for the `url` attribute of the `Resource` element reflects the properties of the tutorialdb database.

On Tomcat, context descriptor files are located in the `[tomcat_root]/conf/Catalina/localhost` directory. Context descriptor file names must use the format `[application_name].xml`. For your application, the file name is `tutorialApp.xml`.

### Create the data source:

1. Open a text editor and create a new document.
2 Copy the Context element provided above to the document.
3 Save the document as tutorialApp.xml in the \[lcds_root]/tomcat/conf/Catalina/localhost directory and close the file.

**Start the server**

Start the Data Services server and open the tutorialApp home page to verify that the server is running.

Start the server:
1 To start Tomcat, use one of the following scripts, depending on your operating system:
   • (Windows) \[lcds_root]/tomcat/bin/startup.bat
   • (Unix-based) \[lcds_root]/tomcat/bin/startup.sh
   
   On Windows, you can click Start > All Programs > Adobe > LiveCycle Data Services ES2 Version 3.1 > Start LiveCycle Data Services Server

   The server is running when the message **INFO: Server startup in xxxxxxx ms** appears in the command window.
2 In a web browser, navigate to the following URL:
   http://localhost:8400/tutorialApp

   The default web page for the tutorialApp application opens:

   *Welcome to Adobe LiveCycle Data Services ES2 Version 3.1!*

   This is an empty LiveCycle Data Services ES application that serves as a template for creating your custom application.

   Visit the **product page** for documentation and further information.

**Create a project in Flash Builder**

Create a project in Flash Builder to begin the application development process. Configure the project so that it is associated with the tutorialApp application on the Tomcat server. Binary files that are compiled are automatically copied to the directories of the web application for testing.

Create a project:
1 Start Flash Builder.
2 Click File > New > Flex Project.
   
   **Note:** The Flex Project item is available only if the Flash perspective is open.
3 In the Project Name box, type Tutorial_GS.
4 For Application Server Type, select J2EE and then click Next.
5 Configure the following properties to associate the project with the tutorialApp application:
   • **Root Folder:** \[lcds_root]/tomcat/webapps/tutorialApp
   • **Root URL:** http://localhost:8400/tutorialApp

Last updated 6/29/2010
Getting Started

Context Root: /tutorialApp

6 Click Validate Configuration, and then click Finish.

Connect to the RDS server

Connect to the RDS server that you enabled in your tutorialApp web application.

Configure a connection to the RDS server:
1 In Flash Builder, click Window > Preferences.
2 In the tree, select Adobe > RDS Configuration.

You can also open the RDS Configuration dialog box from the RDS Data View view in the Data Model perspective.

3 Click New, and configure the following properties:
   - **Description**: tutorialApp RDS Server
   - **Host Name**: 127.0.0.1
   - **Port Number**: 8400
   - **Context Root**: /tutorialApp

*Note: If you installed the Data Services server on a different computer, type the IP address of the server in the Host Name box.*
4 Click Test Connection. A message should appear that indicates the connection was successful.

![Test Connection successful message](image)

5 Click OK to close the message, and then click OK to close the Preferences dialog box.

6 Click Window > Perspective > Data Model to open the Data Model view. This perspective includes the RDS Dataview view.

7 In the RDS Dataview view, expand tutorialApp RDS to see the view of the tutorialdb database. The database has no tables, except for the system tables that were created by default.

![RDS Dataview view](image)

**Next steps**

In this part of the tutorial you set up your environment to prepare for application development using Data Services and Flash Builder:

- You created the tutorialApp web application on the application server, and associated it with the new tutorialdb database.
- You created the Tutorial_GS project in Flash Builder for developing J2EE application clients.
- You connected Flash Builder to the RDS server of the web application, and observed a view of the tutorialdb database.

Your development environment is now ready for you to begin developing a Data Services application:

- Continue the tutorial with the “Create the Data Model” on page 11 module.
• Jump ahead to another module:
  • “Developing the Data-Centric Client Application” on page 22
  • “Server-Side Configuration and Programming” on page 36
  • “Packaging and Deploying the Application” on page 50
• Go to “Tutorial Overview” on page 1 to see module descriptions.
Chapter 3: Create the Data Model

In this module of the Creating a LiveCycle Data Services ES2 Application tutorial, you learn about the following LiveCycle Data Services features:

- Relating model entities to application data
- Adding model entities and properties
- Creating variants
- Creating relationships
- Deploying models and creating database tables

In this module, you create a data model that is similar to the one included with the sample application. You will deploy the model to the Data Services server to create the database tables that are used to persist application runtime data.

**Note:** When you are developing your own Data Services applications, you are more likely to introspect an exiting data source and automatically create the model.

To read more about data models, see Model-driven development in Application Modeling Technology Reference.

Setting up

Before you perform the exercises in this module, you need to complete the “Getting Started” on page 3 module. You will be using the Flash Builder project, Data Services application, and database that you created.

You also need to have downloaded the tutorial files.

Review the application data

As described in “About the sample” on page 1, the application workflow involves the following roles:

- The customer
- The call center agent
- The call center supervisor

All users are assigned a role so that the application can determine which features they are authorized to use. The role also determines the amount of user information that is stored. For example, the customer’s name and contact information is stored. Only the agent’s name is required.

Case information also needs to be persisted:

- Case description
- The customer who logged the case.
- The agent or supervisor who is assigned the case
- Information that is added during the life of the case, such as notes and status.

**Note:** Both customers and agents can add notes to cases.
Entities and their relationships
To model the application data, you identify the entities (the 'things') that are involved:

- Users
- Roles (Customer, Agent, Supervisor)
- Cases
- Notes

Each entity has different properties. For example, users have login names and passwords, and cases have descriptions and creation dates.

Entities also have relationships between each other. For example, each User is assigned a role, and Customers can create cases and notes. The following diagram illustrates some of the entity attributes and relationships. For simplicity, not all attributes and relationships are represented.

The design of the model can be based on this type of diagram.

Create the model

In this exercise you will add a model to the Tutorial_GS project in Flash Builder.

Add the model:
1. Open Flash Builder.
2. In the Package Explorer view, click the Filters button.
3. Deselect Model and click OK.
4. Right-click the Tutorial_GS project and click Data Model > Open Model.
5. If the Open Associated Perspective dialog box appears, select Remember My Decision, and then click Yes to open the Data Model perspective.

The Tutorial_GS.fml file is added to the .model folder in the project, and opens in the editor view.
Create the Customer entity
Now that you have a model for your project, you can add the entities and entity properties that represent the application data.

Entities are analogous to database tables. Properties are analogous to table columns.

Add the entity:
1. From the tool bar, drag the Entity icon onto the editor.
2. Type Customer to replace the default name of Entity.

The entity includes a default property named ID. This property uniquely identifies instances of the Customer entity.

3. Click the ID property and in the Properties view, change the following property values:
   - Name: custId
   - Type: String
   - Auto-generated: Deselected

4. Save the model.

Add Customer properties
In this exercise you will add properties of different data types to the Customer entity.

Add properties:
1. On the tool bar, make sure the Properties button has the Text Property option selected. The button should have a ‘T’ icon.
2. Click the Properties button to select it, and pause the cursor over the Customer entity, below the custId property. When a black line appears below custId, click the mouse to add the property.
3  Type lastName to replace the default name of the property.

4  Repeat steps 2 to 3 to add text properties with the following names:
   - firstName
   - fullName
   - phoneNumber
   - email
   - city
   - state
   - country
   - zipCode
   - address
   - pic

   To change the location of a property, drag it to a new location in the entity.

5  Select the zipCode property and in the Properties view, change Type to integer.
   The property icon changes to indicate the type.

6  Select the pic property and in the Properties view, change Type to blob.

7  Save the model.

**Working with the XML code**

In FML files, models are represented using XML code. The visual representation of the model that you have created in Design mode is automatically translated to XML.

In this exercise you will explore the model code. So far, your model code contains the following model definition elements:
   - model (the root element)
   - entity
   - property
CREATING A LIVECYCLE DATA SERVICES ES2 APPLICATION

Create the Data Model

For a complete description of the XML elements, see Model XML elements in Application Modelling Technology Reference.

Switch to Source mode:
❖ At the top of the model editor, click the Code View button.

You should see XML code similar to the following code:

```xml
<model xmlns="http://ns.adobe.com/Fiber/1.0">
  <entity name="Customer" persistent="true">
    <annotation name="VisualModeler" width="115" height="270" x="63" y="91"/>
    <annotation name="ServerProperties" ServerType="LCDS"/>
    <id name="custId" type= "string" generated="true"/>
    <property name = "lastName" type="string"/>
    <property name = "firstName" type="string"/>
    <property name = "fullName" type="string "/>
    <property name="phoneNumber" type="string"/>
    <property name="email" type="string"/>
    <property name="city" type="string"/>
    <property name="state" type="string"/>
    <property name="country" type="string"/>
    <property name="zipCode" type="integer"/>
    <property name="address" type = "string"/>
    <property name="pic" type="blob"/>
  </entity>
</model>
```

Notice the annotation elements below the entity element. Annotations define name-value pairs that provide additional information about model definition elements. For example, the annotation with the name of VisualModeler provides information to Flash Builder for locating the Customer entity in the editor when using Design view.

