Flex Data Management Services Tutorial for Java Developers

By Christophe Coenraets
Last Update: January 11th 2007

Introduction

The Flex Data Management Services automate the process of synchronizing data between the client application and the middle-tier. The changes made to the data at the client-side are automatically sent to a service running in your application server. This service then passes the changes to your business layer or directly to your persistence layer, whatever your persistence solution is: DAOs with straight JDBC calls, Hibernate, EJBs, JPA, iBatis, or any other solution. In other words, your client application is freed of any data synchronization code. Code you no longer have to write includes:

1. Keeping track of all the items created, updated, and deleted by the end-user at the client-side.

2. Keeping track of the original value of the data as initially retrieved by the client. (The original value is often needed by the persistence layer to implement optimistic locking).

3. Making a series of RPC calls to send changes (creates, updates, deletes) to the middle-tier.

4. Handling the conflicts that may arise during this synchronization process.

Depending on the application you are building, using the Flex Data Management Services leads to ~80% less data manipulation code at the client side.

The goal of this tutorial is to get you started quickly with the Flex Data Management Services.

Prerequisites

This tutorial assumes that you meet the following prerequisites:

- Knowledge of server-side Java development and the structure of a J2EE web application
- Familiarity with SQL and JDBC
- Basic experience with Flex development
- Basic experience with Eclipse

Setting up your environment

Step 1: Install FlexBuilder 2.0.1

Refer to the following instructions to install FlexBuilder 2.0.1:

http://www.adobe.com/support/documentation/en/flex/2/install.html#installingfb2

Step 2: Install the FDS Test Drive Server

The FDS Test Drive Server is a minimal and ready-to-use version of Tomcat (currently version 5.5.20) in which the Flex Data Services (version 2.0.1) WAR file has already been deployed along with tutorials and sample applications. It allows you to get up and running in a matter of minutes.

To install the FDS Test Drive Server:

1. Make sure that you have the JDK 1.5 or higher installed, and that you have a JAVA_HOME or JRE_HOME environment variable pointing to your Java Development Kit installation.

   Note: The JDK 1.5 is a Tomcat 5.5 requirement, not an FDS requirement. The FDS requirement is JDK 1.4 or higher.

2. Download fds-tomcat.zip

3. Expand fds-tomcat.zip

   Note: The instructions in this document assume that you expand fds-tomcat.zip in C:\, which creates a directory called fds-tomcat at the root level. You can expand fds-tomcat.zip anywhere else. Just make sure you adjust the path in the tutorial instructions accordingly.

4. Start Tomcat

   a. Open a command prompt
   b. Navigate to c:\fds-tomcat\bin
   c. Execute the following command:

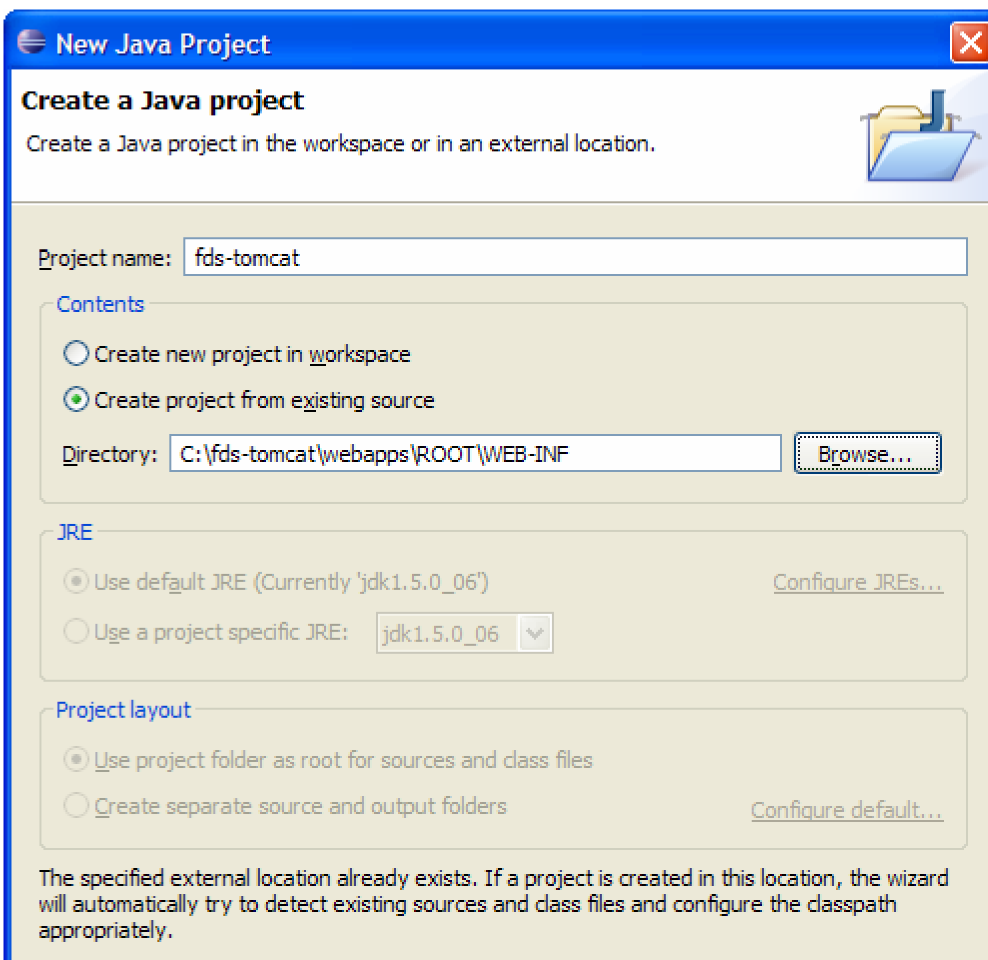
      catalina run

   To allow you to run the tutorial “out-of-the-box” without setting up a database, the Test Drive server includes an HSQLDB database. HSQLDB is a lightweight Java RDBMS that is particularly well suited to run samples. The HSQLDB database server is automatically started as part of the Tomcat startup process.

