Using the
LiveCycle® Data Services ES2 Sample Application
Version 3.1
Contents

Exploring the Engineering Support Center Sample
Install the sample application .......................................................... 5
Build the sample from the source code ........................................... 7
Working with the source code ......................................................... 9
Feature highlights ......................................................................... 10
Exploring the Engineering Support Center Sample

The Engineering Support Center sample is a Flash Builder application with a LiveCycle Data Services server back end. The application enables customers, agents, and supervisors of GlobalCorp to create and manage customer cases regarding their products.

Customers, call center agents, and supervisors perform different tasks during the life cycle of a case. So, each type of user is presented different clients with slightly different features.

Customers perform tasks such as create cases, add notes, and modify profile information.
Exploring the Engineering Support Center Sample

Case Details:
Title: Touchpad is not working in Xoom Kamoio Beta
Status: Open
Created Date: 04/13/2010
Modified Date: 04/23/2010

Case Notes:
I installed the Kamoio beta on my Xoom 1.0. Then I went into the device preferences, did nothing, and then tried "two-finger scrolling" after closing the typing screen. Scrolling seems to work, but after closing the typing screen, there's no way to go back. It seems like something is stuck in a loop. It's a bug.

Last updated 10/13/2010
Customer dashboard

Agents can perform tasks such as create cases, add notes, change the case status, and create customer accounts. Agents can access only the cases that they are assigned.

Agent dashboard

Supervisors can view all cases, receive alerts about problem cases, and see a pie chart that shows the distribution of cases across agents.
### Exploring the Engineering Support Center Sample

**Title:**

**Description:**

The image shows a screenshot of a support ticket tracking system. The system is used for managing and tracking issues related to engineering support. The screenshot includes a table with columns for Case ID, Title, and Status, among others.

**Example Case:**

1. **Title:** Two-finger scrolling not working in SW# 133 Beta
   - **Agent:** John Doe
   - **Status:** Open
   - **Created:** 04.02.2013
   - **Updated:** 04.07.2013

2. **Title:** Two-finger scrolling not working in SW# 133 Beta
   - **Agent:** Jane Smith
   - **Status:** Closed
   - **Created:** 04.02.2013
   - **Updated:** 04.07.2013

**Notes:**

- Indicates that Xname field in my beta Win 10.3. Then I used help, the Microsoft Preferences to drag and small field “two-finger scrolling”. After checking the dialog, two-finger scrolling does not work. Two-finger scrolling no way for me on Apple is 0.90. There is beta 0.90, it’s weird.

**Last updated:** 10/13/2010
Install the sample application

Complete each of the following tasks to install the Engineering Support Center sample:

1. “Download the sample” on page 5.
2. “Install LiveCycle Data Services” on page 5.
3. “Create the database” on page 5.

Note: The sample is supported only on the Tomcat application server.

Download the sample

The sample is provided as a ZIP file at the LiveCycle Developer Center.

After you download the file, extract the contents to your hard drive:

- callcenter-ext.war: This is the prebuilt application WAR file. You can deploy this sample to your Data Services server.
- callcenter-source.zip: This ZIP file contains the source files of the sample as well as the sample database.

Get the sample:

1. In your web browser, navigate to LiveCycle Developer Center to download the ZIP file.
2. Save the file to your hard drive.
3. Extract the contents of the ZIP file.

Note: In the rest of this document, the directory where you extracted the file contents is referred to as [install_location].

Install LiveCycle Data Services

The LiveCycle Data Services 3.1 installation includes the Tomcat application server and HSQLDB database. You will deploy the sample on this application server and store run-time data using this database server.

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

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

Create the database

The sample application uses two databases to store run-time data:

- callcenterext: Stores data about current customer cases.
- callcenterarchived: Stores data about archived customer cases.

The sample includes an HSQLDB database with sample data. An updated server properties file is also provided. The file includes definitions of the new databases and includes definitions of the existing databases that are included with Data Services. You will replace the installed server properties file with the file that comes with the sample.
Note: The following procedure removes any database definitions that you have added to the server.properties file of your Data Services installation. After you complete this procedure, add your database definitions again.