The exercises in this tutorial primarily involve Design mode. To become familiar with the XML modeling syntax, occasionally switch to source mode as you perform the exercises.

Copy other entities to your model

In the previous exercises you learned how to create an entity and add properties. To save time and to make the model more interesting, in this exercise you will copy the remaining entities to your model:

• Agent
• CustCase
• Note
• User

You need the tutorial files that you downloaded from [****url tbd]

Copy entities to your model:
1 In Flash Builder, click File > Open File.
2 Browse for the model/ed_model_tocopy.fml file that you downloaded and open it.
3 Select all of the entity properties in the model code.

Last updated 6/29/2010
4 Right-click the selection and click Copy.

5 In Tutorial_GS.fml, add a new line after the closing </entity> element, place the cursor on the line, and then right-click and select Paste.

6 Save the model, then click the Design View button.

Two entity relationships are already configured:

- The Agent entity includes a property named agentAuthData, which is of type User. This relationship enables each Agent instance to have an associated user account.
- CustCase entities can include zero or more Note entities. Customer cases have notes added to them.

Notice that there is no entity that represents call center supervisors. You will configure the model to represent supervisors in a later exercise.

**Add relationships to the Customer entity**

Several relationships need to be defined between the Customer entity and the entities that you added in the previous exercise. The Customer relationships are similar to the relationships of the Agent entity:

- Customers require a user account. Define a relationship with the User entity so that each Customer instance is associated with one unique User instance.
- Customers create cases. Define a relationship with the Case entity so that each Customer instance can be associated with several Case instances.

Relationships are analogous to foreign keys in the database.

For more information about relationships, see entity associations in Application Modelling Technology Reference.

**Add a relationship to the User entity:**

1 Add a property to the Customer entity and name it custAuthData.

2 In the Properties view, set the Type to User.

   In the model editor, a relationship is drawn between the Customer and User entities.
Click the relationship line between Customer and User and, in the Properties view, and set the following property values:

- **Cardinality:** One To One
- **Required (Relationship Options):** Selected

A description of the relationship is provided:

A property in multiple instances of “Customer” can never refer to the same instance of “User” (In other words uses of “User” must be unique)

In the Properties view, select Required.

---

### Add a relationship to the CustCase entity

1. Add a property to the Customer entity and name it cases.
2. In the Properties view, set Type to CustCase and select This Is A Collection.
3. Add a property to the CustCase entity and name it customer.
4. Set the Type to Customer.

   *Move properties up or down in an entity to avoid crossing relationship lines.*

5. Click the cases property, and then click the Relationships tab in the Property view.
6. In the Mapped By box, select customer.
7. Click the line that represents the relationship you just created. In the properties view, a description of the relationship is provided:

   A property in multiple instances of “CustCase” can refer to the same instance of “Customer”

---

### Add support for the Supervisor role

As previously discussed, the model includes Customer and Agent entities, but no Supervisor entity. The information about supervisors that needs to be persisted is the same as that of agents. Therefore, the same database table can be used to store agent and supervisor information.

However, supervisors and agents perform different tasks, so both roles must be defined to distinguish the two types of users. Therefore, the Agent entity includes a boolean property named isSupervisor. This property is used to distinguish agents and supervisors.
In the call center’s case workflow, special cases are assigned to supervisors as needed. The application needs to distinguish these escalated cases. In this exercise, you will configure the Agent entity so that the value of isSupervisor controls the relationship it has with the CustCase entity:

- A value of false causes a relationship with the agent property of the CustCase entity.
- A value of true causes a relationship with the supervisor property of the CustCase entity.

The value of isSupervisor determines the name of the CustCase column that contains the foreign key for the Agent table. It also determines which column in the Agent table holds the key (either cases or escalated_cases). You use a variant to implement this behavior.

**Variants are similar to a series of if...else statements, or switch statements, in scripting or programming languages. (See variant in Application Modelling Technology Reference.)**

**Add a variant to the Agent entity:**

1. In the tool bar, click the Variant button.
2. Click inside the Agent entity to add the variant.
3. In the Properties view, click the Expression Editor button.
4. Delete the default text in the expression editor, double-click Agent.isSupervisor, and then click OK.
5. In the tool bar, click the Case button, and then click the isSupervisor variant.
6. Set the Value Expression for the case to false.
7. Add another case to the variant and set the Value Expression to true.
8. Create a relationship with CustCase.agent when isSupervisor is false:
   a. Select the agent property of the CustCase entity and set the type to Agent.
b In the Agent entity, add a property below the false case and name it cases. Set the Type to be a collection of CustCase entities.

c Select the cases property and in the Properties view click the Relationships tab. For Mapped By, select agent.

*If the Mapped By property is not available, make sure the This Entity Is Persistent option of the Agent entity is selected.*

9 Create a relationship with CustCase.supervisor when isSupervisor is true:

a Select the supervisor property of the CustCase entity and set the type to Agent.

b Add a property below the true case, name it escalated_cases, and set the Type to be a collection of CustCase entities.

c Select the escalated_cases property and in the Properties view click the Relationships tab. For Mapped By, select supervisor.

10 Save the model.
Create the database tables

In this exercise you will deploy the model to the Data Services server to create the database tables. Before deploying, you will specify some table and columns names.

*If you are interested in generating a model from existing database tables, see Creating a client for an existing service destination in Using Adobe LiveCycle Data Services.*

Specify table and column names:
1. Select the Note entity.
2. In the Properties view, in the Database Information area of the General tab, enter `notes` for Table Name.
3. Select the lastName property of the Customer entity, and set Column Name to `last_name`.
4. Specify table and column names according to the following table:

<table>
<thead>
<tr>
<th>Item</th>
<th>Table or column name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer.firstName property</td>
<td>first_name</td>
</tr>
<tr>
<td>CustCase entity</td>
<td>cases</td>
</tr>
<tr>
<td>Agent.firstName property</td>
<td>first_name</td>
</tr>
<tr>
<td>Agent.lastName property</td>
<td>last_name</td>
</tr>
</tbody>
</table>

Deploy the model:
1. From the RDS Dataview view, drag `java:/comp/env/jdbctutorialdb` to a blank area of the model editor.
   The following values are added to the Properties view:
   - **JDBC Datasource**: `java://comp/env/jdbc/tutorialdb`
   - **Hibernate SQL Dialect**: `org.hibernate.dialect.HSQLDialect`
2. In the model editor, click the Deploy Model To LCDS Server button.
3. For Server, select `tutorialApp RDS Server`.
4. Select Update (Or Create If They Do Not Exist).
5. Click Finish, and then click OK when the model is deployed.
6. In the RDS Dataview, expand `tutorialApp RDS Server > java:/comp/env/jdbctutorialdb/Tables`. The tables are created.
Next steps

In this part of the tutorial you created the data model for the Data Services application:

- You created the Customer entity.
- You added entities from a different model.
- You added a variant to support the Supervisor role.
- You deployed the model to create the database tables.

Your project is now ready for developing client applications:

- Continue the tutorial with the “Developing the Data-Centric Client Application” on page 22 module.
- Jump ahead to another module:
  - “Developing the Data-Centric Client Application” on page 22
  - “Packaging and Deploying the Application” on page 50
- Go to “Tutorial Overview” on page 1 to see module descriptions.
Chapter 4: Developing the Data-Centric Client Application

In this module of the tutorial you generate service wrappers for developing Flash Builder client applications that interact with the tutorialdb database. You are provided with an incomplete MXML application that is similar to the Supervisor Dashboard of the Engineering Support Group sample application.

The exercises in this module involve binding the data components of the application to the model-generated data management services.

Setting up

Before you perform the exercises in this module, you need to complete the “Getting Started” on page 3 module. You will be using the Flash Builder project, Data Services application, and database that you create in that module.

You also need to have downloaded the tutorial files.

Start Tomcat and the database server

1. Start the database server using one of the following scripts, depending on your operating system:
   • (Windows) \lcts_root\sampledb\startdb.bat
   • (Unix-based) \lcts_root\sampledb\startdb.sh
   
   On Windows, you can click Start > All Programs > Adobe > LiveCycle Data Services ES2 Version 3.1 > Start Samples Database

2. Start Tomcat using one of the following scripts, depending on your operating system:
   • (Windows) \lcts_root\tomcat\bin\startup.bat
   • (Unix-based) \lcts_root\tomcat\bin\startup.sh
   
   On Windows, you can click Start > All Programs > Adobe > LiveCycle Data Services ES2 Version 3.1 > Start LiveCycle Data Services Server

The server is running when the message INFO: Server startup in xxxxxxx ms appears in the command window.
Add the model to your Flash Builder project and deploy:
Perform this procedure if you did not complete the previous module, or you want to replace the model that you created.

1. Open Flash Builder.
2. In the Package Explorer view, click the Filters button.
3. Deselect Model and click OK.
4. Right-click the Tutorial_GS project and click Data Model > Open Model.
5. If the Open Associated Perspective dialog box appears, select Remember My Decision, and then click Yes to open the Data Model perspective.
7. From the tutorial files that you downloaded, copy the client/Tutorial_GS.fml file to your [workspace]/Tutorial_GS/.model directory. Replace the existing file of the same name.
8. In the model editor, click the Deploy Model To LCDS Server button.
10. Select Update (Or Create If They Do Not Exist).