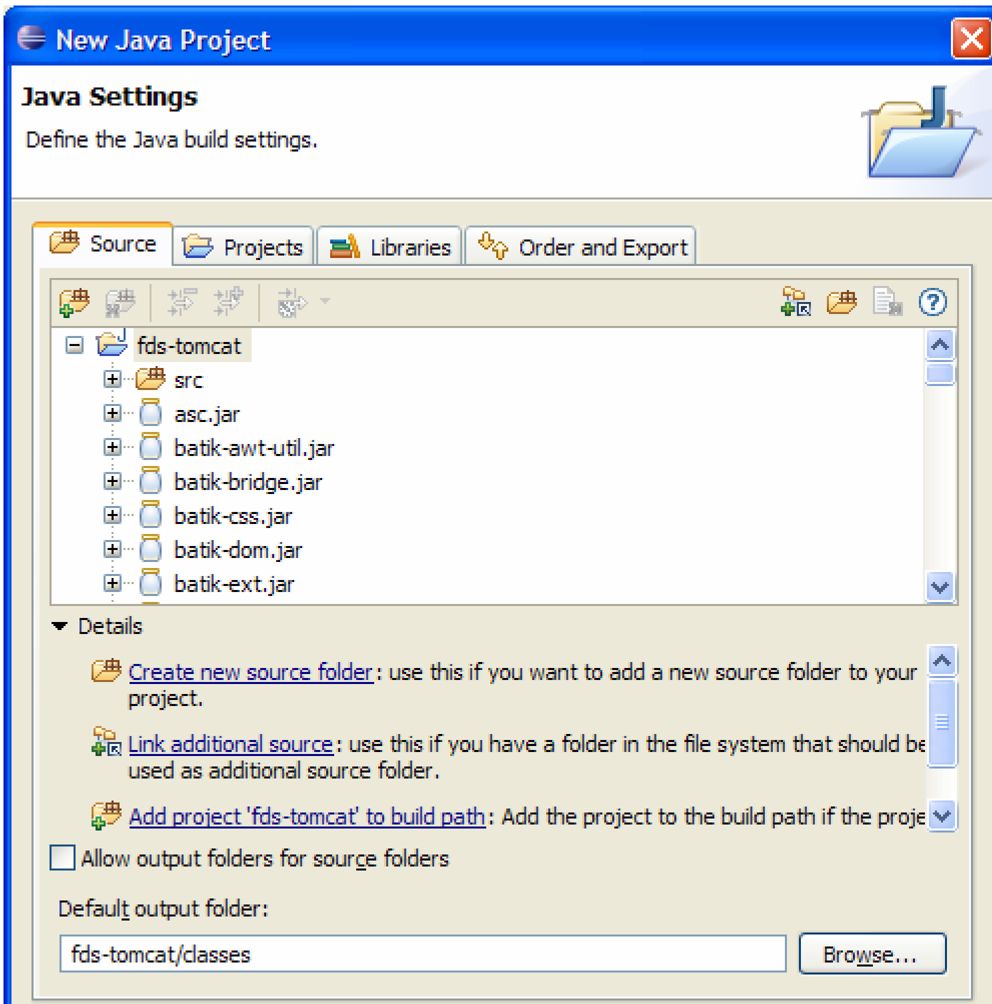
Step 3: Create a Java project

There are several ways you can set up Eclipse to work on the Java classes of a Web application. You can use a simple Java project, work with the Web Tools Platform (WTP), or other plugins like MyEclipse. In this tutorial, to avoid dependencies on a specific plugin, we use a simple Java project.

1. In the Eclipse menu, select File > New > Project
2. Select "Java Project" in the project type tree and click Next
3. On the "Create a Java Project" page of the wizard:
    - Specify "fds-tomcat" as the project name
    - Check "Create project from existing source", and point to the WEB-INF directory of the web application
    - Click Next



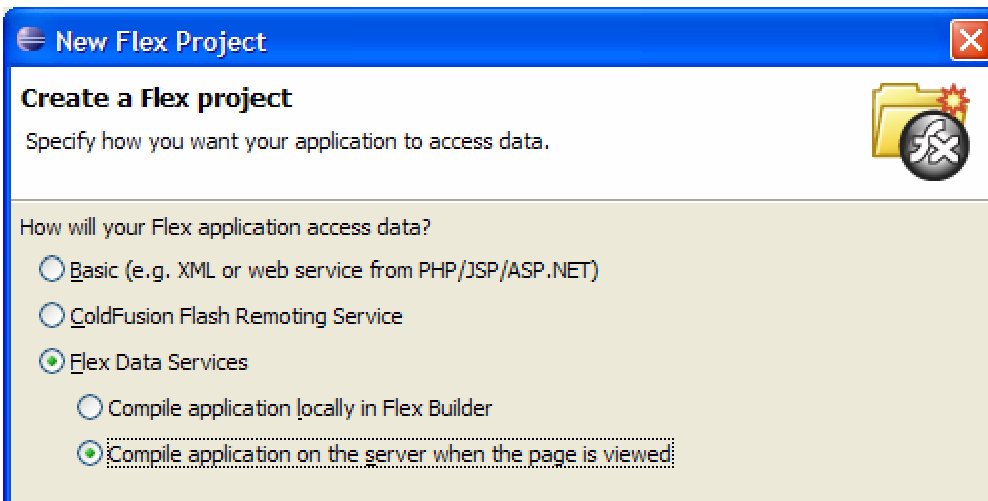
4. On the “Java Settings” page, specify **fdms-tomcat/classes** as the “Default output folder”, and click Finish.



