Flex Quick Start Tutorial

Salesforce, Summer '18

@salesforcedocs

Last updated: June 20, 2018

CONTENTS

CREATING YOUR FIRST FLEX APPLICATION

Flex is a framework for creating Flex-based desktop and Web applications that leverage Salesforce logic, data, and security. The framework provides:

- Seamless handling of connected and disconnected states for desktop apps, including the detection of Internet connectivity
- Adobe's Data Management Services (DMS), which creates and manages local databases for desktop applications that operate online and offline
- Data synchronization between Salesforce and desktop applications
- Generated ActionScript classes that mirror your Salesforce enterprise WSDL and provide access to Salesforce objects
- MXML components to simplify the implementation of user interfaces that render Salesforce data, Flex status bars, and popup notifications called *toasts*
- An engine and user interface for resolving data conflicts and errors in desktop applications

The Flex framework installs as a standalone instance of Adobe® Flash® Builder for Lightning Platform—an enhanced version of Adobe's Eclipse-based integrated development environment (IDE) for developing Flex applications. Adobe Flash Builder for Lightning Platform includes classes that facilitate the development of Flex-based applications for Salesforce. These applications create, update, and delete Salesforce data via asynchronous requests to the Lightning Platform API. Mobile and tablet applications created using Flash Builder version 4.5 or newer aren't supported.

Flex desktop applications run on your workstation or laptop using Adobe® Integrated Runtime (AIR®), and operate both online and offline. The ability to operate offline makes them ideal for users who don't always have an Internet connection but still need to access Salesforce data. For example, traveling sales teams can use Flex desktop applications to update contacts, opportunities, and accounts while visiting customer sites.

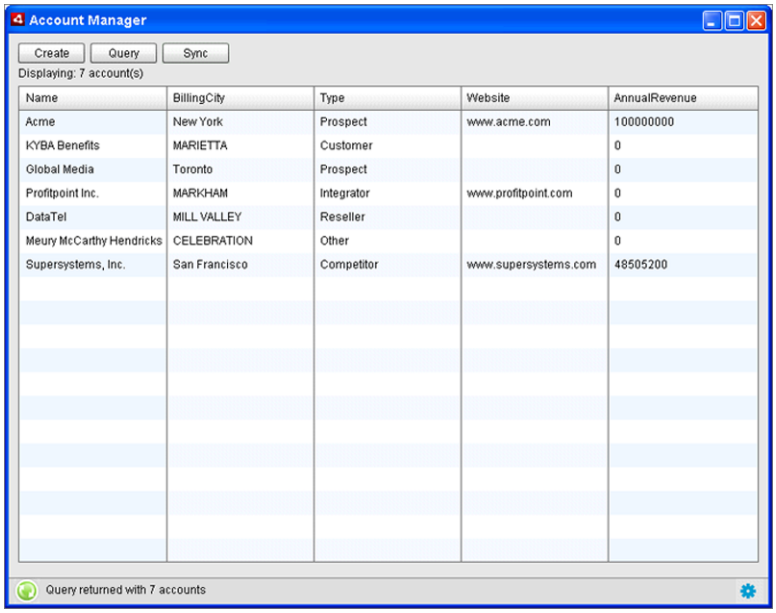
Flex Web applications run in browsers with Adobe® Flash® Player, and can operate as standalone Web applications or as embedded user interface components on Salesforce pages. For example, Flex Web applications can render animated charts and graphs based on Salesforce data.

This tutorial demonstrates how to use Flex to build the Account Manager application—a basic desktop application that provides offline access to account data. After you build the application, sales reps can use it to update Salesforce account records while traveling in areas with inconsistent Wi-Fi.

The top section of the finished Account Manager application contains:

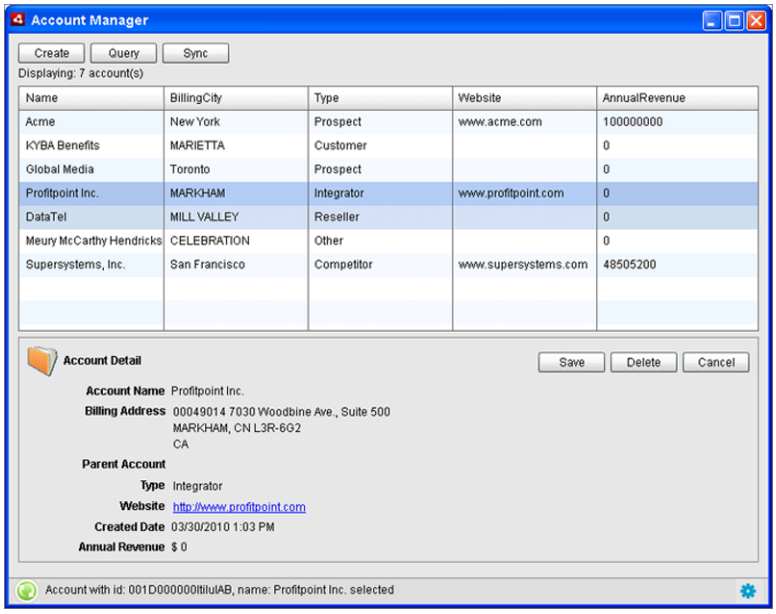
- Buttons to create records, query data, and synchronize with Salesforce
- A count of the number of account records retrieved from Salesforce
- A list of account records with information such as the account's billing city, the type of account, and more

The Account Manager User Interface



Clicking **Create** presents a user interface for creating a new account, with the fields appearing in full edit mode. Clicking **Query** presents a full list of account records. Selecting an account from that list displays its details, which you can click to edit.

Editing Records in the Account Manager Application



Clicking **Save** lets you commit changes to Lightning Platform, while clicking **Delete** removes the record.

The application logic for the Account Manager application consists of:

- Basic ActionScript
- MXML, including two Adobe component sets, Spark and MX
- Flex components

Flex components simplify the process of accessing Lightning Platform logic and data, and allow developers to easily add commonly required functionality, such as:

- A conflict resolution interface that lets users easily resolve data conflicts that occur when values in the Flex application violate validation rules or conflict with changes made by other users in Salesforce.
- A status bar that notifies users whether the Flex application has an Internet connection, and indicates the number of unresolved data conflicts and errors.

This tutorial is for developers who are familiar with Adobe® Flash® Builder and Salesforce. To follow the steps in this tutorial, you must have:

- Administrator credentials in a Salesforce account
- Sample account records in Salesforce for testing

The tutorial provides all the instructions and code you need to get your first Flex application up and running.

STEP 1: DESIGNATE YOURSELF AN OFFLINE USER

Only Salesforce users with the `Offline User` permission can develop or use Flex apps.

1. In Salesforce, from Setup, enter *`Users`* in the `Quick Find` box, then select **Users**.
2. Click **Edit** next to your name.
3. Select the `Offline User` checkbox.
4. Click **Save**.