   ![Deploy Data Model to LCDS](image)

11. Click Finish, and then click OK when the model is deployed.

Create packages and add source files:
You must complete this procedure. You will work with the source files in the exercises of this module.

1. In Flash Builder, right-click the Tutorial_GS/src folder and click New > Other.
2. In the dialog box, click Flash Builder > Package and click Next.
3. In the Name box, type com.adobe.lcds.demo.dashboards and click Finish.
4. Repeat steps 1 to 3 to create the following packages:
   - com.adobe.lcds.demo.banners
   - com.adobe.lcds.demo.forms
5. From the tutorial files that you downloaded, copy files to your Flash Builder workspace according to the following table:
6 In Flash Builder, select the Package Explorer view and click File > Refresh.

7 Right-click the Tutorial_GS project and click Properties.

8 In the properties tree, select Flex Applications, and then click the Add button.

9 Select the com/adobe/lcds/demo/dashboards directory and click OK.

   The CustomerDashboard.mxml and SupervisorDashboard.mxml files are added as runnable applications.

10 In the Properties for Tutorial_GS dialog box, click OK.

   Note: The MXML files have compiler errors until you complete the exercises in this module.

Copy resource files:
You must complete this procedure. The image files are required in the Flex Builder project and the tutorialApp web application.

1 From the tutorial files that you downloaded, copy the client/styles directory to the [workspace]/Tutorial_GS directory.

2 Copy the styles directory to the [lcds_root]/tomcat/webapps/tutorialApp/Tutorial_GS_debug directory.

Replace the database script:
You must complete this procedure. This procedure creates tables and adds example data to the tutorialdb database.

1 If the database server is running, stop the database server using one of the following scripts, depending on your operating system:
   • (Windows) [lcds_root]/sampledb/stodb.bat
   • (Unix-based) [lcds_root]/sampledb/stodb.sh

2 Copy the client_app/start/tutorialdb.script file and replace the file of the same name in the [lcds_root]/sampledb/tutorialdb directory.

3 Start the database server using one of the following scripts, depending on your operating system:
   • (Windows) [lcds_root]/sampledb/startdb.bat
   • (Unix-based) [lcds_root]/sampledb/startdb.sh

   On Windows, you can click Start > All Programs > Adobe > LiveCycle Data Services ES2 Version 3.1 > Start Samples Database

<table>
<thead>
<tr>
<th>Copy from the client directory</th>
<th>Target directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomerDashboard.mxml</td>
<td>[workspace]/Tutorial_GS/src/com/adobe/lcds/demo/dashboards</td>
</tr>
<tr>
<td>SupervisorDashboard.mxml</td>
<td>[workspace]/Tutorial_GS/src/com/adobe/lcds/demo/dashboards</td>
</tr>
<tr>
<td>Banner.mxml</td>
<td>[workspace]/Tutorial_GS/src/com/adobe/lcds/demo/banners</td>
</tr>
<tr>
<td>CustomerFormPopup.mxml</td>
<td>[workspace]/Tutorial_GS/src/com/adobe/lcds/demo/forms</td>
</tr>
<tr>
<td>styles (directory)</td>
<td>[workspace]/Tutorial_GS/</td>
</tr>
</tbody>
</table>
Generate ActionScript classes

The model in the Tutorial_GS package is associated with the tutorialdb database. In this exercise, you use the model to generate ActionScript classes for interacting with the data management services.

Two types of classes are created:

• **Service wrappers**: Represent data management services and provide access to service operations.

• **Value objects**: Represent entities and provide access to entity property values.

A class that extends each service wrapper and value object class is also created. These class extensions enable you to customize the client-side behavior when the service wrapper is used.

In the following exercise, you configure the model so that ActionScript classes are generated. You also specify the package names that the classes belong to.

*Note: Ensure the LiveCycle Data Services server is running.*

For more information about working with model-generated ActionScript, see Customizing client-side functionality in *Using Adobe LiveCycle Data Services*.

**Generate services:**

1. In the Flash Builder, open Tutorial_GS.fml. Switch to the Data Model perspective if it didn’t open automatically.

2. In the Properties view, click the Code Gen tab.

3. In the Service Package box below ActionScript Generation, type `services.callcentercustomer` to name the package that contains service wrapper classes.

4. Select Generate Fully Qualified Package Name References, and select Make LIKE Clauses Case Sensitive In Filters. (HSQLDB treats values as case-sensitive.)

5. For each entity, specify the name of the package for value objects:
   a. Select the entity.
   b. In the Code Gen tab of the Properties view, provide the following values for the ActionScript Generation properties:
      • **Package**: `valueObjects`
      • **Service Package**: `services.callcentercustomer`

6. Click File > Save.

   If you previously opened the Data/Services view, the client classes are generated and appear in the Project Explorer view. If the classes are not generated, continue to the next step.

7. On the model editor, click the Generate Code button.
Switch to the Flash perspective, and go to the Data/Services view. Explore the data types and operations that are available for each entity service. Compare them with the classes and class members in the Project Explorer view.

The names of service wrapper and value object classes begin with _Super_. For example, _Super_CustCaseService is the service wrapper, and CustCaseService extends the wrapper.

### Populate the case list

In this exercise you use a CallResponder object to interact with the CustCaseService service. The results are displayed in a DataGrid instance.

CallResponder objects enable you to make asynchronous calls to retrieve data without blocking your program’s execution. Use the following pattern to bind data components to the service data that is stored in the CallResponder object:

1. Create an instance of the service wrapper object and a CallResponder object.
2. Add the data component that will display the data.
3. Use the CallResponder object as the dataProvider of the component.
4. Create a function that is called when the component’s creationComplete event occurs. This function sets the CallResponder’s token property to the results of the service operation.

At run time, the data service is called when the data component is created. When the results are stored in the CallResponder object, the data component is populated with the data.
In the following exercise, you bind the cases_list DataGrid to the getAllCasesResult CallResponder. Additionally, you create functions that respond to the CallResponder’s Result and Fault events.

**Bind the DataGrid control to CustCase.GetAll**

1. Open SupervisorDashboard.mxml from the com.adobe.lcds.demo.dashboards package. If you are in Design mode, switch to Source mode.

2. In the fx:Declarations section, add the following code to create a CallResponder object:
   ```xml
   <s:CallResponder id="getAllCasesResult"/>
   ```
   An instance of the CustCaseService, which you will use to retrieve cases from the database, is already declared with id custCaseService.

3. Locate the DataGrid that has the id of cases_list (near line 502).

4. Add the following attributes to set the value of the dataProvider and creationComplete attributes:
   ```xml
dataProvider = "{getAllCasesResult.lastResult}"
creationComplete = "cases_list_creationCompleteHandler(event)"
```

5. Add the following code to the fx:script section. This code creates the cases_list_creationCompleteHandler function and registers event listeners on the CallResult object.
   ```actionscript
   //creation complete handler for the cases_list DataGrid
   private function cases_list_creationCompleteHandler(event:FlexEvent):void{
       getAllCasesResult.token = custCaseService.getAll();
       getAllCasesResult.addEventListener(ResultEvent.RESULT, _gotAllCasesResultHandler);
       getAllCasesResult.addEventListener(FaultEvent.FAULT, _gotAllCasesFaultHandler);
   }
   ```

   ```actionscript
   private function _gotAllCasesResultHandler(evt:ResultEvent):void {
       if(evt != null && evt.result != null && evt.result.length >=0)
           cases_list.selectedIndex = 0;
   }
   ```

   ```actionscript
   private function _gotAllCasesFaultHandler(evt:FaultEvent):void {
       Alert.show(evt.fault.faultString, resourceManager.getString("CallCenter","label.error"));
   }
   ```

6. Save the file and run the application.

The cases_list DataGrid contains the two cases that are already stored in the database.

You can also drag and drop operations to components to establish data binding. See Binding service operations to controls in Using Adobe LiveCycle Data Services.
Filter the case list

SupervisorDashboard.mxml includes several ComboBox instances that will be used to filter the cases in the Cases_list DataGrid. For example, when you select an agent from a ComboBox, only the cases that are assigned to that agent appear in the DataGrid.

In this exercise, you will add a filter to the CustCase entity and specify a database query. When the model is saved, a service operation is automatically created that performs the query and returns the results.

After you create the filter, you will implement the following pattern to use the filter operation and update the DataGrid contents:

1. Create a CallResponder object.
2. Create a function that calls the filter operation (_getCasesOnFilterModification). The results are stored in the CallResponder’s token property. The function also registers an event listener on the CallResponder’s Result event.
3. Create the function that is used as the event listener. This function sets the data provider of the DataGrid to be the CallResponder object.
4. Set the creationComplete property of the ComboBoxes to call the _getCasesOnFilterModification function.

At run time, when an item is selected from a ComboBox, the filter operation is called. When the results are returned, the DataGrid is populated with the data.