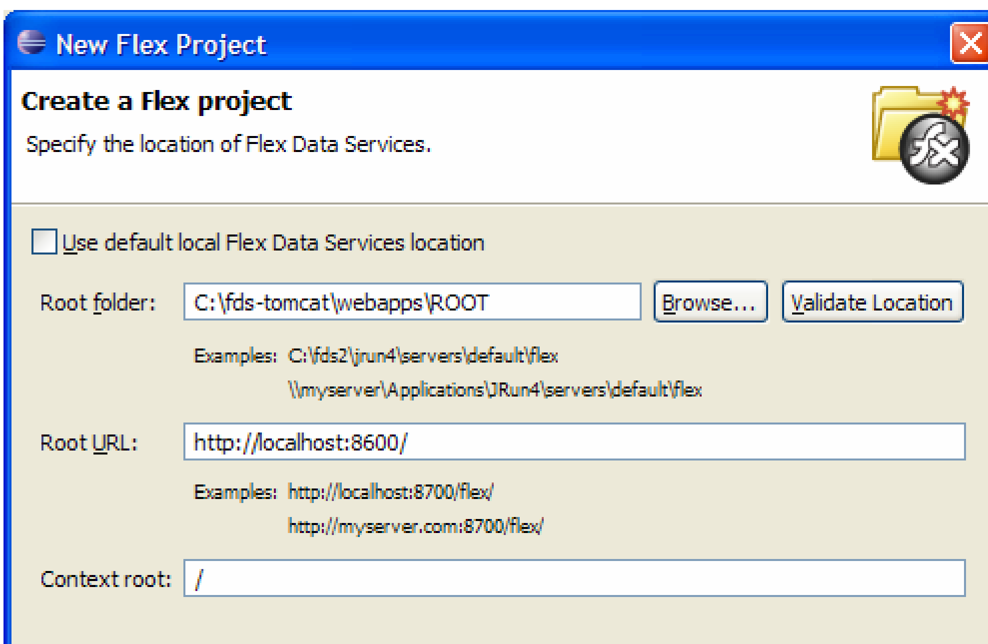
This project configuration allows you to store the source code for your Java classes in WEB-INF\src directory. These classes will automatically be compiled in WEB-INF\classes.

Step 4: Create a Flex project

1. In the Eclipse menu, select File > New > Project
2. Select "Flex Project" in the project type tree and click Next
3. Fill in the "Create a Flex Project" page of the wizard as follows and click Next



4. Fill in the next page of the wizard as follows and click Next



5. Fill in the next page of the wizard as follows and click Finish

Creating the Server Side

The changes made to the data at the client-side are automatically sent to a data service running in the application server. The data service then passes those changes to an object called the **assembler**. The role of the assembler is to pass the changes, in the appropriate format, to your existing business objects, or directly to your persistence objects. The assembler class is the only class that you have to write at the server-side (in addition to your existing business and persistence objects), and is typically a very simple class.

1. Make sure Eclipse is in the Java perspective

2. In the fds-tomcat project, right-click the flex.tutorial.fdms package under src, and select New > Class. Specify ProductAssembler as the class name and click Finish. Define the ProductAssembler class as follows:

```
package flex.tutorial.fdms;

import java.util.List;
import java.util.Collection;
import java.util.Map;
import flex.data.DataSyncException;
import flex.data.assemblers.AbstractAssembler;

public class ProductAssembler extends AbstractAssembler {

  public Collection fill(List fillArgs) {
      ProductDAO dao = new ProductDAO();
      return dao.getProducts();
  }

  public Object getItem(Map identity) {
      ProductDAO dao = new ProductDAO();
      return dao.getProduct(((Integer) identity.get("productId")).intValue());
  }

  public void createItem(Object item) {
      ProductDAO dao = new ProductDAO();
      dao.create((Product) item);
  }

  public void updateItem(Object newVersion, Object prevVersion, List changes) {
      ProductDAO dao = new ProductDAO();
      boolean success = dao.update((Product) newVersion, (Product) prevVersion,
  changes);
      if (!success) {
          int productId = ((Product) newVersion).getProductId();
          throw new DataSyncException(dao.getProduct(productId), changes);
      }
  }

  public void deleteItem(Object item) {
      ProductDAO dao = new ProductDAO();
      boolean success = dao.delete((Product) item);
      if (!success) {
          int productId = ((Product) item).getProductId();
          throw new DataSyncException(dao.getProduct(productId), null);
      }
  }

}
```

Code highlights:

- ProductAssembler extends AbstractAssembler and implements a series of callback methods that the data service invokes when it gets updates from the client application.
- You use the assembler to plug in to your existing business objects or directly to your persistence objects.
- In this simple implementation, we invoke the method corresponding to the type of change in our DAO class.
- We discuss some specific aspects of the FDMS API, such as the DataSyncException, later in this tutorial.