STEP 2: CREATE AN OFFLINE BRIEFCASE CONFIGURATION

Your Flex desktop application can access only the Lightning Platform data specified in your assigned offline briefcase configuration. Offline briefcase configurations are sets of parameters that make Lightning Platform records available to client applications. Salesforce administrators can create multiple briefcase configurations and assign each to different users and profiles to simultaneously suit the needs of various types of offline users. For example, one configuration might include leads and opportunities for sales representatives, while another configuration includes accounts and related opportunities for account executives.

The Account Manager application requires access to account data, so create an offline briefcase configuration that includes the Account object, and assign the configuration to yourself.

1. In Salesforce, from Setup, enter *Offline Briefcase Configurations* in the Quick Find box, then select **Offline Briefcase Configurations**.
2. Click **New Offline Briefcase Configuration**.
3. Enter a name for your offline briefcase configuration, such as *Account Manager App Data*.
4. Select the Active checkbox.
5. Select your name in the Available Members list and click **Add** to move it to the Assigned Members list.
6. Click **Save**. The Offline Briefcase Configuration detail page appears.
7. Click **Edit** in the Data Sets related list.
8. Click **Add...**.
9. Select *Account* and click **OK**.
10. In the Set Max Record Limit section, select No Limit.
11. Click **Done**.

STEP 3: GENERATE YOUR ENTERPRISE WSDL

Web services are interfaces that enable the integration of applications over the Internet. The Web Services Description Language (WSDL) is an XML-based language for describing Web services.

Lightning Platform has its own Web services that support incoming and outgoing calls. To use these Web services, you must generate your *enterprise WSDL*—a Salesforcegenerated XML file that describes your Lightning Platform data model.

Flex lets you import your enterprise WSDL file into development projects. When you import your WSDL, Flex creates ActionScript classes for every object in your enterprise WSDL, allowing you to reference the objects in your ActionScript code.

You will import your enterprise WSDL into your development project later to create a class for the Account object. For now, generate and save a local copy of your enterprise WSDL.

1. In Salesforce, from Setup, enter *API* in the `Quick Find` box, then select **API**.
2. Click **Generate Enterprise WSDL**.
3. In your browser, click **File** > **Save Page As...**.
4. Name the file `enterprise.wsdl` and click **Save**.

STEP 4: INSTALL AND LAUNCH FLEX

Flex installs as a standalone version of Adobe® Flash® Builder. The standalone version does not affect other instances of Adobe Flash Builder on your machine.

1. Download the Flex zip file from http://developer.salesforce.com/flashbuilder.
2. Extract the contents of the zip file.
3. Double-click the installer.
4. When the installation is complete, double-click the Flex shortcut on your desktop.

STEP 5: CREATE A FLEX PROJECT AND IMPORT YOUR WSDL

Flex projects are Flex projects with additional ActionScript classes that add the following functionality to Flex desktop applications:

- Data storage and synchronization
- Online and offline detection
- Data conflict and error resolution
- Status bars
- Toaster alerts

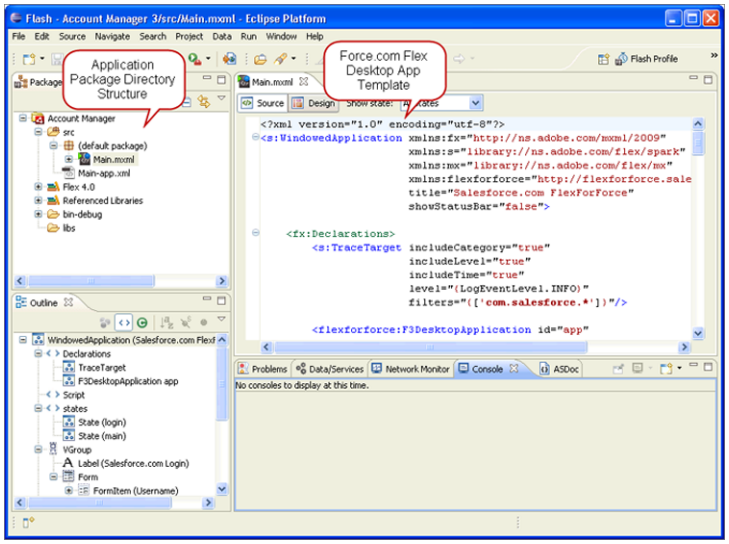
Flex desktop applications projects also have ActionScript classes and MXML components that let you add user interface components with the Salesforce look and feel to your both desktop and Web applications.

To create a Flex project:

1. In Flash Builder, select **File** > **New** > **Project**. The New Project wizard opens.
2. In the Flash Builder folder, select Flex and click **Next**.
3. Enter `Account Manager` as the project name, select `Desktop (runs in Adobe AIR)`, and click **Next**.
4. Accept the default settings for the output folder and click **Next**.
5. Enter `com.salesforce.AccountManager` in the `Application ID` field, and click **Finish**.

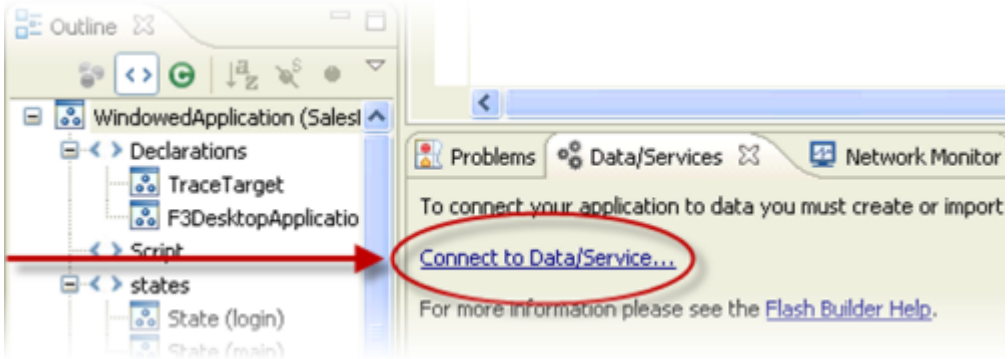
   Flash Builder displays your application package directory structure and opens the `Main.mxml` file, which contains a Flex desktop application template.

Application Files and Template



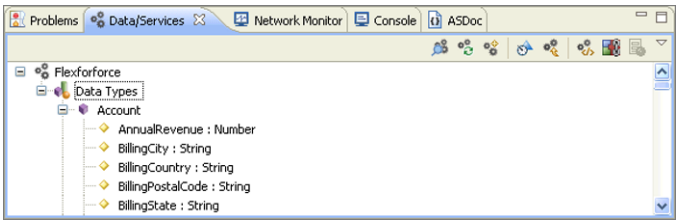
6. Select the Data/Services tab and click the **Connect to Data/Service** link.

Connect to Data/Service Link



7. Select Salesforce, then click **Next**.
8. Select your enterprise WSDL, then click **Next**. Flex creates a Fiber model of your enterprise WSDL.

Fiber Model of Your Enterprise WSDL



You are now ready to begin coding the Account Manager application.

