*Public Beta 1*

# Flex Tutorials

*Public Beta 1*

**Trademarks**



**Third-Party Information**

This guide contains links to third-party websites that are not under the control of Adobe Systems Incorporated, and Adobe Systems Incorporated is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Adobe Systems Incorporated provides these links only as a convenience, and the inclusion of the link does not imply that Adobe Systems Incorporated endorses or accepts any responsibility for the content on those third-party sites.



**Acknowledgments**

Project Management:

Writing:

Editing:

Production Management:

Media Design and Production:

Special thanks to

First Edition: January 2006

Adobe Systems Incorporated
601 Townsend St.
San Francisco, CA 94103

# Contents

# 1 Introduction

This book includes several step-by-step tutorials designed to teach you the fundamentals of Flex, including how to create rich user interfaces with contraint-based layouts and view states, and how to access data. By completing these hands-on lessons, you'll not only quickly acquire the basics, you'll also learn how to use Adobe Flex Builder 2, the Eclipse-based, integrated development environment for Flex.

The lessons are targeted toward beginner to intermediate-level developers who want to get up to speed quickly. Each lesson focuses on a specific Flex feature or topic and takes approximately 10-20 minutes to complete, depending on your experience.

This book is not a comprehensive manual detailing all the features of Adobe Flex. For in-depth information about Flex, see the full documentation in Flex Builder help or on the Adobe Labs website at http://labs.adobe.com.

CHAPTER 8

# Basic: Create a Project

8

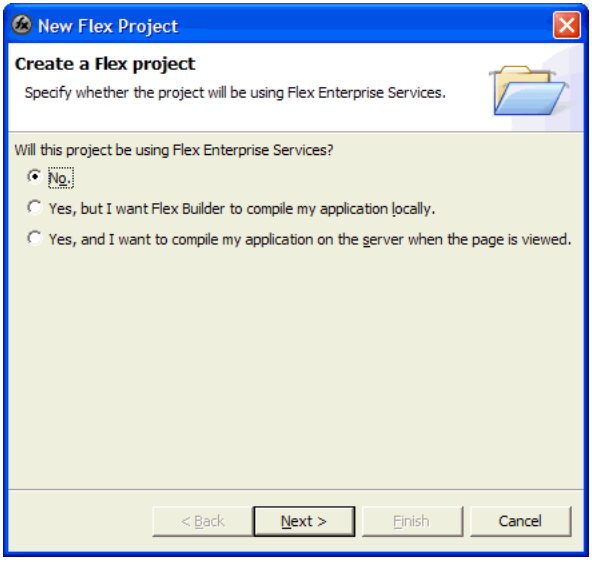
This lesson introduces you to the concept of projects in Adobe Flex Builder 2 and shows you how to create projects. In Flex Builder, all Flex applications are contained within projects.

Before building a Flex application in Flex Builder, you must create a project. When you create a project in Flex Builder, a mainapplication file is created for you. You can add resources to a project, such as custom MXML component files, ActionScript files, and other assets that make up your Flex application.

1. Start Flex Builder and select File > New > Flex Project from the main menu.

   If you have the plug-in version of Flex Builder and you have a non-Flex-Builder perspective open in Eclipse, select New > Other > Flex > Flex Project.

   The New Flex Project wizard appears.




   The wizard guides you through the steps of creating a project.

2. Because you won't be using a Flex server in this tutorial, select the No option and click Next.

   The next screen asks you to specify the name of the project and the location to store its files.

3. In the Project Name text box, enter **Lessons**.

   This is the name of your project. When you create a project, Flex Builder generates a main Flex application file based on the project name. Because a main application file uses the same name, you can't use spaces or special characters for the project name.

4. In the Project Location text box, make sure the location for your project files is as follows:

   C:\Documents and Settings*your_user_name*\My Documents\Flex\Lessons

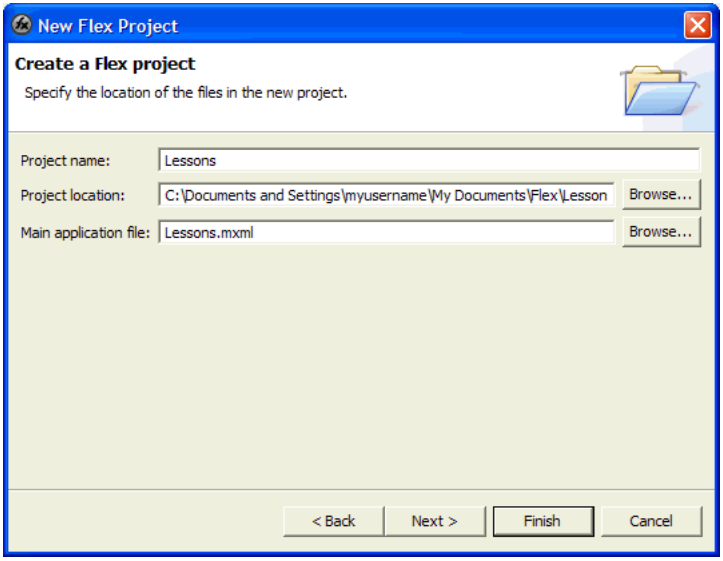
   Flex Builder will create this folder for you.