3. Define the product destination

   a. Open data-management-config.xml in the flex folder of the fds-tomcat project
   b. Add the following destination:

```
<destination id="fdms-tutorial-product">
    <adapter ref="java-dao" />
    <properties>
        <source>flex.tutorial.fdms.ProductAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="productId"/>
        </metadata>
    </properties>
</destination>
```

   This XML fragment essentially maps the logical name for the destination (fdms-tutorial-product) to an assembler class (flex.tutorial.fdms.ProductAssembler).

4. Make sure you saved ProductAssembler.java and data-management-config.xml, and restart the server

Creating the Client Application

Step 1: Creating a basic client

1. Make sure Eclipse is in the Flex Development perspective

2. Right click the fdms-tutorial project and select New > ActionScript class. Enter Product as the class name and click Finish. Define the Product class as follows:

```
package
{
  [Managed]
  [RemoteClass(alias="flex.tutorial.fdms.Product")]
  public class Product
  {
      public var productId:int;

      public var name:String;

      public var category:String;

      public var price:Number = 0;

      public var image:String;

      public var description:String;

      public var qtyInStock:int;
  }
}
```

Code highlights:

- Product.as is the ActionScript version of the Product.java value object.
- The [RemoteClass(alias=" flex.tutorial.fdms.Product")] annotation maps this class to its Java equivalent so that FDS knows how to serialize and deserialize Product objects.
- The [Managed] annotation indicates that the object should be automatically managed: the object will automatically trigger events when its properties are changed.

3. Open inventory.mxml in the fdms-tutorial project. Define inventory.mxml as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
  layout="horizontal" creationComplete="ds.fill(products)">

  <mx:DataService id="ds" destination="fdms-tutorial-product"/>
  <mx:ArrayCollection id="products"/>
  <Product/>

  <mx:Panel title="Product List" width="100%" height="100%">
      <mx:DataGrid id="dg" width="100%" height="100%"
          dataProvider="{products}" editable="true">
          <mx:columns>
              <mx:DataGridColumn dataField="productId" headerText="Id" editable="false"/>
              <mx:DataGridColumn dataField="name" headerText="Name"/>
              <mx:DataGridColumn dataField="category" headerText="Category"/>
              <mx:DataGridColumn dataField="price" headerText="Price"/>
              <mx:DataGridColumn dataField="qtyInStock" headerText="In Stock"/>
          </mx:columns>
      </mx:DataGrid>
      <mx:ControlBar>
          <mx:Button label="Delete" click="ds.deleteItem(dg.selectedItem)"/>
      </mx:ControlBar>
  </mx:Panel>