STEP 6: SET THE WINDOWEDAPPLICATION COMPONENT ATTRIBUTES

The first component in the Flex desktop application template is a standard Flex `WindowedApplication` component. The component defines the contents of an operating system window. Modify the attributes of the component as follows:

1. Open the `Main.mxml` file in **src** > **(default package)** if it is not already open.
2. Set the `title` attribute to *`"Account Manager"`*. This controls the name that appears in the title bar of the application.
3. Set `backgroundColor` to *`"#e7e7e7"`*, `width` to *`"800"`*, `height` to *`"600"`*, and `frameRate` to *`"31"`*. These attributes control the appearance of the window containing the application's UI elements.
4. Set the `xmlns:local` attribute to *`"*"`*. You will use this later to reference another MXML file.
5. Set the `currentState` attribute to *`"login"`*. This configures the application to display the login interface when the application starts.
6. Your final code should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    showStatusBar="false"
    xmlns:local="*"
    backgroundColor="#e7e7e7"
    width="800"
    height="600"
    frameRate="31"
    currentState="login"
    xmlns:flexforforce="http://flexforforce.salesforce.com"
    title="Account Manager">
```

Note: The Account Manager application will have a status bar, but leave the `showStatusBar` attribute set to *`false`*. We'll create the status bar later using a Flex component.

7. Save your work.

STEP 7: CREATE THE ACCOUNT MANAGER LOGIN SCREEN

Create the logic for the Account Manager login screen in the `Main.mxml` file. The login screen launches the main Account Manager user interface if a user logs in with valid credentials, and displays an error message if the credentials are invalid.

The `Main.mxml` default code contains several components that you'll either use as-is or modify:

- `Declarations`—Defines user interface elements within a Flex application.
- `F3DesktopApplication`—Connects the Flex desktop application to Lightning Platform, provides the Lightning Platform login and authentication functionality, and handles the initial synchronization of data between Lightning Platform and the Flex application. Every Flex desktop application is contained within an `F3DesktopApplication` component. The `requiredTypes` attribute on this component determine which Salesforce objects the desktop application can access.
- `Script`—Contains the embedded ActionScript that defines the actions on the user interface elements defined in the `Declarations` component. You will change the default ActionScript in this component.
- `States`—Enables the application to determine which user interface to present.
- The main UI for your application, which is in a Spark `Group` component. The component contains a `layout` component used to arrange your UI elements, a `VGroup` component that creates a vertical grouping of the UI elements, and a Flex component that renders a status bar at the bottom of the application.

Modify the rest of the `Main.mxml` default code as follows:

1. Remove the `VGroup` and `Group` components from the default code, and replace them with the following Spark `Panel` component. The `Panel` component provides a screen entitled "Salesforce.com Login." The screen has two input fields, `Username` and `Password`, and a **Login** button. The component also provides a progress bar that appears while the application is processing the login attempt.

```
<s:Panel
    title="Salesforce.com Login"
    includeIn="login"
    horizontalCenter="0"
    verticalCenter="0">

    <s:layout>
        <s:VerticalLayout paddingLeft="10"/>
    </s:layout>

    <mx:Form>
        <mx:FormItem label="Username">
            <s:TextInput id="username" text=""/>
        </mx:FormItem>
        <mx:FormItem label="Password" direction="horizontal">
            <s:TextInput
                id="password"
                text=""
                displayAsPassword="true"/>
            <s:Button
                label="Login"
                enabled="{!app.loginPending}"
                click="loginClickHandler(event)"/>
        </mx:FormItem>
```

```
    </mx:Form>

    <mx:ProgressBar
        visible="{app.loginPending}"
        indeterminate="true"
        label="{statusMessage}"/>

</s:Panel>
```

2. Replace the default ActionScript code in the `Script` component with the following code, which is similar to the default code but has logic to handle status changes.

```
import com.salesforce.events.LoginFaultEvent;
import com.salesforce.events.LoginResultEvent;
import com.salesforce.events.SessionExpiredEvent;
import com.salesforce.events.StatusChangedEvent;

import mx.controls.Alert;

[Bindable]
private var statusMessage : String = "";

private var _username : String;
private var _password : String;

protected function loginClickHandler( event : MouseEvent ) : void {
    _username = username.text;
    _password = password.text;
    app.loginByCredentials( _username, _password );
}

protected function statusChangedHandler( event : StatusChangedEvent ) : void {
    trace( "status change: " + event.message.description );
    statusMessage = event.message.description;
}

protected function loginFailedHandler( event : LoginFaultEvent ) : void {
    event.preventDefault();
    Alert.show( event.faultString, "Login failed." );
    currentState = "login";
}

protected function loginCompleteHandler( event : LoginResultEvent ) : void {
    currentState = "main";
}

protected function sessionExpiredHandler( event : SessionExpiredEvent ) : void {
    // Alert will be shown if default is not prevented.
    event.preventDefault();
    app.loginByCredentials( _username, _password );
}
```

3. Add the `AccountsView` component to the end of the `Main.mxml` file, just before the closing `WindowedApplication` tag. Your local namespace includes this component. The component does the following:

- Indicates that the `AccountsView` file is part of the main UI
- Passes the Integration component's `F3DesktopApplication` instance to the `AccountsView` component by setting its `app` attribute
- Sets the relative dimensions of the AccountsView window

```
<local:AccountsView
    includeIn="main"
    app="{app}"
    width="100%"
    height="100%"/>
```

4. Save the file.

The final code for the `Main.mxml` file should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    showStatusBar="false"
    xmlns:local="*"
    backgroundColor="#e7e7e7"
    width="800"
    height="600"
    frameRate="31"
    currentState="login"
    xmlns:flexforforce="http://flexforforce.salesforce.com"
    title="Account Manager">

    <s:layout>
        <s:BasicLayout/>
    </s:layout>

    <fx:Declarations>

        <flexforforce:F3DesktopApplication
            id="app"
            statusChanged="statusChangedHandler(event)"
            loginComplete="loginCompleteHandler(event)"
            loginFailed="loginFailedHandler(event)"
            sessionExpired="sessionExpiredHandler(event)"
            requiredTypes="Account,Contact"/>

    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import com.salesforce.events.LoginFaultEvent;
            import com.salesforce.events.LoginResultEvent;
            import com.salesforce.events.SessionExpiredEvent;
            import com.salesforce.events.StatusChangedEvent;

            import mx.controls.Alert;
```

```
            [Bindable]
            private var statusMessage : String = "";

            private var _username : String;
            private var _password : String;

            protected function loginClickHandler( event : MouseEvent ) : void {
                _username = username.text;
                _password = password.text;
                app.loginByCredentials( _username, _password );
            }

            protected function statusChangedHandler( event : StatusChangedEvent ) : void
{
                trace( "status change: " + event.message.description );
                statusMessage = event.message.description;
            }

            protected function loginFailedHandler( event : LoginFaultEvent ) : void {
                event.preventDefault();
                Alert.show( event.faultString, "Login failed." );
                currentState = "login";
            }

            protected function loginCompleteHandler( event : LoginResultEvent ) : void {
                currentState = "main";
            }

           protected function sessionExpiredHandler( event : SessionExpiredEvent ) : void
 {
                // Alert will be shown if default is not prevented.
                event.preventDefault();
                app.loginByCredentials( _username, _password );
            }
        ]]>
    </fx:Script>

    <s:states>
        <s:State name="login"/>
        <s:State name="main"/>
    </s:states>

    <s:Panel
        title="Salesforce.com Login"
        includeIn="login"
        horizontalCenter="0"
        verticalCenter="0">

        <s:layout>
            <s:VerticalLayout paddingLeft="10"/>
        </s:layout>

        <mx:Form>
            <mx:FormItem label="Username">
```

```
                <s:TextInput id="username" text=""/>
            </mx:FormItem>
            <mx:FormItem label="Password" direction="horizontal">
                <s:TextInput
                    id="password"
                    text=""
                    displayAsPassword="true"/>
                <s:Button
                    label="Login"
                    enabled="{!app.loginPending}"
                    click="loginClickHandler(event)"/>
            </mx:FormItem>
        </mx:Form>

        <mx:ProgressBar
            visible="{app.loginPending}"
            indeterminate="true"
            label="{statusMessage}"/>

    </s:Panel>


    <local:AccountsView
        includeIn="main"
        app="{app}"
        width="100%"
        height="100%"/>

</s:WindowedApplication>
```