   The default location for new projects is the Flex folder in the My Documents folder.

5. Ensure that the Main Application File text box specifies Lessons.mxml.

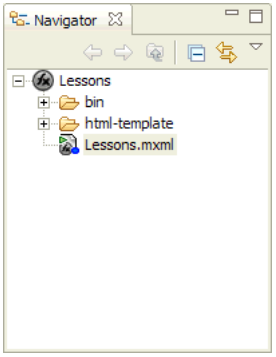
   This option determines which MXML file in your project is the main application file when compiling.

   The New Flex Project wizard should look as follows:



6. Click Finish.

   Flex Builder creates a new project and displays it in the Navigator view.




   The New Flex Project wizard automatically generates project configuration files, the output (bin) folder where your compiled SWF file will be placed, and the main application file, Lessons.mxml.

In this lesson, you learned how to create a project in Flex Builder. To learn more about projects, see Chapter 5, “Working with Projects,” on page 51.

*Public Beta 1 Public Beta 1 Public Beta 1 Public*

CHAPTER 9

# Basic: Build an Application

9

This lesson shows you how to build and run a Flex application in Adobe Flex Builder.

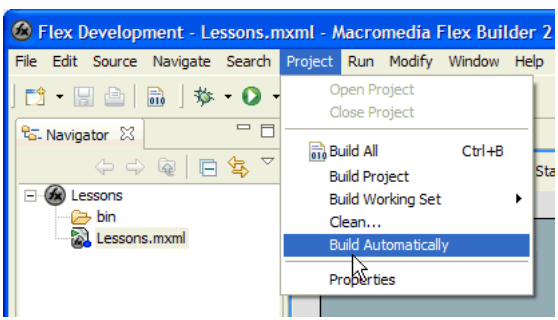
In this tutorial, you'll complete the following tasks:



## Set up your project

Before you begin this lesson, perform the following tasks:

- If not already done, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

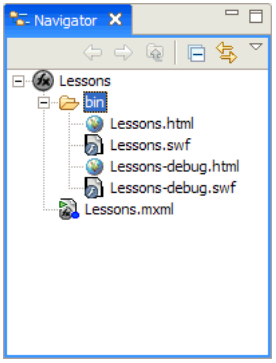


# Learn about building in Flex Builder

Before you build an application in Flex Builder, it's helpful to review some key concepts.

By default, the standalone configuration of Flex Builder automatically builds—or compiles—the application when you add a file to the project or when you save a project file after editing it in Flex Builder. Automatic builds are disabled by default in the plug-in configuration of Flex Builder, but you can enable this option by selecting Project > Build Automatically.

After building an application, Flex Builder places the resulting Flash file (SWF) into the bin folder, which is the default folder in which compiled files are placed.




Flex Builder also generates an HTML wrapper file for the SWF file, in case you want to run the SWF in a web browser.

| NOTE | The Flash Player 8.5 browser plug-in is required to run Flex 2 applications in a browser. |
|---|---|

When you create a project, Flex Builder creates a main application file and names it based on the project name. An *application file* is an MXML file with an `<mx:Application>` parent tag. Your project can have several application files, but the main application file is the file Flex Builder compiles into the application SWF file during builds. You can designate another application file as the main application file to be built, but it's good practice to have only one application file per project.

Now that you understand the basic concepts of building applications in Flex Builder, you can create a small application in Flex Builder, and then build and run it.

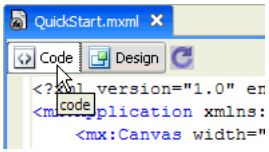
For more information about how projects are built, see "Understanding how projects are built" in the "Building Projects" chapter of *Using Flex Builder 2*.

# Build and run an application

When automatic builds are enabled, Flex Builder automatically builds your application when you add a file to the project or when you save a project file after editing it in Flex Builder.

The steps in this section assume you created the Lessons project and that automatic builds are enabled. For more information, see "Set up your project" on page 99.

1. Double-click the Lessons.mxml file to open it in Flex Builder.
2. Switch to the editor's Code mode by clicking the Code button in the document toolbar.