Create the filter:

1. Open Tutorial_GS.fml and switch to Design mode.
2. Select the CustCase entity.
3. In the Properties view, click the Filters tab, and click the Add Item button to add a filter.
4. Type getCasesByFilter for the filter name.
5. Select the Query option, and add the following command to the Query box:
   
   \[ \text{jpql:Select a From CustCase a Where a.agent.agentId LIKE :agentID AND a.status LIKE :statusName AND a.customer.custId LIKE :customerID AND a.creationDate >= :fromDate AND a.creationDate <= :toDate} \]

   For information about the query, see “Understanding the filter query” on page 29.
6. To define the data type of the query parameters, enter the following text in the Query Arguments box:
   
   \[ \text{agentID:string, statusName:string, customerID:string, fromDate:date, toDate:date} \]

7. Save the model and then open the Flash perspective. In the Data/Services view, the CustCaseService service now provides the getCasesByFilter operation.
Use the getCasesByFilter operation:
1 In the fx:Declarations section of SupervisorDashboard.mxml, add the following code to create a CallResponder object:
   
   `<s:CallResponder id="getByFilterResponder"/>`

2 In the fx:script section, locate the `_getCasesOnFilterModification` function

   This function already includes code that retrieves the values that are selected in the ComboBox components. The variables that store the values are used as the parameters of the getCasesUsingFilter operation.

3 After the final `else` statement in the function, add the following code:

   ```
   getByFilterResponder.token = custCaseService.getCasesByFilter
      (agentQueryString, statusQueryString, customerQueryString, fromDate, toDate);
   getByFilterResponder.addEventListener(ResultEvent.RESULT,_gotCasesByFilter);
   ```

   The token of the CallResponder is set to the filter operation results. The `_gotCasesByFilter` event listener is registered with the CallResponder.

4 Create the `_gotCasesByFilter` event listener. After the `_getCasesOnFilterModification` function, add the following code:

   ```
   private function _gotCasesByFilter(evt:ResultEvent):void
   {
      cases_list.dataProvider  = getByFilterResponder.lastResult;
      if(evt != null && evt.result != null && evt.result.length >=0)
      {
         cases_list.selectedIndex = 0;
      }
   }
   ```

   The data provider of the cases_list DataGrid is now the results of the filter operation.

5 Locate the mx:combobox element with the id attribute of agentSelected, and add the following attribute:

   ```
   close="_getCasesOnFilterModification()"
   ```

6 Repeat step 5 for the mx:combobox elements with id attributes of statusSelected and customerSelected.

7 Save and run the application.

8 From the Customer drop-down, select jjacobs to show the case that jjacobs created.

---

**Understanding the filter query**

Query filters use Java Persistence Query Language (JPQL) statements. JPQL syntax is similar to SQL. However, JPQL is object-oriented, whereas SQL is table-oriented.

You created the following filter query in “Filter the case list” on page 28:

```
jpql:Select a From CustCase a Where a.agent.agentId LIKE :agentID AND a.status LIKE :statusName AND a.customer.custId LIKE :customerID AND a.creationDate >= :fromDate AND a.creationDate <= :toDate
```
The query selects all CustCase instances that contain a specific agent, status, customer, and creation date. The variable a and its properties are referred to throughout the statement:

- The From clause declares the a variable which represents any CustCase value.
- The Where clause compares CustCase property values to parameter values. For example, in a.agent.agentId LIKE :agentID, the ID of the agent in CustCase instances is compared to the parameter agentID.

Parameter values are provided in the call to the service operation.

For information about JPQL syntax, see openjpa.apache.org/builds/1.2.0/apache-openjpa-1.2.0/docs/manual/jpa_langref.html.

**Extend CustCaseService to extract case metrics**

In this exercise you will modify the CustCaseService class to customize the behavior of the service wrapper. A function is added that counts the number of open, closed, and deferred cases in an ArrayCollection of cases. The results are represented in the pie chart on the Supervisor Dashboard application.

The CustCaseService class extends the _Super_CustCaseService class, which is the client-side wrapper of the CustCaseService data service. Modifying the CustCaseService class affects client-side behavior, not the behavior of the service.

Both classes are generated by the model. However, CustCaseService.as is generated once and the service wrapper is generated every time the model is saved. Changes that you make to CustCaseService.as are preserved when the service wrapper class is generated again.

**Extend CustCaseService:**

1. Open CustCaseService.as from the services.callcentercustomer package.
2. Add the following import statements above the class statement:

   ```
   import mx.collections.ArrayCollection;
   import valueObjects.CustCase;
   ```
3. Copy the following function to the CustCaseService class:

   ```
   public function getEachStatusCount(cases :ArrayCollection) : ArrayCollection {
     var expenses:ArrayCollection = new ArrayCollection([{
       Status:"Open", Count:0},
       {Status:"Closed", Count:0},
       {Status:"Deferred", Count:0}]);
     for each (var obj:CustCase in cases) {
       if (obj.status == "Open") {
         expenses.getItemAt(0).Count++;
       } else if (obj.status == "Closed") {
         expenses.getItemAt(1).Count++;
       } else if (obj.status == "Deferred") {
         expenses.getItemAt(2).Count++;
       }
     }
     return expenses;
   }
   ```
4  Save and close CustCaseService.as.

Modify the pie chart:
1  In SupervisorDashboard.mxml, locate the PieChart component (id="myChart").
2  Add the following dataProvider attribute:
   ```
   dataProvider="{custCaseService.getEachStatusCount(cases_list.dataProvider as ArrayCollection)}"
   ```
3  Below the PieChart is an instance of the Legend component. Add the following dataProvider attribute to the Legend instance:
   ```
   dataProvider="{myChart}"
   ```
4  Save and run SupervisorDashboard.

Generate a model-driven form

To quickly enable clients applications to perform CRUD operations on a data source, you can generate model-driven forms. Model-driven forms are Flex forms that are derived from service-defined data types:

• For each property of the data type, the form includes a component for specifying a value.
• Buttons are included for specifying which CRUD operation to perform.
• A FreeMarker template determines the ActionScript code that is generated. FreeMarker is a Java-based template engine. (See http://freemarker.org.)

💡 All code that is generated from a model is based on FreeMarker templates.

In the following exercises, you will customize the form template, and then generate a form to use with the Customer Dashboard application.

Customize the form template

In this exercise you learn how to extract and modify the template that defines the ActionScript code in model-driven forms.

By default, model-driven forms include only Halo components. You will replace the default template with one that uses both Halo and Spark components.

Extract and modify the form template:
1  Right-click Tutorial_GS.fml and select Data Model > Extract Templates For Active Generators.
   
   The form template is extracted and saved in the templates/as/form folder.
2 Open templates/as/form/ASModelDrivenForm.ftl. Notice there is only one namespace defined. In this case, Halo components are used by default.

3 Close ASModelDrivenForm.ftl.

4 From the tutorial files that you downloaded, copy client/ASModelDrivenForm.ftl to the templates/as/form folder to replace the default template.

5 Open templates/as/form/ASModelDrivenForm.ftl.

The s and mx namespaces defined to distinguish Spark and Halo Flex components. Component elements are now prefixed with either the s or mx namespace.

Create the model-driven form

In this exercise you will create a form from the Customer data type. You will also incorporate the form into the Customer Dashboard. From the customer’s perspective, the properties on the form represent their user profile. Customers will use the form to update their profile information.

Model-driven forms enable users to perform CRUD operations on all entity properties by default. Often, only a subset of the CRUD operations or properties are required, so the default form needs to be altered. In this exercise, you will remove the Add and Delete buttons from the form that you create. We do not want users to add or remove customer profiles.

After you create and modify the form, you will incorporate it into the Customer Dashboard. Customer Dashboard uses PopUpManager to display the form in a new window. CustomerFormPopup.mxml defines the window that hosts the form.

Create the form:

1 Select the Customer entity, and in the Properties view, click the Code Gen tab.

2 In the Model Driven Form Package box, enter com.adobe.lcds.demo.forms. This is the package where model-driven forms are saved.

3 Save Tutorial_GS.fml.

4 Open CustomerFormPopup.mxml.

5 In the Data/Services view, expand CustomerService > Data Types. Right-click Customer and select Generate Form.

6 In the Choose Form Type dialog box, select Model Driven Form and then click OK.

CustomerFormPopup.mxml now includes an instance of the new model-driven form named CustomerForm. The com.adobe.lcds.demo.forms package contains CustomerForm.mxml.