Create the sample databases:
1. Extract the sampledb directory from callcenter-source.zip.
2. Copy the contents of the sampledb directory to the [lcds_root]/sampledb directory. Overwrite existing files that have duplicate names.
3. Start the database server using one of the following scripts, depending on your operating system:
   - (Windows) sampledb/startdb.bat
   - (Unix-based) sampledb/startdb.sh

Deploy the sample application
Use the following procedure to install the application to the Tomcat server that is installed with LiveCycle Data Services.

Create the data source:
1. Open a text editor and create a document.
2. Copy the following data source text to the document:

   `<Context privileged="true" antiResourceLocking="false" antiJARLocking="false" reloadable="true">
   <!-- JOTM -->
   <Transaction factory="org.objectweb.jotm.UserTransactionFactory" jotm.timeout="60"/>
   <Resource name="jdbc/callcenterext" auth="Container"
   type="javax.sql.DataSource" username="sa"
   password="" driverClassName="org.hsqldb.jdbcDriver"
   url="jdbc:hsqldb:hsql://<HOSTNAME>:9002/callcenterext" />
   <Resource name="jdbc/callcenterarchived" auth="Container"
   type="javax.sql.DataSource" username="sa"
   password="" driverClassName="org.hsqldb.jdbcDriver"
   url="jdbc:hsqldb:hsql://<HOSTNAME>:9002/callcenterarchived" />
   </Context>`

3. In the text that you copied, change the two instances of `<HOSTNAME>` to the name of your server.
4. Save the document as callcenter-ext.xml in the tomcat/conf/Catalina/localhost directory of your Data Services installation.

Deploy and run the application:
1. Create a directory named callcenter-ext in the [lcds_root]/tomcat/webapps directory.
2. Extract the contents of callcenter-ext.war to the callcenter-ext directory.
3. Open a command prompt or shell command window.
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 INFO: Server startup in xxxxxxx ms appears in the command window.

5 Use your web browser to navigate to http://[server name]:[port]/callcenter-ext/pages/WelcomeScreen.html.

For example, if Data Services is installed on the local computer using the default port of 8400, the URL is http://localhost:8400/callcenter-ext/pages/WelcomeScreen.html.

Log in
Several user accounts are defined in the sample database. The users are associated with different roles.

<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>

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

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

Build the sample from the source code
The sample source code is provided for you to explore and change to learn how to use product features.

Obtain and install required software:
1 Install JDK 1.5 or later version, and the system path includes the path to the JDK binaries.
2 Install and configure Apache Ant. (See http://ant.apache.org/manual/install.html.)
3 Install LiveCycle Data Services 3.1. (See “Install LiveCycle Data Services” on page 5.)
4 If you don’t have Flash Builder 4.0 installed, download and unzip Flex SDK 4.0. (See http://opensource.adobe.com/wiki/display/flexsdk/Flex+SDK.)

Assemble required libraries:
1 Create a directory on your hard drive for storing the required libraries.

   Later in this document, this directory is referred to as [libraries_dir].

2 Copy all of the JAR files from the WEB-INF/lib directory of the callcenter-ext.war file to the [libraries_dir] directory.
Configure and run the Ant build:
1. Unzip the callcenter-source.zip file to a directory on your hard drive.
   Later in this document, this directory is referred to as [callcenter-source].
2. In a text editor, open [callcenter-source]/build/build.properties.
   Each line of this file represents a property-value pair. The format of each pair is [property name] [value].
3. Replace the values of each property:
   - **FLEX_HOME**: The path of the root directory of your Flash Builder 4.0 directory, for example "C:/program files/Adobe/Adobe Flash Builder 4".
   - **context.root**: The context root of the sample application. This value remains callcenter-ext.
   - **j2ee.lib.dir**: The path of the tomcat/lib directory of your Data Services installation, for example C:/lcds/tomcat/lib.
   - **blazeds.spring.jar.location**: The path of the [libraries_dir] directory.
   - **sample.db.directory**: The path of the sample database directory, for example C:/sampledb.
   - **lcds.resources**: The path to the [lcds_root]/resources/lcds_swcs/FlexSDK4 directory.
   *Note: If a property value includes spaces, place quotation marks around the value.*
4. Open a command prompt and change the current directory to [callcenter-source]/build.
5. Type `ant` and press enter.