STEP 8: CREATE THE ACCOUNT MANAGER USER INTERFACE AND APPLICATION LOGIC

Create the Account Manager user interface in a new file called `AccountsView.mxml`.

1. Click **File** > **New** > **MXML Component** and enter *`AccountsView`* in the `Name` field. The file opens in Adobe Flash Builder. You'll modify its default code to implement the Account Manager application logic.
2. Remove the `width` and `height` attributes in the `Group` component and `Declarations` component.
3. Add a `flexforforce` namespace attribute.

```
<s:Group
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:flexforforce="http://flexforforce.salesforce.com">
```

4. In the `Group` component, add the following components, which specify the user interface layout and components, and contain the ActionScript that manages the operations in the main application window.

```
<s:layout>
</s:layout>

<fx:Script>
    <![CDATA[
    ]]>
</fx:Script>

<s:states>
</s:states>

<s:VGroup
    width="100%"
    height="100%"
    paddingBottom="10"
    paddingLeft="10"
    paddingRight="10"
    paddingTop="10">
</s:VGroup>

<flexforforce:StatusBar/>
```

5. In the `layout` component, add a `VerticalLayout` component to set up the vertical orientation of the user interface elements in the main window.

```
<s:layout>
    <s:VerticalLayout/>
</s:layout>
```

6. In the `States` component, add `State` components with `name` attributes for the following states:
   - Default—Displays the main interface including a list of records

- Create—Displays the user interface for creating a new account
- Edit—Displays the user interface for editing the details of an existing account

The code should look like this:

```
<s:states>
    <s:State name="default"/>
    <s:State name="create"/>
    <s:State name="edit"/>
</s:states>
```

7. In the `VGroup` component, add the following components:
   - `HGroup`—A horizontal group that will contain the buttons for creating new records, querying the local database, and synchronizing with Salesforce.
   - `Label`—Displays the total number of records.
   - `DataGrid`—References the list of accounts through an ActionScript variable and presents the records in the user interface.
   - `Fieldcontainer`—Displays the user interfaces to either edit an existing account or create a new one.

   The code should look like this:

```
<s:HGroup width="100%" verticalAlign="middle">
</s:HGroup>

<s:Label text="Displaying: {_gridDataProvider.length} account(s)"/>

<mx:DataGrid
    id="_dataGrid"
    width="100%"
    height="100%"
    resizeEffect="Resize"
    dataProvider="{_gridDataProvider}"
    itemClick="onDataGridItemClick()">
</mx:DataGrid>


<!-- edit mode: show UI components in inline edit mode -->
<flexforforce:FieldContainer
    id="_editFieldContainer"
    width="100%"
    includeIn="edit">

</flexforforce:FieldContainer>

<!-- create mode: show UI components in full edit mode -->
<flexforforce:FieldContainer
    id="_createFieldContainer"
    startState="Create"
    width="100%"
    includeIn="create">
</flexforforce:FieldContainer>
```

8. In the `HGroup` component, add the top bar containing the **Create**, **Query**, and **Sync** buttons. When a user clicks the buttons, the application logic calls ActionScript event handlers specified in the `click` attributes. You will create these handlers later.

```
<s:HGroup width="100%" verticalAlign="middle">
    <s:Button label="Create" click="onCreateClick()"/>
    <s:Button label="Query" click="onQueryClick()"/>
    <s:Button label="Sync" click="onSyncClick()"/>
    <mx:Spacer width="100%"/>
</s:HGroup>
```

9. Add the following `columns` component in the `DataGrid` component. The `columns` component data fields for the account name, billing city, type, website, and annual revenue. The `DataGrid` component presents a full listing of the account records retrieved from the local database when a user clicks the **Sync** button. When a user clicks a record, the application calls the ActionScript method presenting the edit user interface.

```
<mx:DataGrid
    id="_dataGrid"
    width="100%"
    height="100%"
    resizeEffect="Resize"
    dataProvider="{_gridDataProvider}"
    itemClick="onDataGridItemClick()">
    <mx:columns>
        <mx:DataGridColumn dataField="Name"/>
        <mx:DataGridColumn dataField="BillingCity"/>
        <mx:DataGridColumn dataField="Type"/>
        <mx:DataGridColumn dataField="Website"/>
        <mx:DataGridColumn dataField="AnnualRevenue"/>
    </mx:columns>
</mx:DataGrid>
```

10. Create two Flex `FieldContainer` components, which group data and simplify the process of manipulating several fields in a single operation. The `FieldContainer` components you're creating here will also render the details of the selected record at the bottom of the application window. The first component edits an existing account, and the second creates a new account. Each has an `id` attribute that the ActionScript code uses as a variable to render the interface elements contained in the component. The `includeIn` attribute contained in each component indicates which UI the application presents.