7 Move the instance of CustomerForm inside the VGroup:
8 Switch to Design mode to see the form.

Modify the CustomerForm.mxml:
1 Open CustomerForm.mxml and switch to Source mode.
2 Delete the Add button (id = "_btnAdd") and Delete button (id = _btnDelete).
   <s:Button id="_btnAdd" label="Add" click="_btnHandler(event)"
   enabled="(addEnabled)"/>
   <s:Button id="_btnDelete" label="Delete" click="_btnHandler(event)"
   enabled="(deleteEnabled)"/>
3 Delete references to the buttons that you just deleted:
   a In the updateEnablement function (line 461), delete the following lines of code:
      _btnAdd.enabled = addEnabled;
      _btnDelete.enabled= deleteEnabled;
   b In the _btnHandler function, delete the if statements that pertain to _btnAdd and _btnDelete:
   c Change the else if statement that pertains to _btnSave so that it is an if statement. The updateEnablement
      function should now have the following structure:
      if (event.target == _btnSave)
      {
         ...
      }
      else if (event.target == _btnReset)
      {
         ...
      }
      updateEnablement();
4 Add code to close the popup window when the _btnSave button is clicked:
   • In the fx:script section, add the following code to import PopUpManager:
CREATING A LIVECYCLE DATA SERVICES ES2 APPLICATION
Developing the Data-Centric Client Application

import mx.managers.PopUpManager;

- Add this code to create an instance of CustomerFormPopup:

  [Bindable]
  public var customerFormPopUp:CustomerFormPopup;

- Add the following code in the first if statement of the _btnHandler function:

  PopUpManager.removePopUp(customerFormPopUp);

  The if statement now appears like the following code:

  if (event.target == _btnSave)
  {
    // Set the id, if it is not auto generated.
    _serviceWrapper.serviceControl.commit(); // Commit the change.
    if (_prevValueObject != null)
      _prevValueObject = null;
    PopUpManager.removePopUp(customerFormPopUp);
  }

5  Save and close CustomerForm.mxml.

Modify CustomerFormPopup.mxml:

1  Locate the <forms:CustomerForm> element, and make the following changes:

   - Delete the valueObject attribute.
   - Add the following attribute: customerFormPopUp="{this}"
   - Change the value of the id attribute to edit_custom_info_frm.

   The element should look like the following code:

   <forms:CustomerForm id="edit_custom_info_frm" customerFormPopUp="{this}">

2  Add the following code to the setup function:

   edit_custom_info_frm.valueObject=customer;
   edit_custom_info_frm.executeBindings(true);

3  Save and close CustomerFormPopup.mxml.

4  Run the CustomerDashboard application, and click the Details button in the Customer Information area to open the form.

5  Use the form to change the profile information, and then notice the Customer Information area reflects the changes.
Next Steps

In this module of the tutorial, you learned how to create Flash Builder applications that use model-generated data management services:

- You used the model to generate service wrapper and value object ActionScript classes.
- You bound a DataGrid instance to a service operation.
- You created a filter and applied the associated service operation to the DataGrid.
- You customized the form template and generated a form for interacting with the customer table in the database.

You can continue to the “Server-Side Configuration and Programming” on page 36 module or any other module in the tutorial:

- Previous modules: “Create the Data Model” on page 11
- Jump ahead: “Packaging and Deploying the Application” on page 50
- Read descriptions of each module: “Tutorial Overview” on page 1
Chapter 5: Server-Side Configuration and Programming

In this module of the tutorial you integrate Data Services with Spring security. You configure the server such that authentication requests are sent to the Spring message broker. You will also learn how to retrieve role information for implementing authorization using client logic.

The Spring framework is a Java platform that provides infrastructure support for your Java applications. The framework enables you to build plain old Java objects (POJO) that easily integrate with enterprise services. For information about Spring, go to http://www.springsource.org/.

Secondly, you generate source files for the server-side Java assembler classes. After customizing the assemblers with more authorization logic, you will deploy them to the server.

Setting up

Before you perform the exercises in this module, you need to complete the “Getting Started” on page 3 module. You will be using the Tutorial_GS Flash Builder project, the tutorialApp web application, and tutorialdb database that you created.

The application that you are developing uses libraries that Spring BlazeDS Integration 1.0.2 (with dependencies) provides. You can download the files from http://www.springsource.com/download/community. However, the Engineering Support Center sample application provides the files (in the WAR file).

You also need to have downloaded the tutorial files.

Import source files:

1. In Flash Builder, right-click Tutorial_GS and click Import.
2. Select General > File System and click Next.
3. Browse for the spring/start folder of the tutorial files.
4. Select all the files and folders below the start folder.
5. Select Overwrite Existing Resources Without Notifying, and click Finish.

Replace the database script:
This procedure creates tables and adds example data to the tutorialdb database.

1. If the database server is running, stop the database server using one of the following scripts, depending on your operating system:
   - (Windows) [lcds_root]/sampledb/stopdb.bat
   - (Unix-based) [lcds_root]/sampledb/stopdb.sh
2. Copy the spring/dbscript/tutorialdb.script file and replace the file of the same name in the [lcds_root]/sampledb/tutorialdb directory.
3. Start the database server using one of the following scripts, depending on your operating system:
   - (Windows) [lcds_root]/sampledb/startdb.bat
Integrate with Spring security

In the following exercises you will configure authentication and authorization on the server. The Data Services server uses the Spring Security module to perform these functions.

Configure Spring MessageBroker integration

Integration with Spring requires that messages from Flash Builder clients are routed to the Spring message broker instead of the Data Services MessageBroker servlet. Three configurations implement this behavior:

- Bootstrap the Spring DispatchServlet.
- Bootstrap the Spring message broker.
- Create the default channel to use.
- Remove incompatible channels.

Notice that there is no need to configure the Spring MessageBroker servlet itself.

If you are not familiar with Data Services channels, see Channels and channel sets in Using Adobe LiveCycle Data Services.

For more information about integrating with Spring, see Spring BlazeDS Integration Reference Guide.

Bootstrap Spring DispatcherServlet:

1. Open the tutorialApp//WEB-INF/web.xml file in a text editor.
2. Insert the following XML after the <webapp> element:

   ```xml
   <context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/spring/*-config.xml</param-value>
   </context-param>

   <servlet>
   <servlet-name>flex</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
   </servlet>

   The configuration code locates the Spring configuration files, declares the DispatcherServlet class (servlet name is flex), and loads the class when the Data Services server starts.

3. Add the following XML below the <servlet> element that you just added. This code maps a URL pattern to the flex servlet. All messages that are directed to the URL pattern are sent to DispatchServlet:

   ```xml
   <servlet-mapping>
   <servlet-name>flex</servlet-name>
   <url-pattern>/messagebroker/*</url-pattern>
   </servlet-mapping>

   4. Below the <servlet-mapping> element, insert the following code that loads the Spring context listener. The ContextLoadListener class bootstraps the Spring root web application context.

   ```xml
   <servlet>
CREATING A LIVECYCLE DATA SERVICES ES2 APPLICATION
Server-Side Configuration and Programming

<listener>
   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

5 Insert the following code inside the RDS servlet definition, after the init-param element:

   <init-param>
      <param-name>messageBrokerId</param-name>
      <param-value>_messageBroker</param-value>
   </init-param>

6 Delete the following code that defines the existing message broker:

   <servlet>
      <servlet-name>MessageBrokerServlet</servlet-name>
      <display-name>MessageBrokerServlet</display-name>
      <servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>
      <init-param>
         <param-name>services.configuration.file</param-name>
         <param-value>/WEB-INF/flex/services-config.xml</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

7 Delete the following code that defines the servlet-url mapping for the message broker that you deleted:

   <servlet-mapping>
      <servlet-name>MessageBrokerServlet</servlet-name>
      <url-pattern>/messagebroker/*</url-pattern>
   </servlet-mapping>

8 Save and close web.xml.

Configure the DispatchServlet and bootstrap MessageBroker:
The MessageBrokerFactoryBean is configured as a bean in the Spring WebApplicationContext by default. The following steps configure the flex servlet to enable the bootstrapping of the message broker.

1 Create a text file in the tutorialApp/WEB-INF directory named flex-servlet.xml.
   This file configures the flex servlet that you declared in the previous procedure.

2 Open the file in a text editor and add the following XML:

   <beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:flex="http://www.springframework.org/schema/flex"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://www.springframework.org/schema/flex
      http://www.springframework.org/schema/flex/spring-flex-1.0.xsd">

   </beans>

   These namespace definitions and schema locations enable access to the default MessageBrokerFactoryBean configuration.

3 Add the following message-broker element before the closing beans element. The message-broker element causes the message broker beans to be automatically registered:
<flex:message-broker/>
</flex:message-broker>

4 Save and close flex-servlet.xml.

Create the channel definition

The following configuration in the tutorialApp/WEB-INF/flex/services-config.xml file defines the default channel that the message broker uses. Also, Spring BlazeDS Integration 1.0.2 does not support non-servlet based endpoints. Remove the default NIO and RTMP endpoints from your tutorialApp web application.

1 Open the tutorialApp/WEB-INF/flex/services-config.xml file in a text editor.
2 Change the default channel to my-amf-stream:
   
   ```xml
   <default-channels>
      <channel ref="my-amf-stream"/>
   </default-channels>
   ```
3 Add the following channel definition inside the channels element:
   
   ```xml
   <channel-definition id="my-amf-stream" class="mx.messaging.channels.StreamingAMFChannel">
      <endpoint url="http://{server.name}:{server.port}/{context.root}/messagebroker/streamingamf" class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
   </channel-definition>
   ```
4 Delete the following elements (and their contents) from the tutorialApp/WEB-INF/flex/services-config.xml file:
   
   - `<channel-definition id="my-rtmp" class="mx.messaging.channels.RTMPChannel">`
   - `<channel-definition id="my-nio-amf" class="mx.messaging.channels.AMFChannel">`
   - `<channel-definition id="my-nio-amf-poll" class="mx.messaging.channels.AMFChannel">`
   - `<channel-definition id="my-nio-http" class="mx.messaging.channels.HTTPChannel">`
   - `<channel-definition id="secure-nio-amf" class="mx.messaging.channels.SecureAMFChannel">`
   - `<channel-definition id="secure-nio-http" class="mx.messaging.channels.SecureHTTPChannel">`
5 Save and close the services-config.xml file.

Configure default channel sets:
Configure the Data Management and messaging services to use the my-amf-stream channel by default

1 Use a text editor to open the tutorialApp/WEB-INF/flex/data-management-config.xml file.
2 Change the channel ref to my-amf-stream:
   
   ```xml
   <default-channels>
      <channel ref="my-amf-stream"/>
   </default-channels>
   ```
3 Save and close the data-management-config.xml file.
4 Use a text editor to open the tutorialApp/WEB-INF/flex/messaging-config.xml file.
5 Change the channel ref to my-amf-stream:
   
   ```xml
   <default-channels>
      <channel ref="my-amf-stream"/>
   </default-channels>
   ```

Last updated 6/29/2010
6 Save and close the messaging-config.xml file.

**Configure Spring security**

The sample application uses the Spring security infrastructure for user authentication and authorization. Authentication is performed against the user name that is provided when logging in. Each user name is associated with a role which determines the dashboard that is presented.

With the following configurations, Spring security integration is detected and the required support files are automatically installed.

**Declare the Spring security filter chain:**
The following configuration enables the Spring security filter chain:

1 Open tutorialApp/WEB-INF/web.xml in a text editor.
2 Add the following XML below the `webapp/context-param` element:

```xml
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```
3 Save and close web.xml.

**Establish Spring security integration:**
Spring security integration is established using the following configuration.

1 Open tutorialApp/WEB-INF/flex-servlet.xml in a text editor.
2 Add the following element inside the `flex:message-broker` element:

```xml
<flex:secured />
```
3 Save and close flex-servlet.xml.

**Identify the URLs to secure:**
Create the tutorialApp/WEB-INF/spring directory.

1 Create a text file in the new spring directory named security-config.xml and open the file in a text editor.
2 Copy the following XML to the file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:sec="http://www.springframework.org/schema/security"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security-3.0.xsd">
</beans>
```
Insert the following `<sec:http>` element inside the `<beans>` element:

```xml
<sec:http entry-point-ref="preAuthenticatedEntryPoint">
    <sec:intercept-url pattern="/secured/**" access="IS_AUTHENTICATED_FULLY"/>
</sec:http>
```

Save `security-config.xml`. Do not close the file.

Declare the authentication manager:

The configuration in the `tutorialApp/WEB-INF/spring/security-config.xml` file identifies the authentication manager:

1. Copy the following code to the `security-config.xml` file, below the `<sec:http>` element:

```xml
<bean id="preAuthenticatedEntryPoint" class="org.springframework.security.web.authentication.Http403ForbiddenEntryPoint"/>
<sec:authentication-manager>
    <sec:authentication-provider user-service-ref="sampleUserDetailsService"/>
</sec:authentication-manager>

<bean id="sampleUserDetailsService" class="com.adobe.lcds.callcentersample.security.SampleUserDetailService">
    <property name="dataSource" ref="dataSourceUser"/>
</bean>
<bean id="flexAccessDecisionManager" class="org.springframework.security.access.vote.AffirmativeBased">
    <property name="decisionVoters">
        <list>
            <bean class="org.springframework.security.access.vote.RoleVoter"/>
            <bean class="org.springframework.security.access.vote.AuthenticatedVoter"/>
        </list>
    </property>
</bean>
```

The `SampleUserDetailService` class implements the `org.springframework.security.core.userdetails.UserDetailsService` interface. This class queries the `tutorialdb` database for user names, passwords, and role names and returns a `User` object that contains the results.

2. From the tutorial files, copy the `spring/classes_security/com` directory (and all child directories) to the `tutorialApp/WEB-INF/classes` directory.

Create the `dataSourceUser` data source:

The `dataSourceUser` data source provides information about users. The `dataSourceUser` data source is defined in the `tutorialApp/WEB-INF/spring/infrastructure-config.xml` file.

1. Create a text file in the `tutorialApp/WEB-INF/spring` directory named `infrastructure-config.xml` and open the file in a text editor.

2. Copy the following XML to the file:
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
 <bean id="dataSourceUser"
 class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
  <property name="url" value="jdbc:hsqldb:hsql://localhost:9002/tutorialdb" />
  <property name="username" value="sa"/>
  <property name="password" value=""/>
 </bean>
</beans>

Save and close infrastructure-config.xml.

Initialize session security

Problems occur when logging in as multiple users in the same browser session. To enable multiple users to log in, use
the login manager of the message broker to log off the session.

The com.adobe.lcds.callcentersample.security.LogoutService class that is provided implements this strategy:

```java
package com.adobe.lcds.callcentersample.security;

import flex.messaging.MessageBroker;