**Location of JAR files**
The following table indicates the location of the JAR files that are compiled with the sample 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></td>
</tr>
<tr>
<td>acrntlr-2.7.5H3.jar</td>
<td></td>
</tr>
<tr>
<td>asm-attrs.jar</td>
<td></td>
</tr>
<tr>
<td>asm.jar</td>
<td></td>
</tr>
<tr>
<td>cfdataservicesadapter.jar</td>
<td></td>
</tr>
<tr>
<td>cgatewayadapter.jar</td>
<td></td>
</tr>
<tr>
<td>cglib-2.1.2.jar</td>
<td></td>
</tr>
<tr>
<td>collections-generic-4.01.jar</td>
<td></td>
</tr>
<tr>
<td>commons-codec-1.3.jar</td>
<td></td>
</tr>
<tr>
<td>commons-collections-2.1.1.jar</td>
<td></td>
</tr>
<tr>
<td>commons-httpclient-3.0.1.jar</td>
<td></td>
</tr>
<tr>
<td>commons-logging.jar</td>
<td></td>
</tr>
<tr>
<td>concurrent.jar</td>
<td></td>
</tr>
<tr>
<td>dom4j-1.6.1.jar</td>
<td></td>
</tr>
<tr>
<td>ehcache-1.1.jar</td>
<td></td>
</tr>
<tr>
<td>ejb3-persistence.jar</td>
<td></td>
</tr>
<tr>
<td>fiber-core.jar</td>
<td></td>
</tr>
<tr>
<td>fiber-lcds.jar</td>
<td></td>
</tr>
<tr>
<td>fiber-runtime.jar</td>
<td></td>
</tr>
<tr>
<td>fiber-tools.jar</td>
<td></td>
</tr>
<tr>
<td>flex-acrobat.jar</td>
<td></td>
</tr>
<tr>
<td>flex-messaging-common.jar</td>
<td></td>
</tr>
<tr>
<td>flex-messaging-core.jar</td>
<td></td>
</tr>
<tr>
<td>flex-messaging-data.jar</td>
<td></td>
</tr>
<tr>
<td>flex-messaging-data-opt.jar</td>
<td></td>
</tr>
<tr>
<td>flex-messaging-proxy.jar</td>
<td></td>
</tr>
<tr>
<td>flex-messaging-remoting.jar</td>
<td></td>
</tr>
<tr>
<td>flex-rds-lcds.jar</td>
<td></td>
</tr>
<tr>
<td>flex-rds-server.jar</td>
<td></td>
</tr>
<tr>
<td>freemarker-2.3.9.jar</td>
<td></td>
</tr>
<tr>
<td>hibernate-annotations.jar</td>
<td></td>
</tr>
<tr>
<td>hibernate-commons-annotations.jar</td>
<td></td>
</tr>
<tr>
<td>hibernate3.jar</td>
<td></td>
</tr>
<tr>
<td>javassist.jar</td>
<td></td>
</tr>
<tr>
<td>hsqldb.jar</td>
<td>[lcds_root]/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>
Source code for the Flash client applications and the Java server-side classes is provided in the callcenter-source.zip file.

**Import the Flash Builder project**

The source code of the sample client application includes a Flash Builder project. Import the project to modify or explore the source.

**Note:** If you have not deployed the sample application, follow the instructions to start the database, create the data source, and deploy the application.

To obtain Flash Builder 4.0, visit the Adobe website. You can try Flash Builder without purchasing a license.

When you installed Data Services, the application modeler plugin for Flash Builder was copied to your hard drive. Install the plugin file to your Flash Builder 4.0 installation. (See *Installing Adobe LiveCycle Data Services ES2 Version 3.1*.)