    In addition to horizontal groups of buttons needed in each UI, you will also add one `LabelAndField` component for each field of the Account object you want to display, and reference each field by its internal Lightning Platform API name. A `LabelAndField` component is a Flex component that renders a Salesforce field value with its label. The fields you create with the `LabelAndField` component automatically function in Flex applications the same way they do on the Salesforce user interface. For example, date fields display a calendar when clicked. The fields also respect field dependencies, and automatically provide inline editing, hover details, error notification, and Info icon help text.

```
<!-- edit mode: show UI components in inline edit mode -->
<flexforforce:FieldContainer
    id="_editFieldContainer"
    width="100%"
    includeIn="edit">
    <s:HGroup
        width="100%"
        paddingLeft="5"
        paddingRight="5"
        paddingBottom="5"
```

```
        paddingTop="5"
        verticalAlign="middle">

        <mx:Image source="{StaticAssets.ACCOUNT_IMAGE_32}"/>
        <s:Label text="Account Detail" fontWeight="bold"/>
        <mx:Spacer width="100%"/>
        <s:Button label="Save" click="onEditSaveClick()"/>
        <s:Button label="Delete" click="onEditDeleteClick()"/>
        <s:Button label="Cancel" click="onEditCancel()"/>
    </s:HGroup>

    <flexforforce:LabelAndField field="Account.Name"/>
    <!-- This expands to include all of the sub-fields in a Billing Address -->
    <flexforforce:LabelAndField field="Account.BillingStreet[Group]"/>
    <flexforforce:LabelAndField field="Account.ParentId"/>
    <flexforforce:LabelAndField field="Account.Type"/>
    <flexforforce:LabelAndField field="Account.Website"/>
    <flexforforce:LabelAndField field="Account.CreatedDate"/>
    <flexforforce:LabelAndField field="Account.AnnualRevenue"/>

</flexforforce:FieldContainer>

<!-- create mode: show UI components in full edit mode -->
<flexforforce:FieldContainer
    id="_createFieldContainer"
    startState="Full Edit"
    width="100%"
    includeIn="create">

    <s:HGroup
        width="100%"
        paddingLeft="5"
        paddingRight="5"
        paddingBottom="5"
        paddingTop="5"
        verticalAlign="middle">

        <mx:Image source="{StaticAssets.ACCOUNT_IMAGE_32}"/>
        <s:Label text="New Account" fontWeight="bold"/>
        <mx:Spacer width="100%"/>
        <s:Button label="Save" click="onCreateSaveClick()"/>
        <s:Button label="Cancel" click="onCreateCancelClick()"/>
    </s:HGroup>

    <flexforforce:LabelAndField field="Account.Name"/>
    <flexforforce:LabelAndField field="Account.ParentId"/>
    <flexforforce:LabelAndField field="Account.Type"/>
    <flexforforce:LabelAndField field="Account.Website"/>
    <flexforforce:LabelAndField field="Account.AnnualRevenue"/>
</flexforforce:FieldContainer>
```

11. The Account Manager user interface is now in place, but you need to add the ActionScript that retrieves Salesforce data and saves the data in the Flex application. Begin by importing the classes you need for the ActionScript in the `<![CDATA[` section inside the `<fx:Script>` element. The classes are:

- `StaticAssets`—A Flex component used to retrieve assets such as the account image to be presented in the UIs.
- `ToasterEvent`—A Flex event class used to raise a Flex toaster alert.
- `F3DesktopApplication`—A Flex application container.
- `F3DesktopWrapper`—A Flex component that wraps the functionality of Adobe's Data Management Service (DMS), a key component of Adobe AIR platform that provides a model to manage client-server data synchronization.
- `F3Message`—A Flex component that provides standard contextual error and information messages, such as data conflicts.
- `Toaster`—A Flex component that displays a popup containing a message.
- `ArrayCollection`—An ActionScript wrapper containing an array representation of a collection. It will be used to create the variable containing the Account records.
- `IManaged`—An ActionScript interface that provides the contract for a managed object. It will be used to commit information to the database.
- `ILogger` and `Log`—An ActionScript interface and class used for logging error messages.
- `Responder`—An ActionScript class containing methods called by a remote service. It is used to create objects sent with requests to either the local database or the Salesforce database.
- `FaultEvent`—An ActionScript class used when handling errors. It is used to handle errors when the user attempts to save accounts containing invalid fields.
- `Account`—The ActionScript class representing the Account object generated by Flex.

The code looks like this:

```
import com.salesforce.assets.StaticAssets;
import com.salesforce.events.ToasterEvent;
import com.salesforce.flexforforce.F3DesktopApplication;
import com.salesforce.flexforforce.F3DesktopWrapper;
import com.salesforce.notifications.F3Message;
import com.salesforce.notifications.Toaster;

import mx.collections.ArrayCollection;
import mx.data.IManaged;
import mx.logging.ILogger;
import mx.logging.Log;
import mx.rpc.Responder;
import mx.rpc.events.FaultEvent;

import services.flexforforce.Account;
```

12. Create variables that your code can use to access the application container, database functionality, and error logging:

```
private static const LOG : ILogger = Log.getLogger( "AccountsView" );
protected var _app : F3DesktopApplication;
protected var _desktopWrapper : F3DesktopWrapper;
```

13. Create a variable that your code can use to display the account data retrieved in the query.

```
// After executing a query, this is populated with the resulting rows
[Bindable]
protected var _gridDataProvider : ArrayCollection = new ArrayCollection();
```

14. Define the following `set` function. The function passes the `F3DesktopApplication` object to the `AccountsView` component using `app="{app}"` attribute, and assigns the `F3DesktopApplication` object to the `_app` variable you declared earlier in the ActionScript code.

```
public function set app( app : F3DesktopApplication ) : void {
    _app = app;
    _desktopWrapper = app.wrapper;
}
```

15. Create the event handlers for the **Create**, **Query**, and **Sync** buttons. Use the following functions:

    - `onCreateClick`—Renders the UI where the user creates the new account.
    - `onSyncClick`—Synchronizes the local database with the Salesforce database and updates the status.
    - `onQueryClick`—Sends a query to the local database to retrieve all the Account records, and calls `onQueryResult`, a function we will write that displays the new list of records.

    The code looks like this:

```
protected function onCreateClick() : void {
    _app.setStatus( null );

    // switch to create mode and simply render a new account
    currentState = "create";
    _createFieldContainer.fieldCollection.render( new Account());
}

protected function onSyncClick() : void {
    _app.setStatus( null );

    _desktopWrapper.syncWithServer(
        null,
        new mx.rpc.Responder(
            function( result : F3Message ) : void {
                _app.setStatus( result );
            },
            function( result : F3Message ) : void {
                LOG.error( "syncWithServer failed: " + result.toString());
            }
        )
    );
}

protected function onQueryClick() : void {
    _app.setStatus( null );
    _desktopWrapper.query(
        "select * from Account",
        new mx.rpc.Responder(
            onQueryResult,
            function( result : F3Message ) : void {
                LOG.error( "query failed: " + result.toString());
            }
        )
    );
}
```

16. Create the event handlers for the **Save**, **Delete**, and **Cancel** buttons that display when the user edits an account record Use the following functions:

    - `onEditSaveClick`—Commits the updated object to the local database.
    - `onEditDeleteClick`—Deletes the record from the local database, updates the status, and displays a toaster alert.
    - `onEditCancelClick`—Cancels the current edit operation.