public class LogoutService {
    public void doLogout() {
        MessageBroker mb = MessageBroker.getMessageBroker("_messageBroker");
        mb.getLoginManager().logout();
    }
}
```

Now, configure the server so that the class is available through the Remoting Service.

**Note:** You deployed the LogoutService class to your web application in the previous exercise when you copied the
spring/classes_security/com directory.

Create the destination for LCDS remoting:

1. Open the tutorialApp/WEB-INF/flex-servlet.xml file in a text editor.
2. Add the following remoting destination below the `flex:message-broker` element:

   ```xml
   <flex:remoting-destination ref="logoutService"
   channels="my-amf-stream,my-polling-amf,my-amf"/>
   ```
3. Save and close the flex-servlet.xml file.

Expose the logout service bean

1. Create a text file in the tutorialApp/WEB-INF/spring directory named app-config.xml.
2. Open the file in a text editor and add the following code:
<xml version="1.0" encoding="UTF-8">
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">
  
  3  Add the following bean element inside the beans element:  
  <bean id="logoutService" 
     class="com.adobe.lcds.callcentersample.security.LogoutService">  
  </bean>  
  
  4  Save and close the app-config.xml file.  
  The WelcomeScreen.mxml file already contains the following init function that calls the service:  
  private function _init():void  
  {  
    var token:AsyncToken = logoutService.doLogout();  
    token.addResponder(new mx.rpc.Responder(_logoutSuccess, _logoutFail));  
  }

Deploy Spring libraries

The final step in configuring the server for Spring integration is deploying the Spring libraries to the server. Library files that your server-side Java classes reference must be saved in the tutorialApp\WEB-INF\lib directory on the server.

Deploy the libraries:
❖ Extract the following JAR files from the tutorialApp/WEB-INF/lib directory of the callcenter-ext.war file to the tutorialApp/WEB-INF/lib directory:

The following libraries are required. However, for simplicity you can copy all of the libraries.

• com.springsource.org.aopalliance-1.0.0.jar
• com.springsource.org.aspectj.runtime-1.6.3.RELEASE.jar
• com.springsource.edu.emory.mathcs.backport-2.2.0.jar
• com.springsource.org.codehaus.jackson-1.0.0.jar
• org.springframework.aop-3.0.0.RELEASE.jar
• org.springframework.asm-3.0.0.RELEASE.jar
• org.springframework.beans-3.0.0.RELEASE.jar
• org.springframework.context-3.0.0.RELEASE.jar
• org.springframework.context.support-3.0.0.RELEASE.jar
• org.springframework.core-3.0.0.RELEASE.jar
• org.springframework.expression-3.0.0.RELEASE.jar

Last updated 6/29/2010
Configure login prompts on the client

In this exercise you review how authentication is implemented in the client application. It will be beneficial for you to open the WelcomeScreen.mxml file from the com.adobe.lcds.demo.login package of the Tutorial_GS project and study the code as it is discussed.

WelcomeScreen.mxml contains a ViewStack instance (id="myviewstack", line 301) with several views defined:

- login_view is the default view that displays the user name and password fields.
- agent_view, cust_view, and supervisor_view include the UI that is displayed to agents, customers, and supervisors.

When a user logs in, they are authenticated, and their user role is determined. The view that is associated with the role is then displayed. To authenticate users and retrieve their role, the following pattern is used:

1. The destination for the CustomerService service is retrieved and stored in a ChannelSet object (_chSet).
2. The login function of the ChannelSet object is called and an AsyncToken object is returned (loginToken).
3. A Responder object is created (responder) with event listeners registered on its result and fault events (_handleLoginSuccess and _handleLoginFail).
4. The Responder object is added to the AsyncToken object.

The login_view view contains two TextInput components with id’s of userID and password. The click event of a Button component (id="login_btn") calls the _loginHandler function (line 140) that authenticates the credentials that users enter in the TextInput components:
private function _loginHandler():void
{
    usrName = userId.text.toLowerCase();
    passwd = password.text;
    login_fail_txt.visible = false;
    if(_chSet == null)
    {
        _chSet = ServerConfig.getChannelSet(customerService.destination);
    }

    if(!_chSet.authenticated)
    {
        var loginToken:AsyncToken = _chSet.login(usrName, passwd);
        var responder:mx.rpc.Responder = new mx.rpc.Responder
        {
            _handleLoginSuccess,
            _handleLoginFail
        };
        loginToken.addResponder(responder);
    }
}

Run the application and log in:
1 In Flash Builder, run WelcomeScreen.mxml from the com.adobe.lcds.demo.login package.
2 Log in using any of the following user names and passwords. Log out and log in as different users to open different dashboards.

<table>
<thead>
<tr>
<th>User name</th>
<th>Password</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>atanaka</td>
<td>password</td>
<td>Supervisor</td>
</tr>
<tr>
<td>jacobs</td>
<td>password</td>
<td>Customer</td>
</tr>
<tr>
<td>kbowman</td>
<td>password</td>
<td>Customer</td>
</tr>
<tr>
<td>srose</td>
<td>password</td>
<td>Agent</td>
</tr>
<tr>
<td>csimms</td>
<td>password</td>
<td>Agent</td>
</tr>
</tbody>
</table>

Customize the assembler classes

When models are deployed to the server, assembler classes that correspond with model entities are created automatically. The assemblers are Java classes that, by default, reside only in server memory.

Assemblers are Java classes that the Data Management service uses to interact with the data source. (See About the Data Management Service.)

In the following exercises you will generate the Java source files for the assembler classes. After customizing them, you will deploy the Java classes to the server.
**Customize the Java assembler template**

In the “Developing the Data-Centric Client Application” on page 22 module, you extracted the template that defines the code that is generated for model-driven forms. Similarly, in this exercise you will modify the template that defines how Java code is created for server-side assembler classes.

Specifically, you will modify the template so that assemblers use java.util.Logger to log any create, update, and delete (CRUD) activities.

**Customize the Java code template:**

1. In Flash Builder, click Window > Preferences.
2. In the Preferences tree, select Adobe > Data Model > Code Generation.
3. For Server-Side Java Generation, select Default, and click OK.
4. Right-click Tutorial_GS.fml and select Data Model > Extract Templates For Active Generators.
5. In the dialog box, select No To All so that existing templates are not replaced.
6. Open templates > java > JavaAssemblerSkeleton.ftl.
7. Add the following code after `<#assign classbody>` (near line 30):

   ```java
   import java.util.logging.Logger;
   ```

8. Locate the code that defines the following class declaration:

   ```java
   public class ${classname} extends ${typeHelper.addImport("fiber.data.assemblers.FiberAssembler", false)}
   ```

   Add the following line of code after the class declaration:

   ```java
   private static final Logger logger = Logger.getLogger(${classname}.class.getSimpleName());
   ```

9. Locate the code that generates the `createItem` method and add the following code as the first line in the method:

   ```java
   logger.info("Creating new "+ ${type}.class.getSimpleName());
   ```

   The method should appear as the following code:

   ```java
   public void createItem(Object item)
   {
   logger.info("Creating new "+ ${type}.class.getSimpleName());
   // ${type} myItem = (${type}) item;
   super.createItem(item);
   }
   ```

10. Repeat step 8 for the `deleteItem` and `updateItem` methods:

    - `deleteItem`:
      ```java
      logger.info("Deleting "+ ${type}.class.getSimpleName());
      ```

    - `updateItem`:
      ```java
      logger.info("Updating "+ ${type}.class.getSimpleName());
      ```

   The methods should appear as the following code:
public void deleteItem(Object item)
{
    logger.info("Deleting " + ${type}.class.getSimpleName());
    super.deleteItem(item);
}

/**
 * Updates an item.
 * @param newVersion the updated item
 * @param previousVersion the previous version of the item
 * @param updatedProperties an optional list of properties that were updated
 */
@Override
public void updateItem(Object newVersion, Object previousVersion, List updatedProperties)
{
    logger.info("Updating " + ${type}.class.getSimpleName());
    super.updateItem(newVersion, previousVersion, updatedProperties);
}

11 Save and close JavaAssemblerSkeleton.ftl.

Generate the assembler source files:
1 In Flash Builder, open Tutorial_GS.fml. Switch to the Data Model perspective.
2 Select the Customer entity, and in the Properties view click the Code Gen tab.
3 In the Assembler Package box, enter com.adobe.lcds.callcentersample.assembler.customer.
4 In the Java Generation section, for Generate Assembler, select Yes.
5 Repeat steps 3 and 4 for all other entities. Use the following package names:
   - Note: com.adobe.lcds.callcentersample.assembler.notes
   - CustCase: com.adobe.lcds.callcentersample.assembler.cases
   - Agent: com.adobe.lcds.callcentersample.assembler.agent
   - User: com.adobe.lcds.callcentersample.assembler.user
6 Save the model.
   The source files for the assembler classes are added to the Tutorial_GS project in the java folder.
7 Open java/com/adobe/lcds/callcentersample/assembler/customer/CustomerAssembler.java.
   The CRUĐ methods include the logging code.

Override the CRUĐ methods
In this exercise, you further customize the generated assembler classes to make sure users are authorized to perform
CRUĐ operations. The assemblers check the authorization details that are contained in the Spring ServletContext.

The Spring SecurityContext object contains the security context for the application. This object contains an
Authentication object which includes information about the user that is logged in.

The following code retrieves the role of the current user in an array of GrantedAuthority objects.

The Spring authentication manager sets the GrantedAuthority objects that Authority objects contain.
SecurityContext secCtx = SecurityContextHolder.getContext();
Authentication auth = secCtx.getAuthentication();
User principal = (User)auth.getPrincipal();
Collection result = null;
GrantedAuthority authority = null;

if(auth.getAuthorities().iterator().hasNext())
    authority = principal.getAuthorities().iterator().next();

Note: Recall that our users are assigned only one role. Therefore, there is no need to iterate through the array of GetAuthority objects. Instead, the first object in the array is retrieved.

After the role is known, you can use it to create application logic. For example, in our application, we want only customers to be able to create cases. The CustCaseAssembler class is therefore modified so that the create method can be used only by customers.

if("ROLE_CUSTOMER".equals(authority.getAuthority()))
{
...
}

The compiled assembler classes are provided with the tutorial. Copy them to your project and then configure the model to use them.

Copy the assembler class files:
❖ From the tutorial files, copy the spring/classes_assemblers/com directory (and all child directories) to the tutorialApp/WEB-INF/classes directory.

Configure the model:
1 Open TutorialModel.fml.
2 Select the Customer entity, and in the Properties view, select the Data Mgt tab.
3 In the Assembler Information area, enter the following value in the Assembler Class box:
   com.adobe.lcds.callcentersample.assembler.customer.CustomerAssembler
4 Repeat step 3 for the other entities, using the following values:
   • Note:
     com.adobe.lcds.callcentersample.assembler.notes.NoteAssembler
   • CustCase:
     com.adobe.lcds.callcentersample.assembler.cases.CustCaseAssembler
   • Agent:
     com.adobe.lcds.callcentersample.assembler.agent.AgentAssembler
   • User:
     com.adobe.lcds.callcentersample.assembler.user.UserAssembler
5 Save and deploy the model.

Test the custom assemblers:
1 Run the WelcomeScreen.mxml application.
2 Log in with the user name jjacobs and password of password. The Customer Dashboard should open.
3. Click the Details button in the Customer Information area to open the form and use the form to change the profile information.

4. From the [lcds_root]/tomcat/logs directory, open the catalina.log file to see that the update operation is logged.

**Next steps**

In this module of the tutorial, you learned how to integrate with Spring security for handling user authentication and authorization. You also learned how to customize the model-generated assembler classes.

You can continue to the “Packaging and Deploying the Application” on page 50 module, or any other module in the tutorial:

- Previous modules:
  - “Getting Started” on page 3
  - “Create the Data Model” on page 11
  - “Developing the Data-Centric Client Application” on page 22
- Read descriptions of each module: “Tutorial Overview” on page 1
Chapter 6: Packaging and Deploying the Application

In this module of the Creating a LiveCycle Data Services ES2 Application tutorial, you compile the application MXML and Java source files. You package the application in a WAR file and deploy it to the Tomcat J2EE application server.