Flex Builder inserted the following code in the Lessons.mxml file when it created it:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">
</mx:Application>
```

3. Enter the following tag between the opening and closing `<mx:Application>` tags:

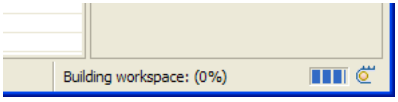
```
<mx:Label text="Welcome to Flex!" mouseDownEffect="WipeRight" x="20"
  y="20" />
```

This tag inserts and positions a Label control in the layout.

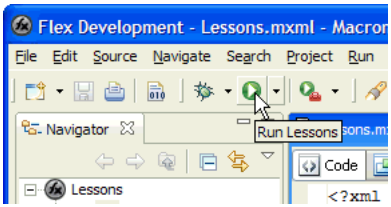
| TIP | You can preview the control by clicking the Design button in the document toolbar (see the image in step 2). |
|---|---|

4. Save the file.

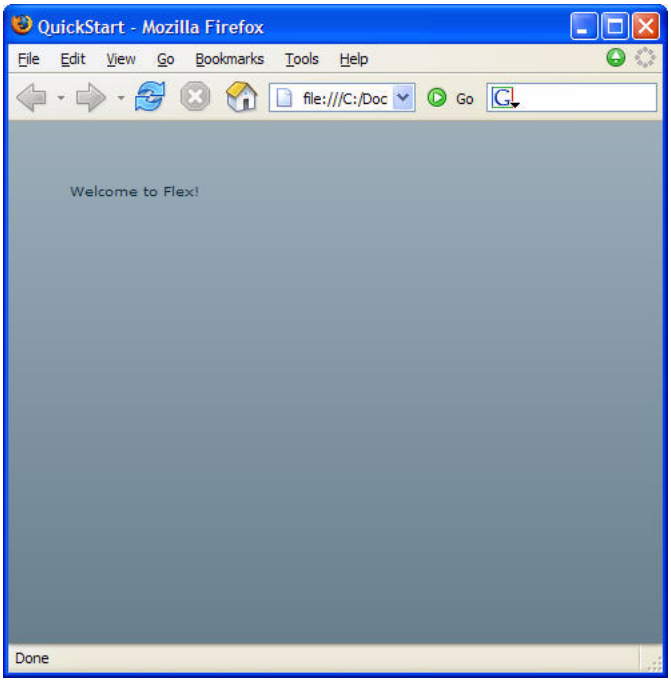
Flex Builder automatically builds the application when you save a file. You can monitor the build progress with the indicator at the bottom-right corner of the window. Otherwise, you can keep working.



5. After the build is complete, click the Run button in the toolbar to start the application.




A browser opens and runs the application.



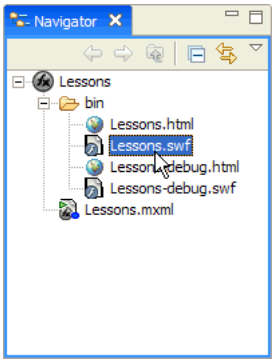

| NOTE | The browser must have Flash Player 8.5 installed to run the application. You have the option of installing this version of Flash Player in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 8.5, select Window › Preferences › General › Web Browser. |
|---|---|

6. Click the "Welcome to Flex!" text to see the WipeRight effect.

   To learn more about effects, do the lesson in Chapter 13, "Design: Use Behaviors," on page 133, or see Chapter 22, "Using Behaviors" in *Developing Flex Applications*.

## *Public Beta 1 Public Beta 1 Public Beta 1 Public*

You can deploy your application to a web server by uploading the SWF file generated by Flex Builder. You can also upload its HTML wrapper file (Lessons.html) to run the SWF in a web browser. The files are located in the bin folder.




In this lesson, you learned how to build and run a Flex application in Flex Builder. To learn more about this topic, see "Building Projects" on page 189.

CHAPTER 11

# Design: Create a Constraint-based Layout

11

This lesson shows you how to create a constraint-based layout with Adobe Flex Builder. A constraint-based layout ensures that the components in your user interface adjust automatically when a user resizes the application window.

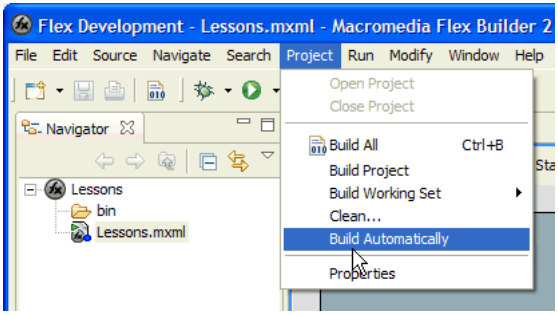
In this tutorial, you'll complete the following tasks:



## Set up your project

Before you begin this lesson, perform the following tasks:

- If not already done, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.



# Learn about constraint-based layouts in Flex

When a user resizes a Flex application window, you want the components in your layout to adjust automatically. A constraint-based layout adjusts the size and position of components when the user resizes the application window.

To create a constraint-based layout, you must use a container with a `layout` property set to absolute (`layout="absolute"`). This property gives you the flexibility of positioning and sizing components with absolute positioning while also providing you with the ability to set constraints that stretch and move the components when the container is resized.