</mx:Application>
```

Code highlights:

- The DataService points to the "fdms-tutorial-product" destination defined earlier in data-management-config.xml.
- In the Application tag's creationComplete event, the DataService's fill() method populates the "products" array. When you invoke fill()at the client-side, the assembler's fill() method is invoked at the server-side. This is where you specify how the data should actually be retrieved. See the fill() method in ProductAssembler.java.
- The DataGrid is bound to the products array to display the list of products.

4. Test the application

   a. Open a browser and access http://localhost:8600/fdms-tutorial/inventory.mxml
   b. Change some data (for example, the qty in stock) for a few items
   c. Reload the application and notice that your changes have been persisted

5. Test the automatic client synchronization feature of the application

   a. Open a second browser and access the same URL
   b. Modify some data in one browser, and notice that the changes automatically appear in the second browser.

   Note: Automatically pushing the changes made by one client to the other clients working on the same data is the default behavior of FDS. We will see how this default behavior can be changed later in this tutorial.

Step 2: Controlling when changes are sent to the server

By default, FDMS sends a change message to the server immediately after you change a property of an object. This behavior is appropriate for some applications (for example, a collaborative form-filling application), but might not be desirable in other applications, because:

- It generates too much network traffic
- There may be dependencies between object attributes, and the server may not be able to persist partially filled objects

For these reasons, you often want to programmatically control when the changes are sent to the server.

1. To programmatically control when the changes are sent to the server:

   a. In inventory.mxml, add `autoCommit="false"` to the DataService declaration

   b. In the ControlBar, just after the Delete button, add an "Apply Changes" button defined as follows:

   ```
   <mx:Button label="Apply Changes" click="ds.commit()" enabled="{ds.commitRequired}"/>
   ```

2. Now that we have control over when the changes are sent to the server, we can also handle the creation of new products. In the ControlBar, after the "Apply Changes" button, add a "New" Button defined as follows:

   ```
   <mx:Button label="New" click="products.addItem(new Product())"/>
   ```

   Note: This logic for creating new products would have failed with autoCommit set to true (default) because the server-side component would have tried to insert an empty row and some columns of the product table are defined as not supporting null values.

3. Test the application

   a. Open a browser and access http://localhost:8600/fdms-tutorial/inventory.mxml
   b. Change some data (for example, the qty in stock) for a few items
   c. Reload the application and notice that your changes have not been persisted
   d. Change some data again and click the Apply Changes button
   e. Reload the application and notice that your changes have now been persisted

4. Test the automatic client synchronization feature of the application

   a. Open a second browser and access the same URL
   b. Modify some data in one session, and notice that, with autoCommit set to false, the changes are not pushed to the second browser session until you click the "Apply Changes" button.

Step 3: Controlling client synchronization

By default, FDMS pushes the changes made by one client to the other clients working on the same data. This default behavior is appropriate for many applications. Imagine, for example, looking at a store inventory, or at the seats available on an airplane: seeing the changes as they happen, and thus always looking at a "real time" version of the data is definitely a good thing. However in some other applications, this behavior may be considered confusing.

1. To change the default behavior, and prevent the DataService from receiving real time data changes: add `autoSyncEnabled="false"` to the DataService declaration.

2. Test the application

    a. Open two browsers
    b. Access http://localhost:8600/fdms-tutorial/inventory.mxml in both browsers
    c. Modify some data in one browser, and notice that the changes are not pushed to the second browser

Note: With autoSyncEnabled set to false, the chances that you are trying to update stale data increase. For example, you may be trying to update a product that another user has already deleted, or modified in an incompatible way. This is referred to as a "conflict". FDS provides a conflict resolution API that we explore later in this tutorial.

Step 4: Single Item vs Batch Updates

In many data management applications, the data entry process happens in a form where updates are performed one item at a time. In this section, we modify the inventory application to allow the user to enter data in a form (rather than directly in the DataGrid), and apply changes one product at a time.

1. Make sure Eclipse is in the Flex Development perspective

2. Right click the fdms-tutorial project and select New > MXML Component. Enter ProductForm as the Filename and click Finish. Define ProductForm.mxml as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Form xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*" width="100%"
height="100%">

  <Product id="product"
      name="{productName.text}"
      category="{category.text}"
      price="{Number(price.text)}"
      qtyInStock="{Number(qty.text)}"
      image="{image.text}"
      description="{description.text}"/>

  <mx:FormItem label="Name">
      <mx:TextInput id="productName" text="{product.name}" width="250"/>
  </mx:FormItem>

  <mx:FormItem label="Category">
      <mx:TextInput id="category" text="{product.category}" width="250"/>
  </mx:FormItem>

  <mx:FormItem label="Image">
      <mx:TextInput id="image" text="{product.image}" width="250"/>
  </mx:FormItem>

  <mx:FormItem label="Price">
      <mx:TextInput id="price" text="{product.price}" width="70"/>
  </mx:FormItem>

  <mx:FormItem label="Qty in Stock">
      <mx:TextInput id="qty" text="{product.qtyInStock}" width="70"/>
  </mx:FormItem>

  <mx:FormItem label="Description" width="100%">
      <mx:TextArea id="description" text="{product.description}" width="100%"
height="200"/>
  </mx:FormItem>

</mx:Form>
```

3. In inventory.mxml, add the following script block at the beginning of the file, just after the <mx:Application> tag

```
<mx:Script>
  <![CDATA[

      private function newItem():void
      {
          products.addItem(new Product());
          dg.selectedIndex = products.length - 1;
      }

      private function deleteItem():void
      {
          ds.deleteItem(dg.selectedItem)
          ds.commit();
      }

  ]]>
</mx:Script>
```