Setting up

Before you perform the exercises in this module, complete the following exercises in the “Getting Started” on page 3 module:

- “Install required software” on page 3
- “Create a database” on page 4
- “Create the Context” on page 6
- “Start the server” on page 7

You also need to have downloaded the tutorial files. Install other required software as described in the following procedure.

Obtain and install required software:

1. Install JDK 1.5 or later version, and ensure the system path includes the path to the JDK binaries. (See http://java.sun.com/javase/downloads/index.jsp.)
2. Install and configure Apache Ant. (See http://ant.apache.org/manual/install.html.)
5. Download the Streaming API for XML (StAX) libraries from http://stax.codehaus.org/Home.
6. Create a directory for storing the Java libraries that you will include in the application. This directory is referred to as [java_libs].
7. Copy all of the JAR files from the WEB-INF/lib directory of the callcenter-ext.war file to the [java_libs] directory.

Location of JAR files

The following table indicates the location of the JAR files that are compiled with the application.
<table>
<thead>
<tr>
<th>JAR file</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>acrobat-core-charset.jar</td>
<td>/lcds_root/tomcat/webapps/lcds/WEB-INF/lib</td>
</tr>
<tr>
<td>acrobat-core.jar</td>
<td>/lcds_root/tomcat/webapps/lcds/WEB-INF/lib</td>
</tr>
<tr>
<td>asm-attrs.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>asm.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>cfdata services adapter.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>cfgatewayadapter.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>cglib-2.1.2.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>collections-generic.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>commons-codec.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>commons-collections.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>commons-httpclient.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>commons-log.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>concurrent.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>dom4j.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>ehcache-1.1.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>ejb3-persistence.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>fiber-core.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>fiber-lcds.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>fiber-runtime.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>fiber-tools.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-acrobat.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-messaging-common.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-messaging-core.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-messaging-data.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-messaging-data.req.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-messaging-opt.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-messaging-proxy.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-messaging-remoting.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-rds-lcds.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>flex-rds-server.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>freemarker.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>hibernate-annotations.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>hibernate-commons-annotations.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>hibernate3.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>javassist.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>hsqldb.jar</td>
<td>/sampledb</td>
</tr>
<tr>
<td>org.springframework.flex-1.0.2.RELEASE.jar</td>
<td>Spring BlazeDS Integration 1.0.2 (with dependencies) from <a href="http://www.springsource.com/download/community">http://www.springsource.com/download/community</a></td>
</tr>
</tbody>
</table>
CREATING A LIVECYCLE DATA SERVICES ES2 APPLICATION
Packaging and Deploying the Application

Build and package the application

In the following exercises you will become familiar with the files that are provided for building the application. You will then build and deploy the tutorialApp application.

Review the provided files

The package directory of the tutorial files include the source files and configuration files for the application:

- The config.xml file contains properties for compiling MXML files.
- The build directory includes an Ant build file and property file for executing the build.
- The java directory contains the Java source files, for example the custom assembler.

<table>
<thead>
<tr>
<th>JAR file</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.springsource.org.aopalliance-1.0.0.jar</td>
<td>org.springframework.flex-1.0.2.RELEASE.jar</td>
</tr>
<tr>
<td>com.springsource.org.aspectj.runtime-1.6.3.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>com.springsource.edu.emory.mathcs.backport-2.2.0.jar</td>
<td></td>
</tr>
<tr>
<td>com.springsource.org.codehaus.jackson-1.0.0.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.aop-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.asm-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.beans-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.context-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.context.support-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.core-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.expression-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.jmx-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.security.config-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.security.core-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.security.web-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.transaction-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.web-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.web.servlet-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>spring-integration-core-1.0.3.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.web-3.0.0.RELEASE.jar</td>
<td></td>
</tr>
<tr>
<td>com.springsource.org.apache.taglibs.standard-1.1.2.jar</td>
<td></td>
</tr>
<tr>
<td>org.springframework.jdbc-2.5.6.SEC01.jar</td>
<td>Spring Framework 3.0.0 RELEASE from <a href="http://www.springsource.com/download/community?project=Spring%20Framework">http://www.springsource.com/download/community?project=Spring%20Framework</a></td>
</tr>
<tr>
<td>stax-1.2.0.jar</td>
<td><a href="http://stax.codehaus.org/Download">http://stax.codehaus.org/Download</a></td>
</tr>
<tr>
<td>stax-api-1.0.1.jar</td>
<td></td>
</tr>
</tbody>
</table>
The project directory contains the Flash Builder project, including MXML files, the model, and all other supporting files that were added during development.

The web directory includes the directory structure and configuration files for the web application.

**Review the build file**

The build.xml file in the build directory contains the Ant commands that compile the application source files and packages the application in a WAR file. The build directory also contains the build.properties file that you will use to specify property values that are specific to your environment.

- **clean**: Removes files that were created from previous build attempts.
- **prepare**: Creates the directory structure for the application. Copies existing web files from the project/web directory.
- **compile-flex**: Executes the Flex SDK-provided tools to compile the MXML files.
- **html-wrapper**: Uses the Flex SDK-provided html-wrapper Ant command to create the HTML file that wraps the SWF file.
- **copy-resources**: Copies the SWF files, model file, Flex libraries, and other application files to the application directories.
- **copy-locale**: Copies resource bundles to the application.
- **setup-config-file**: Replaces tokens in the config.xml file with values from the build.properties file.
- **compile-java**: Compiles the Java source files.
- **deploy-app**: Copies all files to the web/tutorialApp directory. All of the application files are now assembled.
- **create-war**: Creates the WAR file that you can deploy to Tomcat.

**Run the build**

In this exercise you will use Ant to build the application. Before you run the build, you need to modify the build.properties file to specify information about your environment.

*Note: Make a copy of the package directory from the tutorial files and work with the copied files.*

**Modify build.properties:**

1. Open the package/build/build.properties file in a text editor.

   The file contains property-value pairs, with properties at the beginning of each line. Example values are provided.

2. Provide values for each property:

   - **FLEX_HOME**: The path of the root folder of the Flex SDK, version 4. The Flex SDK is installed with Flash Builder, in the 
     `[Flash Builder home]/sdks/4.0.0` directory.
   - **context.root**: The context root of the application. Leave the value as `tutorialApp`.
   - **j2ee.lib.dir**: The path of the directory that contains J2EE libraries. These libraries are installed with LiveCycle Data Services, in the 
     `[Data Services home]/tomcat/lib` directory.
   - **blazeds.spring.jar.location**: The path of the `[java_libs]` directory that you created in the “Setting up” on page 50 exercise.
   - **lcds.resources**: The path to the Flex SDK 4 files that are installed with Data Services, which is the 
     `[Data Services home]/resources/lcds_swcs/FlexSDK4` directory.
Save and close build.properties.

Run Ant:
1. Open a command prompt and change the current directory to the build directory.
2. Type Ant and press Enter.

Deploy the application

In this exercise you will deploy the WAR file to Tomcat, and run the client application in your web browser.

Before you deploy, the application databases must be created. You created the tutorialdb database when you completed the exercises in the “Getting Started” on page 3 module. You need to also create the tutorialdbarchived database.

Create and start the database:
1. Use a text editor to open the server.properties file from the [Data Services home]/sampledb directory.
2. Above the line server.port=9002, insert the following text:
   
   server.database.x=file:tutorialdbarchived/tutorialdbarchived
   server.dbname.x=tutorialdbarchived

3. In the text that you just inserted, change the two instances of x to the next integer in the sequence of existing database definitions. For example, if the highest integer is 5 (such as server.dbname.5=tutorialdb), then change x to a 6:
   
   server.database.6=file:tutorialdb/tutorialdb
   server.dbname.6=tutorialdb

4. From the tutorial files, copy the package/database/tutorialdb.script file to the [Data Services home]/sampledb/tutorialdb directory. Replace the existing file. The script provides sample data and user accounts for the application.

5. Start the database server using one of the following scripts, depending on your operating system:
   - (Windows) [install directory]/sampledb/startdb.bat
   - (Unix-based) [install directory]/sampledb/startdb.sh

   On Windows, you can click Start > All Programs > Adobe > LiveCycle Data Services ES2 Version 3.1 > Start Samples Database

Deploy the WAR file:
1. If the Data Services server is running, stop the server.

2. If you have an instance of the tutorialApp application in the [Data Services home]/tomcat/webapps directory, move it to a different location in your file system. Moving the tutorialApp directory ensures the WAR file deploys successfully, and creates a backup copy of your development efforts.

3. Copy tutorialApp.war from the dist directory to the [Data Services home]/tomcat/webapps directory.

4. Enter the following command to start the Tomcat server:

   [install directory]/tomcat/bin/catalina run

   On Windows, you can click Start > All Programs > Adobe > LiveCycle Data Services ES2 Version 3.1 > Start LiveCycle Data Services Server
The server is running when the message \texttt{INFO: Server startup in xxxxxxx ms} appears in the command window.

5. Use your web browser to navigate to \texttt{http://[server name]:[port]/tutorialApp}.

For example, if Data Services is installed on the local computer using the default port of 8400, the URL is \texttt{http://localhost:8400/tutorialApp}.

6. Log in using one of the following user accounts.

<table>
<thead>
<tr>
<th>User name</th>
<th>Password</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>atanaka</td>
<td>password</td>
<td>Supervisor</td>
</tr>
<tr>
<td>jjacobs</td>
<td>password</td>
<td>Customer</td>
</tr>
<tr>
<td>kbowman</td>
<td>password</td>
<td>Customer</td>
</tr>
<tr>
<td>srose</td>
<td>password</td>
<td>Agent</td>
</tr>
<tr>
<td>csimms</td>
<td>password</td>
<td>Agent</td>
</tr>
</tbody>
</table>

Next steps

In this module you reviewed the Ant build file and used it to create the WAR file for the tutorialApp application. You also deployed the application to Tomcat.

This is the final module of the tutorial.

- Previous modules:
  - “Getting Started” on page 3
  - “Create the Data Model” on page 11
  - “Developing the Data-Centric Client Application” on page 22
  - “Server-Side Configuration and Programming” on page 36
- Read descriptions of each module: “Tutorial Overview” on page 1

Last updated 6/29/2010