NOTE

The Canvas container does not require the `layout="absolute"` property because its layout is absolute by default.

For example, if you want a TextInput text box to stretch when the user makes the application window wider, you can anchor the control to the left and right edges of the container so that the width of the text box is set by the width of the window.

In Flex, all constraints are set relative to the edges of the container. They cannot be set relative to other controls.

Now that you understand the basic concepts, you can create a simple layout and define layout constraints for it in Flex Builder.

# Insert and position the components

The first step in creating a constraint-based layout is to position the components in a container with a `layout` property set to absolute. This property allows you to drag and position components anywhere in the container. For pixel-point accuracy, you can set *x* and *y* coordinates.

In this section, you insert and position the controls of a simple feedback form.

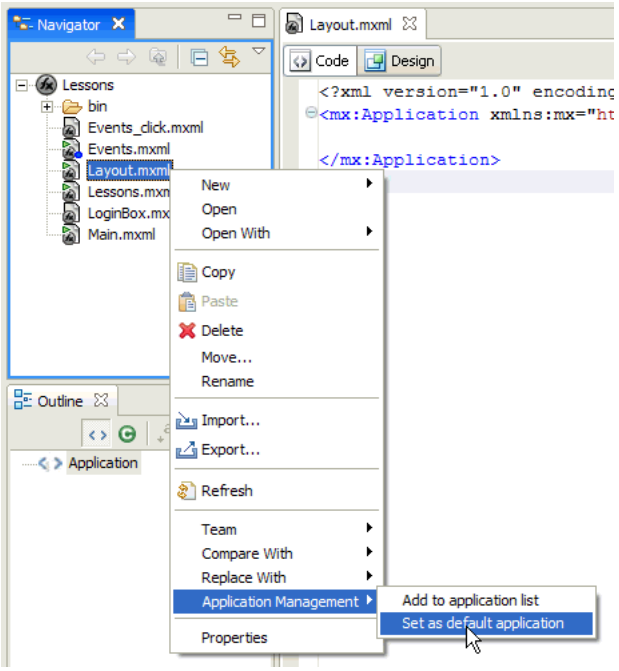
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Layout.mxml.

   By default, Flex Builder includes the `layout="absolute"` property in the Application tag.

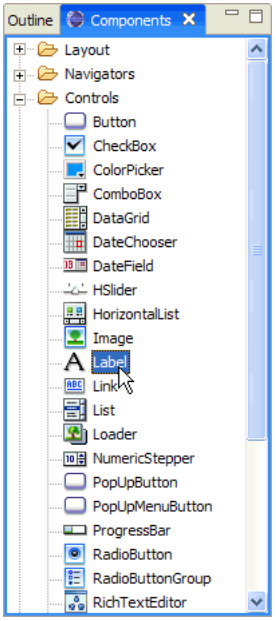
   NOTE

   For the purpose of these short tutorials, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Layout.mxml file as the default file to be compiled by right-clicking the file in the Navigator view and selecting Application Management > Set As Default Application from the context menu.



3. In the editor's Design mode, add a Label and TextInput control to the Layout.mxml file by dragging them from the Components view (Window > Show View > Components).

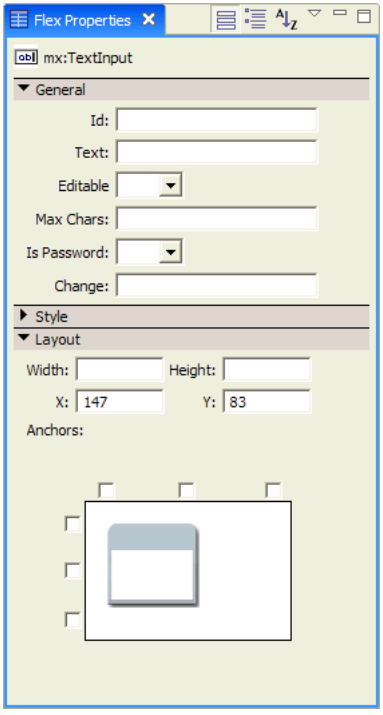



4. Use the pointer to position the Label and TextInput controls side-by-side about 60 pixels (two centimeters)from the top of the container.

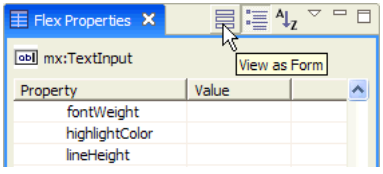
5. In the Flex Properties view, expand both the General and the Layout categories of properties.

| TIP | You may need to collapse the States view to see the Layout category. |
|---|---|

Options for setting the general and layout properties appear.




If you see a table of properties instead of the previous view, click the View As Form button in the view's toolbar.



6. Select the Label control in the layout and set the following Label properties in the Flex Properties view:
   - Text: **Email**
   - X: **20**
   - Y: **60**
7. Select the TextInput control in the layout and set the following TextInput properties:
   - X: **90**
   - Y: **60**
   - Width: **300**
8. Switch to the editor's Source mode by clicking the Code button in the document toolbar.

   The Layout.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">
  <mx:Label x="20" y="60" text="Email"/>
  <mx:TextInput x="90" y="60" width="300"/>
</mx:Application>
```