4. Remove the `<Product/>` tag

5. Remove `editable="true"` from the DataGrid declaration

6. Add `enabled="{!ds.commitRequired}"` to the DataGrid declaration and the “New” button declaration

7. Change the “New” button click handler: replace `products.addItem(new Product())` with a call to the `newItem()` function

8. Remove the “Delete” and “Apply Changes” buttons from the Product List Panel

9. Add the following Panel immediately after the “Product List” Panel.

```
<mx:Panel title="Details" width="100%" height="100%">
  <ProductForm id="form" product="{dg.selectedItem as Product}"/>
  <mx:ControlBar>
      <mx:Button label="Delete" click="deleteItem()"
          enabled="{form.product.productId!=0}"/>
      <mx:Button label="Apply Changes" click="ds.commit()"
          enabled="{ds.commitRequired}"/>
      <mx:Button label="Cancel Changes" click="ds.revertChanges()"
          enabled="{ds.commitRequired}"/>
  </mx:ControlBar>
</mx:Panel>
```

10. Test the application

    a. Open a browser and access http://localhost:8600/fdms-tutorial/inventory.mxml
    b. Change some data (for example, the qty in stock) for a few items
    c. Experiment with the “Apply Changes” and “Cancel Changes” buttons

Step 5: Conflict resolution

Conflicts can occur when different clients are trying to update the same data at the same time. The number of conflicts in an application depends on the locking strategy (concurrency level) implemented in the application.

For example, open ProductDAO.java and examine the difference between the update() and the updateStrict() methods:

- In the update() method, the WHERE clause is built using the primary key only. As a result, your update will succeed even if the row has been changed by someone else since you read it.

- In the updateStrict() method, the WHERE clause is built using the primary key and by verifying that each column still has the same value it had when you read the row. Your update will fail if the row has been changed by someone else since you read it.

  Note: Another way of implementing the same concurrency level without adding all the columns to the WHERE clause is to add a "version" column to the table. Every time the product is updated, the product's version is incremented. Using this approach, you can make sure a row has not been updated since you read it by building the WHERE clause using the primary key and the version column only.

The level of concurrency you choose depends on your application. FDMS doesn't force you to implement locking in any particular way, and doesn't even attempt to identify when a conflict has occurred. You are free to define what represents a conflict in the context of your application. You tell FDMS when a conflict has occurred by throwing a DataSyncException. For example, open ProductAssembler.java and examine the updateItem() method. Notice that the method throws a DataSyncException when the update fails in the DAO (no product matched the WHERE clause).

At the client-side, FDMS provides a sophisticated conflict resolution API. The DataService's "conflict" event is triggered when a conflict occurs. The conflict resolution API provides several options to handle the conflict as appropriate in the event handler.

In this section, we will use the updateStrict() method of ProductDAO because it makes it easier to generate conflicts and thus experiment with conflict resolution.

1. Open ProductAssembler.java. In the updateItem() method, replace dao.update(…) with dao.updateStrict(…). Recompile ProductAssembler.java and restart your application server.

2. In inventory.mxml, add the following import statements at the beginning of the <mx:Script> block, just after <![CDATA[

```
import mx.rpc.events.FaultEvent;
import mx.data.events.DataConflictEvent;
import mx.data.Conflicts;
import mx.data.Conflict;
import mx.controls.Alert;
```

3. In the <mx:Script> block, add a faultHandler() function defined as follows:

```
private function faultHandler(event:FaultEvent):void
{
  Alert.show(""+event);
}
```

4. In the <mx:Script> block, add a conflictHandler() function defined as follows:

```
private function conflictHandler(event:DataConflictEvent):void
{
  Alert.show("This product has been changed by someone else. The current state of the
product has been reloaded.", "Conflict");
  var conflicts:Conflicts = ds.conflicts;
  var conflict:Conflict;
  for (var i:int=0; i<conflicts.length; i++)
  {
      conflict = conflicts.getItemAt(i) as Conflict;
      if (!conflict.resolved)
      {
          conflict.acceptServer();
      }
  }
}
```

   Note: acceptServer() represents one approach to resolve conflicts. Refer to the documentation to explore the other options.