    The code should like this:

```
protected function onEditSaveClick() : void {
    _app.setStatus( null );

    // The existing account object has been updated in memory.
    // Now commit the updated object to the database.
    _editFieldContainer.fieldCollection.updateObject(
        new mx.rpc.Responder( commitToDB, saveFaultHandler )
    );
}

protected function onEditDeleteClick() : void {
    _app.setStatus( null );

    var account : Account = Account(
        _editFieldContainer.fieldCollection.managedObject
    );

    // clear and go back to empty mode
    _editFieldContainer.fieldCollection.clear();
    currentState = "default";

    _desktopWrapper.deleteItem( account )
    var status : F3Message = new F3Message(
                        F3Message.STATUS_INFO,
                        "Account deleted"
                    );

                    // Show the message in the status bar.
                    _app.setStatus( status );

                    // Show the toaster.
                    _app.showToaster(
                        new Toaster(
                            StaticAssets.REPORT_IMAGE_32,
                            "Account deleted",
                            status.description
                        )
                    );
                }

protected function onEditCancel() : void {
    _app.setStatus( null );

    // clear and go back to empty mode
    _editFieldContainer.fieldCollection.clear();
```

```
    currentState = "default";
}
```

17. Create the event handlers for the **Save** and **Cancel** buttons that display when the user is creating an account record. The `onCreateSaveClick` function commits the new object to the local database. The function `onCreateCancelClick` named cancels the current create operation. The code should like this:

```
protected function onCreateSaveClick() : void {
    _app.setStatus( null );

    // The new account object has been created in memory.
    // Now commit the new object to the database.
    _createFieldContainer.fieldCollection.updateObject(
        new mx.rpc.Responder( commitToDB, saveFaultHandler )
    );
}

protected function onCreateCancelClick() : void {
    _app.setStatus( null );

    // clear and go back to empty mode
    _createFieldContainer.fieldCollection.clear();
    currentState = "default";
}
```

18. Add the following ActionScript functions:

    - `saveFaultHandler`—Handles errors for invalid fields.
    - `commitToDB`—Commits the data in memory to the local database and clears the data from the `FieldContainer`.
    - `onDataGridItemClick`—Renders the UI and alerts for editing an account.
    - `onQueryResult`—Stores the results of the query and updates the status.
    - `onToasterSelected`—Used when displaying toaster alerts.

    The code should like this:

```
protected function saveFaultHandler( msg : F3Message ) : void {
    // If there are invalid fields, stay on the edit screen
    // so the user has a chance to fix the errors.
    // Otherwise, clear the record.
    if (
        msg != null &&
        msg.messageType != F3Message.ERROR_INVALID_FIELDS
    ) {
        currentState = "default";
    }

    if ( msg != null ) {
        var faultEvent : FaultEvent = msg.context as FaultEvent;
        if ( faultEvent ) {
            LOG.error( faultEvent.fault.faultString );
            LOG.error( faultEvent.fault.faultDetail );
            LOG.error( faultEvent.fault.faultCode );
        }
    }
```

```
}

protected function commitToDB( managedObject : IManaged ) : void {
    _desktopWrapper.save(
        managedObject )                    // After saving, go to edit mode and render
 the saved object.
                    currentState = "edit";
                    _editFieldContainer.fieldCollection.render( managedObject );

                    // Show the status message
                    _app.setStatus(
                        new F3Message(
                            F3Message.STATUS_INFO,
                            "Account saved"
                        )
                    );
                }

protected function onDataGridItemClick() : void {
    _app.setStatus( null );

    var account : Account = _dataGrid.selectedItem as Account;

    if ( account == null ) {
        // nothing selected -> go back to default state
        currentState = "default";
    }
    else {
        // Go to edit mode and render the selected account
        currentState = "edit";
        editFieldContainer.fieldCollection.render( account );

        var status : F3Message = new F3Message(
            F3Message.STATUS_INFO,
            "Account with id: " +
                account.Id +
                ", name: " +
                account.Name +
                " selected"
        );

        // Show the message in the status bar
        _app.setStatus( status );

        // Show the toaster
        _app.showToaster(
            new Toaster(
                StaticAssets.ACCOUNT_IMAGE_32,
                "Account Selected",
                status.description,
                account.Name,
                onToasterSelected
            )
        );
```

```
    }
}

protected function onQueryResult( rows : ArrayCollection ) : void {
        _desktopWrapper.releaseQueryResults( _gridDataProvider );

    // populate datagrid
    _gridDataProvider = rows;

    if ( _gridDataProvider == null )
        _gridDataProvider = new ArrayCollection();

    var status : F3Message = new F3Message(
        F3Message.STATUS_INFO,
        "Query returned with " +
            _gridDataProvider.length +
            " accounts"
    );

    // show message in status bar
    _app.setStatus( status );

    // show toaster
    _app.showToaster(
        new Toaster(
            StaticAssets.PRODUCT_IMAGE_32,
            "Query Results",
            status.description
        )
    );
}

protected function onToasterSelected( event : ToasterEvent ) : void {
    var len : int = _gridDataProvider.length;
    var context : String = Toaster( event.toaster ).context as String;
    for ( var i : int = 0; i < len; i++ ) {
        var entity : Object = _gridDataProvider.getItemAt( i );
        if (
            entity.hasOwnProperty( "Name" ) &&
            entity.Name == context
        ) {
            _dataGrid.selectedIndex = i;
            break;
        }
    }
}
```

19. Save the file.

The final code for the `AccountsView.mxml` file should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
```

```
    xmlns:mx="library://ns.adobe.com/flex/mx" width="400" height="300"
    xmlns:flexforforce="http://flexforforce.salesforce.com">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import com.salesforce.assets.StaticAssets;
            import com.salesforce.events.ToasterEvent;
            import com.salesforce.flexforforce.F3DesktopApplication;
            import com.salesforce.flexforforce.F3DesktopWrapper;
            import com.salesforce.notifications.F3Message;
            import com.salesforce.notifications.Toaster;

            import mx.collections.ArrayCollection;
            import mx.data.IManaged;
            import mx.logging.ILogger;
            import mx.logging.Log;
            import mx.rpc.Responder;
            import mx.rpc.events.FaultEvent;

            import services.flexforforce.Account;


            protected var _app : F3DesktopApplication;
            protected var _desktopWrapper : F3DesktopWrapper;
            private static const LOG : ILogger = Log.getLogger( "AccountsView" );
            // After executing a query, this is populated with the resulting rows
            [Bindable]
            protected var _gridDataProvider : ArrayCollection = new ArrayCollection();

            public function set app( app : F3DesktopApplication ) : void {
                _app = app;
                _desktopWrapper = app.wrapper;
            }

            protected function onCreateClick() : void {
                _app.setStatus( null );

                // switch to create mode and simply render a new account
                currentState = "create";
                _createFieldContainer.fieldCollection.render( new Account());
            }

            protected function onSyncClick() : void {
                _app.setStatus( null );

                _desktopWrapper.syncWithServer(
                    null,
                    new mx.rpc.Responder(
                        function( result : F3Message ) : void {
                            _app.setStatus( result );
```

```
                    },
                    function( result : F3Message ) : void {
                        LOG.error( "syncWithServer failed: " + result.toString());
                    }
                )
            );
        }

        protected function onQueryClick() : void {
            _app.setStatus( null );
            _desktopWrapper.query(
                "select * from Account",
                new mx.rpc.Responder(
                    onQueryResult,
                    function( result : F3Message ) : void {
                        LOG.error( "query failed: " + result.toString());
                    }
                )
            );
        }

        protected function onEditSaveClick() : void {
            _app.setStatus( null );

            // The existing account object has been updated in memory.
            // Now commit the updated object to the database.
            _editFieldContainer.fieldCollection.updateObject(
                new mx.rpc.Responder( commitToDB, saveFaultHandler )
            );
        }

        protected function onEditDeleteClick() : void {
            _app.setStatus( null );

            var account : Account = Account(
                _editFieldContainer.fieldCollection.managedObject
            );

            // clear and go back to empty mode
            _editFieldContainer.fieldCollection.clear();
            currentState = "default";