9. Insert the remaining Flex controls by entering the following additional tags after the `<mx:TextInput>` tag:

```
<mx:Label x="20" y="90" text="Comments"/>
<mx:TextArea x="90" y="90" width="300" />
<mx:Button x="330" y="150" label="Send"/>
```

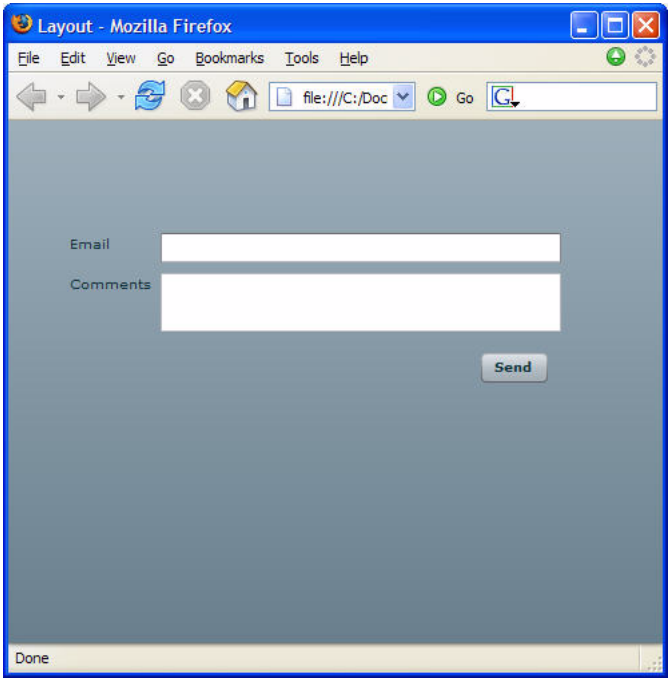
You can preview the layout by clicking the Design button in the toolbar. The layout should look similar to the following:


10. Save the file.

Flex Builder compiles the application.

**11.** Click the Run button in the toolbar.

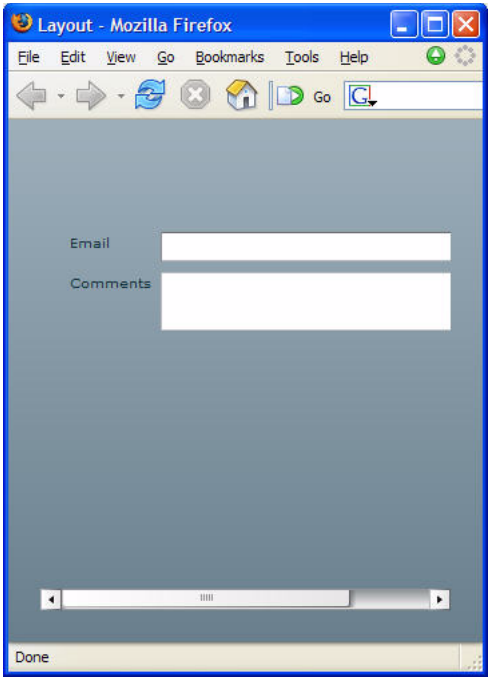
A browser opens and runs your small Flex application.




| NOTE | The browser must have Flash Player 8.5 installed to run the application. You have the option of installing this version of the Player in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 8.5, select Window › Preferences › General › Web Browser. |
|---|---|

**12.** Drag the edges of the browser window to make the application bigger and smaller.

The components maintain their position relative to the left and top edges of the window, but they don't stretch or compress as you resize the browser window. For example, if you make the window too narrow, the Send button disappears, and the TextInput and TextArea controls are clipped.




NOTE

When content is clipped, Flex automatically provides users with a vertical or horizontal scrollbar to access the content.

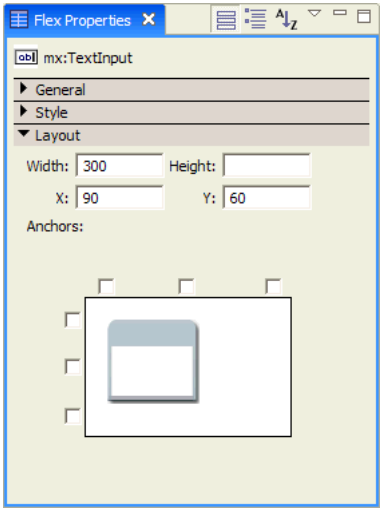
The next step is to set layout constraints for the controls so that they adjust when the user resizes the application window.