**Import and configure the Flex project:**

1. Start Flash Builder.
2. Click File > Import > Flash Builder Project.
3. Select Project Folder, browse for the [install_location]/callcenter-source/src/callcenter-flex directory, and then click Finish.
4. Right-click the callcenter-flex project and select Properties.
5. In the property tree, select Flex Server and configure the following properties:
Using the LiveCycle Data Services ES2 Sample Application

Exploring the Engineering Support Center Sample

Root Folder: [lcds_root]/tomcat/webapps/callcenter-ext

Root URL: http://[server name]:[port]/callcenter-ext

For example, if Data Services is installed on the same computer as Flash Builder, and uses the default ports, Root URL is http://localhost:8400/callcenter-ext.

Context Root: /callcenter-ext

6 Click Validate Configuration
7 For Output Folder, browse for the [lcds_root]/tomcat/webapps/callcenter-ext/pages directory.
8 Click Finish.

To run the sample application in Flash Builder:
1 In the Package Explorer view, expand the callcenter-flex/src/com.adobe.lcds.demo/login package.
2 Right-click WelcomeScreen.mxml and click Run Application.

Import the Eclipse project

The sample includes the source code of the Java classes that are executed on the server. You can open and edit Java files in Flash Builder.

The sample includes a project file that you can import into Eclipse.

Feature highlights

The following topics briefly describe some of the LiveCycle Data Services features that are used in the Engineering Support Center sample.

Spring integration
The sample application integrates with the Spring framework to implement several behaviors.

Spring MessageBroker integration
HTTP messages from the Flex clients are routed through the Spring DispatcherServlet to the Spring-managed MessageBroker. Three files are configured to implement this behavior.

The following configurations are the minimum required changes for integrating with Spring. For more information, see the Spring BlazeDS Integration Reference Guide.

Spring server declaration
The callcenter-ext/WEB-INF/web.xml file contains the following configuration to identify the Spring configuration files and the DispatchServlet class:
<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>

Bootstrap MessageBroker
To bootstrap MessageBroker, the MessageBrokerFactoryBean is configured as a bean in the Spring WebApplicationContext. Also, MessageBrokerHandlerAdapter and a HandlerMappingSimpleUrlHandlerMapping are configured to route incoming requests to the Spring-managed MessageBroker.

The message-broker tag in the bean definition file causes these beans to be registered automatically. The callcenter-ext/WEB-INF/flex-servlet.xml file includes the configuration:

```xml
<flex:message-broker>
   <flex:secured />
</flex:message-broker>
```

Channel set
The following configuration in the callcenter-ext/WEB-INF/flex/services-config.xml file defines the channels that route message requests to MessageBroker:

```xml
<default-channels>
   <channel ref="my-amf-stream"/>
</default-channels>
```

Spring Remoting
The Spring bean ArchivalService is exposed as remoting destination via Spring. In the sample, this destination is called from the Agent Dashboard to archive closed customer cases. ArchivalService archives cases to a data-source that is separate from the one that stores the application real-time data.

💡 To persist the XML data using a different service, you would use a call to any other web service as required. You would replace the spring bean with any other bean implementation dynamically at run-time using the Spring framework.

Remoting destination
The archivalService destination for the archivalService Spring bean is defined in the callcenter-ext/WEB-INF/flex-servlet.xml file. The associated channels are listed in the preferred order:

```xml
<flex:remoting-destination ref="archivalService"
   channels="my-amf-stream,my-polling-amf,my-amf"/>
```

Bean declaration
The class that implements archivalService Spring bean is identified in the callcenter-ext/WEB-INF/spring/app-config.xml file. The following configuration exposes the bean as a remoting destination:

```xml
<bean id="archivalService"
   class="com.adobe.lcds.callcentersample.archival.ArchivalService">
   <constructor-arg ref="dataSource" />
</bean>
```

The class file is saved in the callcenter-ext/WEB-INF/classes directory.
Data source
The data source named dataSource that is referenced in the app-config.xml file is defined in the callcenter-ext/WEB-INF/spring/infrastructure-config.xml file:

```xml
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9002/callcenterarchived"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
</bean>
```

Spring Messaging
The sample application uses the Spring messaging infrastructure for sending alerts about new cases to the Supervisor Dashboard. The alerts appear in a DataGrid with the id of alert_dg.