          _desktopWrapper.deleteItem( account );               var status : F3Message
= new F3Message(
                                F3Message.STATUS_INFO,
                                "Account deleted"
                            );

                            // Show the message in the status bar.
                            _app.setStatus( status );

                            // Show the toaster.
                            _app.showToaster(
                                new Toaster(
```

```
                                StaticAssets.REPORT_IMAGE_32,
                                "Account deleted",
                                status.description
                            )
                        );
                    }

    protected function onEditCancel() : void {
        _app.setStatus( null );

        // clear and go back to empty mode
        _editFieldContainer.fieldCollection.clear();
        currentState = "default";
    }

    protected function onCreateSaveClick() : void {
        _app.setStatus( null );

        // The new account object has been created in memory.
        // Now commit the new object to the database.
        _createFieldContainer.fieldCollection.updateObject(
            new mx.rpc.Responder( commitToDB, saveFaultHandler )
        );
    }

    protected function onCreateCancelClick() : void {
        _app.setStatus( null );

        // clear and go back to empty mode
        _createFieldContainer.fieldCollection.clear();
        currentState = "default";
    }

    protected function saveFaultHandler( msg : F3Message ) : void {
        // If there are invalid fields, stay on the edit screen
        // so the user has a chance to fix the errors.
        // Otherwise, clear the record.
        if (
            msg != null &&
            msg.messageType != F3Message.ERROR_INVALID_FIELDS
        ) {
            currentState = "default";
        }

        if ( msg != null ) {
            var faultEvent : FaultEvent = msg.context as FaultEvent;
            if ( faultEvent ) {
                LOG.error( faultEvent.fault.faultString );
                LOG.error( faultEvent.fault.faultDetail );
                LOG.error( faultEvent.fault.faultCode );
            }
        }
    }
```

```
            protected function commitToDB( managedObject : IManaged ) : void {
                _desktopWrapper.save(
                    managedObject )                    // After saving, go to edit mode
and render the saved object.
                                currentState = "edit";
                                _editFieldContainer.fieldCollection.render( managedObject
 );

                                // Show the status message
                                _app.setStatus(
                                    new F3Message(
                                        F3Message.STATUS_INFO,
                                        "Account saved"
                                    )
                                );
                            },

            protected function onDataGridItemClick() : void {
                _app.setStatus( null );

                var account : Account = _dataGrid.selectedItem as Account;

                if ( account == null ) {
                    // nothing selected -> go back to default state
                    currentState = "default";
                }
                else {
                    // Go to edit mode and render the selected account
                    currentState = "edit";
                    _editFieldContainer.fieldCollection.render( account );

                    var status : F3Message = new F3Message(
                        F3Message.STATUS_INFO,
                        "Account with id: " +
                            account.Id +
                            ", name: " +
                            account.Name +
                            " selected"
                    );

                    // Show the message in the status bar
                    _app.setStatus( status );

                    // Show the toaster
                    _app.showToaster(
                        new Toaster(
                            StaticAssets.ACCOUNT_IMAGE_32,
                            "Account Selected",
                            status.description,
                            account.Name,
                            onToasterSelected
                        )
                    );
                }
```

```
        }

        protected function onQueryResult( rows : ArrayCollection ) : void {
                _desktopWrapper.releaseQueryResults( _gridDataProvider );

            // populate datagrid
            _gridDataProvider = rows;

            if ( _gridDataProvider == null )
                _gridDataProvider = new ArrayCollection();

            var status : F3Message = new F3Message(
                F3Message.STATUS_INFO,
                "Query returned with " +
                    _gridDataProvider.length +
                    " accounts"
            );

            // show message in status bar
            _app.setStatus( status );

            // show toaster
            _app.showToaster(
                new Toaster(
                    StaticAssets.PRODUCT_IMAGE_32,
                    "Query Results",
                    status.description
                )
            );
        }

        protected function onToasterSelected( event : ToasterEvent ) : void {
            var len : int = _gridDataProvider.length;
            var context : String = Toaster( event.toaster ).context as String;
            for ( var i : int = 0; i < len; i++ ) {
                var entity : Object = _gridDataProvider.getItemAt( i );
                if (
                    entity.hasOwnProperty( "Name" ) &&
                    entity.Name == context )
                {
                    _dataGrid.selectedIndex = i;
                    break;
                }
            }
        }

    ]]>
</fx:Script>

<s:states>
    <s:State name="default"/>
        <s:State name="create"/>
        <s:State name="edit"/>
</s:states>
```

```
<s:VGroup
    width="100%"
    height="100%"
    paddingBottom="10"
    paddingLeft="10"
    paddingRight="10"
    paddingTop="10">

    <!-- top bar with top level buttons -->
    <s:HGroup width="100%" verticalAlign="middle">
        <s:Button label="Create" click="onCreateClick()"/>
        <s:Button label="Query" click="onQueryClick()"/>
        <s:Button label="Sync" click="onSyncClick()"/>
        <mx:Spacer width="100%"/>
    </s:HGroup>

    <s:Label text="Displaying: {_gridDataProvider.length} account(s)"/>

    <mx:DataGrid
        id="_dataGrid"
        width="100%"
        height="100%"
        resizeEffect="Resize"
        dataProvider="{_gridDataProvider}"
        itemClick="onDataGridItemClick()">

        <mx:columns>
            <mx:DataGridColumn dataField="Name"/>
            <mx:DataGridColumn dataField="BillingCity"/>
            <mx:DataGridColumn dataField="Type"/>
            <mx:DataGridColumn dataField="Website"/>
            <mx:DataGridColumn dataField="AnnualRevenue"/>
        </mx:columns>
    </mx:DataGrid>


    <!-- edit mode: show UI components in inline edit mode -->
    <flexforforce:FieldContainer
        id="_editFieldContainer"
        width="100%"
        includeIn="edit">

        <s:HGroup
            width="100%"
            paddingLeft="5"
            paddingRight="5"
            paddingBottom="5"
            paddingTop="5"
            verticalAlign="middle">

            <mx:Image source="{StaticAssets.ACCOUNT_IMAGE_32}"/>
            <s:Label text="Account Detail" fontWeight="bold"/>
            <mx:Spacer width="100%"/>
```

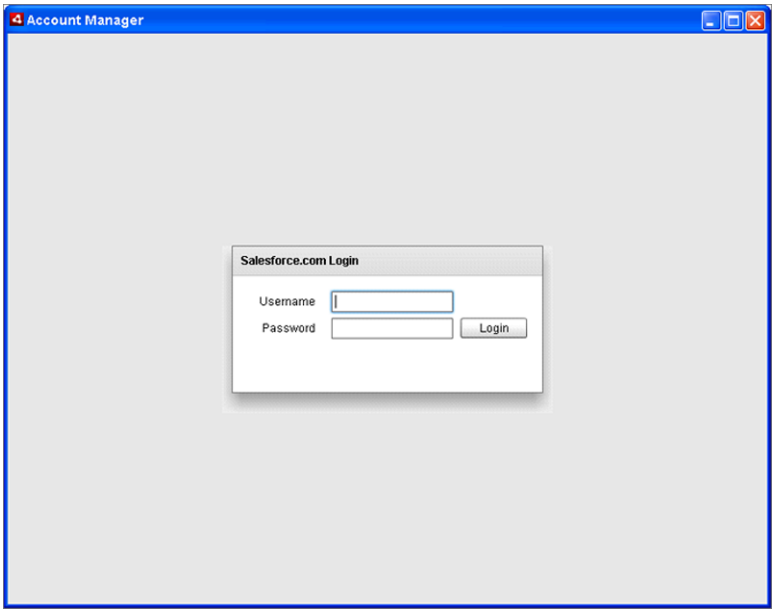
```
                <s:Button label="Save" click="onEditSaveClick()"/>
                <s:Button label="Delete" click="onEditDeleteClick()"/>
                <s:Button label="Cancel" click="onEditCancel()"/>
            </s:HGroup>