# Define the layout constraints

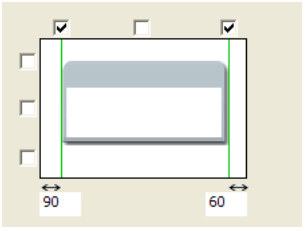
After positioning the components in your layout, you define layout constraints so that the components adjust when a user resizes the application window.

1. In the editor's Design mode, select the TextInput control (the text box for the e-mail address).
2. In the Flex Properties view, ensure that the Layout category of properties is expanded.

   The Layout category contains options for setting anchors.




3. Define the layout constraints for the TextInput control by selecting the left and right anchor check boxes in the view, and then specifying **90** as the distance to maintain from the left window edge and **60** as the distance to maintain from the right edge, as follows:




   The two check boxes anchor the TextInput control to the left and right edges of the window. The numbers associated with the text boxes specify how far from the edges (in pixels) to anchor the controls.

The left edge anchor is necessary to fix the control in place so that it stretches or compresses when the user resizes the application horizontally. Without the left anchor to hold it in place, the control would slide to the left or right.

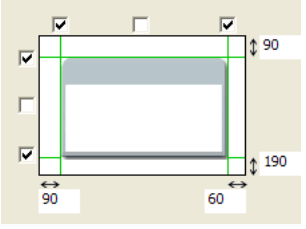
These constraints are expressed as follows in the MXML code:

```
<mx:TextInput y="60">
  <mx:layoutConstraints>
  <mx:Anchor left="90" right="60"/>
  </mx:layoutConstraints>
</mx:TextInput>
```

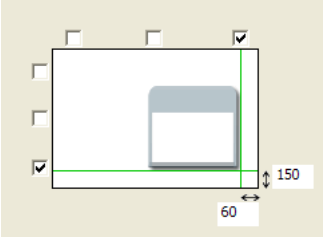
4. In the editor's Design mode, select the TextArea control in the layout and, in the Flex Properties view, select the four corner check boxes and specify the following distances to maintain from the edges:

   - Left: **90**
   - Right: **60**
   - Top: **90**
   - Bottom: **190**

   The Layout category in the Flex Properties view for the TextArea control should look as follows:



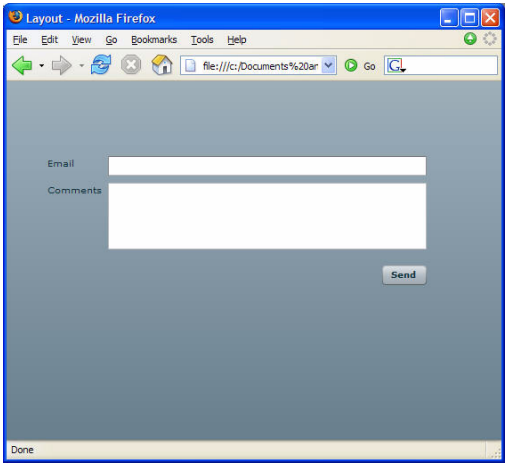

5. Select the Button control in the layout and, in the Flex Properties view, click the right and bottom anchor check boxes, and specify **60** as the distance to maintain from the right edge and **150** as the distance to maintain from the bottom edge, as follows:



The two check boxes anchor the Button control to the right and bottom edges. With no anchors to fix the control to the left and top edges, the control moves horizontally or vertically as the user resizes the application.

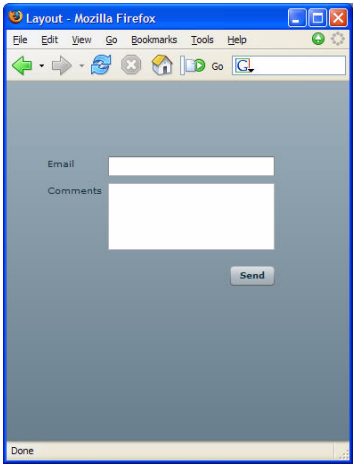
6. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar.

   A browser opens and runs your small Flex application.