5. Add the following event handlers to the DataService declaration

```
fault="faultHandler(event)" conflict="conflictHandler(event)"
```

6. Test the application

   a. Open two browsers
   b. Access http://localhost:8600/fdms-tutorial/inventory.mxml in both browsers
   c. Modify the price of a product in one browser, and click “Apply Changes”
   d. Modify the qty in stock of the same product in the other browser and click “Apply Changes”

      Based on how we wrote the persistence logic at the server side, this is considered a conflict. The client application uses the conflict resolution API to revert to the current server value of the object.

Step 6: Supporting multiple fill queries

In the current version of the application the Product List always displays all the products. In real life applications, you often need to allow the user to specify one or more search criteria to bring back a subset of items. The fill() method allows you to pass a list of arguments that you can use as query parameters at the server-side. For example, if you wanted to support the ability to search products by name, you could modify the implementation of the fill() method in ProductAssembler.java as follows:

```
public Collection fill(List fillArgs) {
      ProductDAO dao = new ProductDAO();
      if (fillArgs == null || fillArgs.size() == 0) {
          return dao.getProducts();
      } else if (fillArgs.size() == 1) {
          return dao.getProductsByName((String) fillArgs.get(1));
      } else {
          throw new RuntimeException("Incorrect number of parameters");
      }
}
```

This implementation however isn't very flexible. Imagine, for example, that you have to provide the ability to search by name and by category. If provided with a single String argument, it would be impossible for the assembler's fill method to determine if the user wanted to search by name or by category (since both searches would take one argument of type String). A good solution to this problem is to work with named queries, using the first argument passed to the fill method as the query name.

1. In ProductAssembler.java, change the implementation of the fill method as follows:

```
public Collection fill(List fillArgs) {
  ProductDAO dao = new ProductDAO();
  if (fillArgs == null || fillArgs.size() == 0) {
      return dao.getProducts();
  }
  String queryName = ((String) fillArgs.get(0));
  if (queryName.equals("by_name")) {
      return dao.getProductsByName((String) fillArgs.get(1));
  } else {
      throw new RuntimeException("Unknown query");
  }
}
```

2. Recompile ProductAssembler.java and restart your application server

3. In inventory.mxml, add a TextInput inside the Product List Panel's ControlBar (after the "New" button) as follows:

```
<mx:TextInput id="productName" width="100%"/>
```

4. Add a Search button defined as follows immediately after the TextInput

```
<mx:Button label="Search" click="ds.fill(products, 'by_name', productName.text)"/>
```

5. Test the application. For example, type 3100 in the TextInput and click the Search button

Step 7: Single Objects vs Collections

In the current version of the application, you are always working on collection objects, even when you update a single product in the form. As a result, if the list goes away (for example because you execute a new search), the object you were working on in the form goes away too.

FDMS allows you to work with collections of objects or single items. In this section, we change the inventory application to use the single item approach when manipulating a product in the form.

1. In inventory.mxml, add the following import statement:

```
import mx.data.ItemReference;
```

2. Define an instance variable in the <mx:Script> block as follows:

```
[Bindable]
private var itemReference:ItemReference;
```

3. Define a productChange() function as follows:

```
private function productChange(product:Product):void
{
  itemReference = ds.getItem({productId: product.productId}, product);
}
```

4. Add a change event to the DataGrid declaration as follows:

```
change="productChange(dg.selectedItem as Product)"
```

5. Change the data binding of ProductForm as follows

```
<ProductForm id="form" product="{itemReference.result as Product}"/>
```

6. Test the application

    a. Open a browser and access http://localhost:8600/fdms-tutorial/inventory.mxml
    b. Select a Product and change some data (for example, the qty in stock)
    c. Execute a search that does not include the currently selected product in the form. Notice that the selected product doesn't disappear as the new list is retrieved. This is because the form is bound to a separate object that is not part of the collection.

Next Steps

This tutorial provided an overview of some of the key FDMS features. However, it didn't cover all the FDMS features. For example FDMS supports relationships between objects. In other words, you can persist entire object graphs. FDMS also supports generic assemblers for some persistence approaches. For example, using the Hibernate assembler (available out of the box), you can build an end-to-end persistence solution without writing any server-side code. Refer to the FDMS documentation for more details on these features.