The standard Flex Publisher-Consumer components are used for publishing and consuming. The client is not aware that the Spring messaging framework is used to send messages.

The following messaging destination is configured in the callcenter-ext/WEB-INF/flex-servlet.xml file:

```xml
<flex:message-destination id="alert"
    channels="my-amf-stream"
    send-security-constraint="customr-only-access"
    subscribe-security-constraint="supervisor-only"/>
```

This configuration causes the use of Spring messaging for the Publishers and Consumers that are subscribed to the alert destination. The security constraints make sure that only customers can send messages, and only supervisors can consume them. The Spring security configuration defines these roles.

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.

When the server starts, Spring security integration is detected and the required support files are automatically installed.

Spring security filter chain
The following configuration is added to callcenter-ext/WEB-INF/web.xml to enable the Spring security filter chain:

```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>
```

Integration
Spring security integration is established using the following configuration in the callcenter-ext/WEB-INF/flex-servlet.xml file:
Authentication provider
The configuration in the callcenter-ext/WEB-INF/spring/security-config.xml file identifies the authentication provider.

```xml
<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>
```

The SampleUserDetailService class extends UserDetailsService that Spring security provides to authenticate users based on their details in the database. The dataSourceUser data source provides information about users.

The SampleUserDetailService class is saved in the callcenter-ext/WEB-INF/classes directory. The dataSourceUser data source is defined in the callcenter-ext/WEB-INF/spring/infrastructure-config.xml file.

Spring bean for case assignments
When a customer creates a case, it is assigned to the agent that has the lowest number of assigned cases. The CallCenterDAO Spring bean retrieves the agent information from the database.

A custom assembler class looks up the Spring bean using ServletContext, and then calls the bean. The following changes are implemented to configure the Spring bean.

Bean declaration
The bean is declared in the callcenter-ext/WEB-INF/spring/app-config.xml file:

```xml
<bean id="callCenterDAOBean" class="com.adobe.lcds.callcentersample.dao.CallCenterDAO">
    <property name="dataSource" ref="dataSourceUser" />
</bean>
```

The CallCenterDAO class file is saved in the callcenter-ext/WEB-INF/classes directory.

Data source
The dataSourceUser data source is defined in the callcenter-ext/WEB-INF/spring/infrastructure-config.xml file:

```xml
<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/callcenterext" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

Code generation
ActionScript client classes and Java server-side classes are generated from application models.
Model-driven forms
The template that is used to generate forms from models is modified to use Spark components. The mx and s namespaces are added and used to distinguish Spark and Halo components. The template file is ASModelDrivenForm.ftl.

For more information, see Generating a model-driven form.

Custom assemblers
Server-side Java assembler classes are automatically generated from the model. The source code for the assemblers can be optionally generated. Code generation is based on the template file named JavaAssemblerSkeleton.ftl.

The template in the sample is modified so that assemblers use java.util.Logger to log any create, update, and delete (CRUD) activities.

The generated assembler classes are further customized to make sure that users are authorized to perform CRUD operations. The assemblers check the authorization details that are contained in the Spring ServletContext.

For more information, see Implementing an assembler method on the server.

Modelling features
The following model features are configured to automatically generate code that implements the features.

Variants
The model in the sample application defines a single entity to represent both agent and supervisor roles. A variant is used to differentiate the roles, based on the isSupervisor boolean property.

This variant is used to create a filter (getOnlyAgents) on the Agent entity. The corresponding service operation retrieves lists of agents or supervisors from the database.

For more information, see variant.

Externalized strings
Style captions can be used as form labels. The sample uses style captions that reference strings in a resource bundle. The user selects their locale when logging in. The corresponding bundle is used when the form is rendered.

For more information, see style message group.

Value validation
Styles are used to provide validation expression to validate the value of the email property of the Customer entity. When a model-driven form is created from the Customer entity, value provided in the e-mail form field must conform to the validation criteria.

CustomerForm is a model-driven form that is based on the Customer entity, and uses the email validation. This form is used to create and edit customers.

For more information, see style validation.

Derived properties
Entity properties can be derived from other properties of the same entity. The fullName property of the Customer entity is derived from the firstName and lastName properties.
When a model-driven form is created, the form item for the derived property is read-only. On the CustomerForm form, the value of the fullName field is generated automatically from the values provided in the firstName and lastName fields.

For more information, see property.

**Constraints**
Constraints can be defined for entity properties. Constraints make sure that corresponding create and update service operations are successful only under certain circumstances. Constraints are implemented in validate functions of assembler classes.

The sample application includes a constraint called ProperAddress in the Customer entity. This constraint ensures that the user has included location information in their profile.

For more information, see constraint.

**Blob type**
Entity properties can be of type blob. The pic property of the Customer entity is a blob that represents the customer’s digital photograph.

For more information, see property.

**Data caching**
In the sample application, the Cache-items property of the model is set to true (the default setting).

Caching prevents data loss when clients lose their server connection during create and update requests. When the server connection is reestablished, the cached data is pushed to the server.

For example, while a customer fills the form to create a case, network problems cause the client application to lose the server connection. The customer clicks OK to submit the new case, and the information is cached locally. When the application reconnects to the server, the new case is sent to the server.

For more information, see entity.

**Filters**
Filters are used to retrieve a set of entities based on specific criteria. The criteria is specified using standard JPQL. Entity filters result in service operations that retrieve the entity objects.

The following filter on the CustCase entity is used to retrieve cases for a specific customer:

```xml
<entity name="CustCase" persistent="true">
  <filter name="getByCustomerId" arguments="custName:string">
    <query><![CDATA[Select c From CustCase c Where c.customer.custId LIKE :custName]]></query>
  </filter>
</entity>
```

The CustCase service provides the getByCustomerId operation that takes a string value (a customer ID) as the argument.

For more information, see filter.
Property specifiers

Property specifiers determine which fields are returned when a filter is used to retrieve entity data. Property specifiers are typically used to avoid transmitting large amounts of unneeded data.

The Agent entity includes a filter called getAgentName. A property specifier causes the corresponding service calls to return Agent entities (objects) with values for only three properties.

For more information, see filter.

Associations

Associations denote a relationship between data types. Such a relationship can be one-to-one, one-to-many, or many-to-many. The custAuthData of the Customer entity represents a one-to-one relationship between Customer and User entities:

```xml
<entity name="Customer" persistent="true">
  <property name="custAuthData" type="User" required="true" cardinality="one-to-one">
    <annotation name="DMS">
      <item name="lazy">false</item>
    </annotation>
  </property>
</entity>
```

Each Case can have many Notes and the following declaration defines this relationship:

```xml
<entity name="CustCase" persistent="true">
  <property name="notes" type="Note\[]" mappedBy="custcase">
    <annotation name="DMS">
      <item name="lazy">false</item>
    </annotation>
  </property>
</entity>
```

For more information, see entity associations.

Lazy loading

Lazy loading enables you to avoid loading entire associated items when the parent object is loaded. The custAuthData property of the Customer entity includes the following annotation to configure lazy loading:

```xml
<entity name="Customer" persistent="true">
  <property name="custAuthData" type="User" required="true" cardinality="one-to-one">
    <annotation name="DMS">
      <item name="lazy">false</item>
    </annotation>
  </property>
</entity>
```

For more information, see property.

Transactions

The "Use Transactions" option for an entity enables CRUD operations to execute inside a server-managed transaction. The value of this option is true by default.

For more information, see entity.
**Load on demand**

The load on demand property for associated entity properties avoids retrieving entire entity associations from the server. Associated properties are instead retrieved only when they are requested from the client.

For example, the callcenter.fml model includes the following annotation for the custcase property of the CustCase entity:

```xml
<property name="custcase" type="CustCase">
  <annotation name="DMS" lazy="true" load-on-demand="true"/>
</property>
```

For more information, see property.