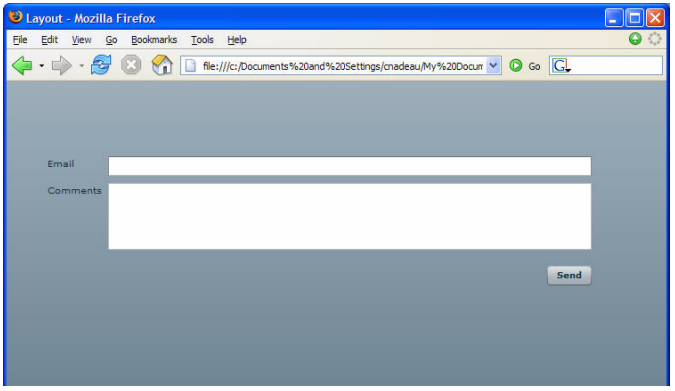



7. Drag the edges of the browser window to make the application bigger and smaller.

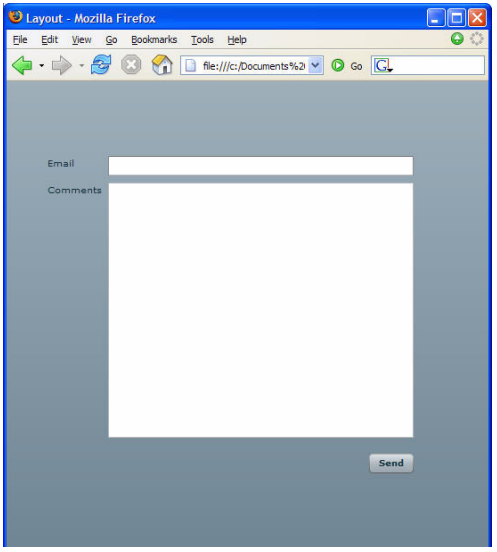
   For example, if you make the window narrower, the Send button moves to the left and the TextInput and TextArea text boxes become narrower.



If you make the browser window wider, the Send button moves to the right and the TextInput and TextArea text boxes become wider.




If you make the window taller, the Send button moves down and the TextArea becomes taller.



## *Public Beta 1 Public Beta 1 Public Beta 1 Public*

In this lesson, you learned how to create a constraint-based layout with Flex Builder. The following table summarizes the anchors to set for obtaining certain layout effects when the user resizes the application window.

| Effect | Anchors |
|---|---|
| Maintain the component's position and size | None |
| Move the component horizontally | Right |
| Move the component vertically | Bottom |
| Move the component both horizontally and vertically | Right + Bottom |
| Resize the component horizontally | Left + Right |
| Resize the component vertically | Top + Bottom |
| Resize the component both horizontally and vertically | Left + Right + Top + Bottom |
| Center the component horizontally | Horizontal center |
| Center the component vertically | Vertical center |

To learn more about this topic, see "Laying out your user interface" on page 114.

CHAPTER 12

# Design: Use View States and Transitions

12

You can use view states and transitions in Flex to create richer, more interactive user experiences. For example, you can use view states to create a user interface that changes its appearance based on the task the user is performing.

A *view state* is a named layout that you define for a single MXML application or component. You can define several view states for an application or component, and switch from one view state to another depending on the user's actions. View states allow you to dynamically change the user interface in response to users' actions or progressively reveal more information based on the context.

A *transition* is one or more effects grouped together to play when a view state changes. The purpose of a transition is to smooth the visual change from one state to the next.

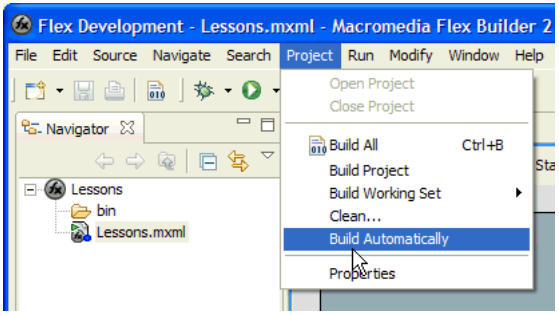
This lesson shows you how to use view states and transitions to create a user interface that reveals more information when users request it.

In this lesson, you'll complete the following tasks:

# Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.




# Design the base state

Before you can use view states, you must design the base state of the application or component. The base state is the default layout of the application or custom component.

In this section, you insert and position the controls of a simple search form to create the base state.

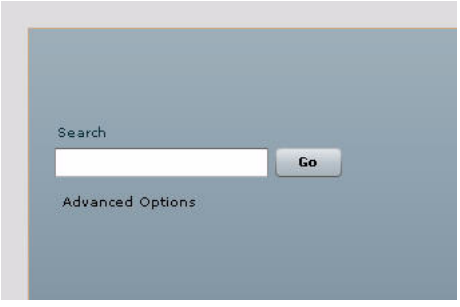
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called ViewStates.mxml.

| NOTE | For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project. |
|---|---|

2. Designate the ViewStates.mxml file as the default file to be compiled by right-clicking the file in the Navigator view and selecting Application Management > Set As Default Application from the context menu.