            <flexforforce:LabelAndField field="Account.Name"/>
            <!-- This expands to include all of the sub-fields in a Billing Address -->
            <flexforforce:LabelAndField field="Account.BillingStreet[Group]"/>
            <flexforforce:LabelAndField field="Account.ParentId"/>
            <flexforforce:LabelAndField field="Account.Type"/>
            <flexforforce:LabelAndField field="Account.Website"/>
            <flexforforce:LabelAndField field="Account.CreatedDate"/>
            <flexforforce:LabelAndField field="Account.AnnualRevenue"/>
        </flexforforce:FieldContainer>

        <!-- create mode: show UI components in full edit mode -->
        <flexforforce:FieldContainer
            id="_createFieldContainer"
            startState="Create"
            width="100%"
            includeIn="create">

            <s:HGroup
                width="100%"
                paddingLeft="5"
                paddingRight="5"
                paddingBottom="5"
                paddingTop="5"
                verticalAlign="middle">

                <mx:Image source="{StaticAssets.ACCOUNT_IMAGE_32}"/>
                <s:Label text="New Account" fontWeight="bold"/>
                <mx:Spacer width="100%"/>
                <s:Button label="Save" click="onCreateSaveClick()"/>
                <s:Button label="Cancel" click="onCreateCancelClick()"/>
            </s:HGroup>

            <flexforforce:LabelAndField field="Account.Name"/>
            <flexforforce:LabelAndField field="Account.ParentId"/>
            <flexforforce:LabelAndField field="Account.Type"/>
            <flexforforce:LabelAndField field="Account.Website"/>
            <flexforforce:LabelAndField field="Account.AnnualRevenue"/>
            </flexforforce:FieldContainer>

    </s:VGroup>

    <flexforforce:StatusBar/>

</s:Group>
```

STEP 9: TEST THE APPLICATION

The Account Manager application is ready to test.

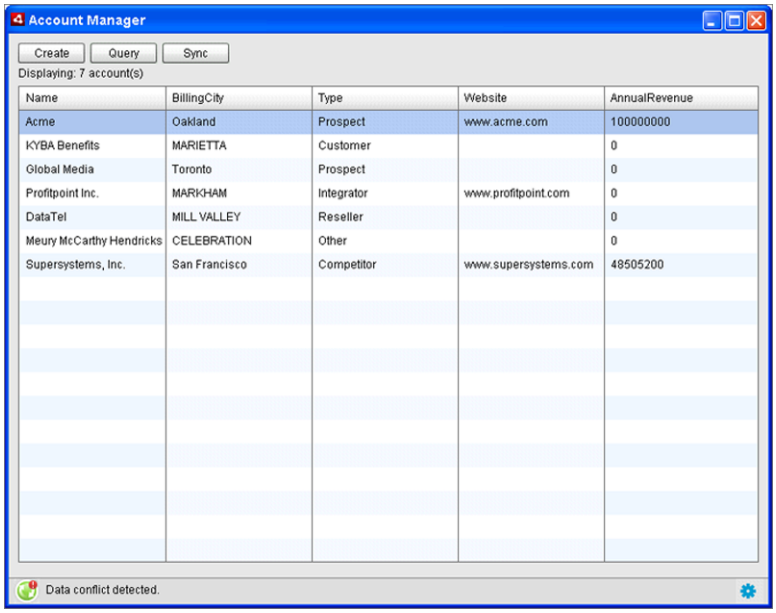
1. Verify your machine has an active Internet connection.
2. In Flash Builder, right-click on the Account Manager project and select **Run as...** > **Desktop Application**. The login screen appears.

The Account Manager Login Screen



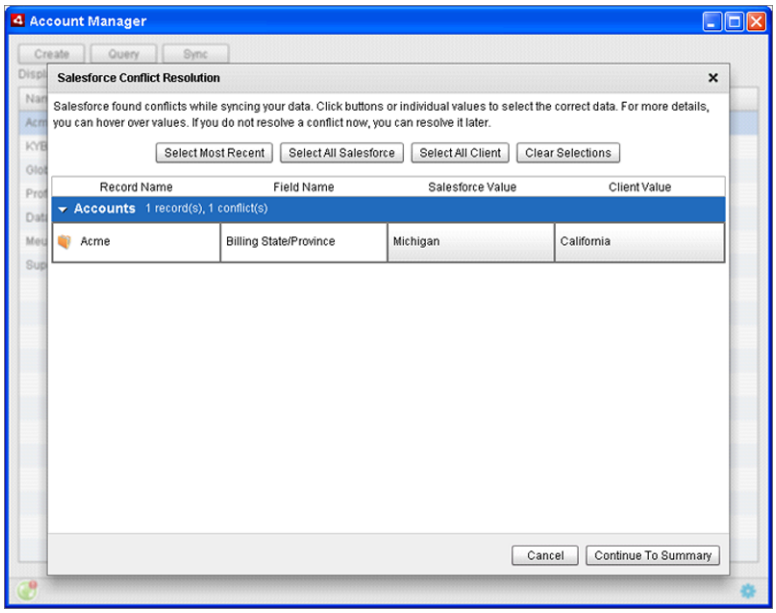
3. In the login screen, enter your Salesforce username and password and click **Login**. The Account Manager application appears.
4. Click **Query**. A list of account records appears.
5. Select a record in the list, change its data, and click **Save**.
6. Now log into Salesforce, navigate to the record you just modified, and verify that your change appears.
7. To test the data conflict resolution capability, change a field on an account record in Salesforce and click **Save**, then change the same field in the Account Manager application to a different value. When you click **Save** in the Account Manager, notice that the status bar alerts you that there is a conflict.

Data Conflict Notification



8. Click the green button and select **Resolve** in the status bar to launch the conflict resolution interface.

Conflict Resolution Interface



The conflict resolution interface lets you see the values in both Salesforce and the Account Manager application. Select the value you want to keep, then click **Continue To Summary**.

SUMMARY

In this tutorial, you created a simple Flex application that gives users the ability to view and edit Lightning Platform data both *with* and *without* an Internet connection. It demonstrates the fundamentals of Flex without much coding. As you can probably imagine, you can use Flex to create far more powerful Flex applications to enhance the ways in which users interact with your Lightning Platform applications.