3. In the editor's Design mode, add the following controls to the ViewStates.mxml file by dragging them from the Components view (Window > Show View > Components):
   - Label
   - TextInput
   - Button
   - Link
4. Select the Label control in the layout and set the following Label properties in the Flex Properties view:
   - Text: **Search**
   - X: **20**
   - Y: **70**
5. Select the TextInput control and set the following TextInput properties in the Flex Properties view:
   - X: **20**
   - Y: **90**
6. Select the Button control and set the following Button properties in the Flex Properties view:
   - Label: **Go**
   - X: **185**
   - Y: **90**
7. Select the Link control and set the following Link properties in the Flex Properties view:
   - Label: **Advanced Options**
   - X: **20**
   - Y: **120**

   The layout should look similar to the following:



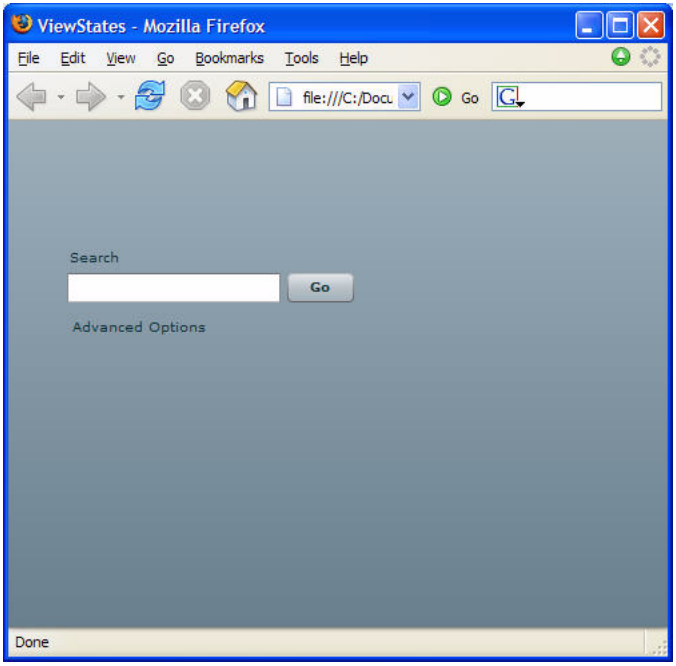
8. Switch to the editor's Source mode by clicking the Code button in the document toolbar.

   The ViewStates.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute"><<Update URL (macromedia.com)?-ls>>
  <mx:Label x="20" y="70" text="Search"/>
  <mx:TextInput x="20" y="90"/>
  <mx:Button x="185" y="90" label="Go"/>
  <mx:Link x="20" y="120" label="Advanced Options"/>
</mx:Application>
```

9. Save the file, wait until Flex Builder compiles the application, and click the Run button in the toolbar.

   A browser opens and runs the application.

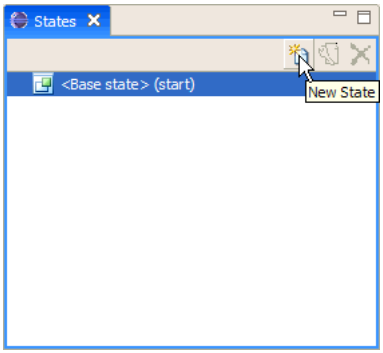



| NOTE | The browser must have Flash Player 8.5 installed to run the application. You have the option of installing this version of Flash Player in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 8.5, select Window › Preferences › General › Web Browser. |
|---|---|

# Design a view state

The sample application provides a simple search mechanism that meets the needs of most users. However, some users might prefer to have more search options. You can use view states to provide these options on request.

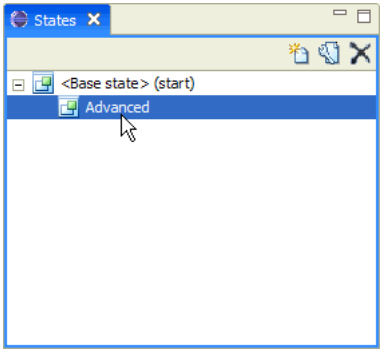
1. In the editor's Design mode, click the New State button in the States view (Window > Show View > States).




The New State dialog box appears.


2. Enter **Advanced** in the Name text box and click OK.

The new state appears in the States view.

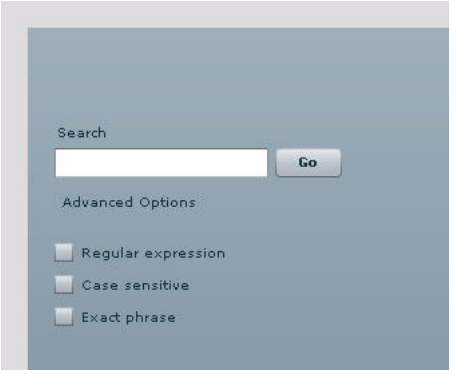


You can use the layout tools in Flex Builder to make changes to the appearance of the new state. You can modify, add, or delete components. As you work, the changes describing the new state are recorded in the MXML code.

3. In the editor's Design mode,insert a VBox container below the Advanced Search link, and then set the following VBox properties in the Flex Properties view:
   - ID: **myVBox**
   - Width: **140**
   - Height: **80**
   - X: **20**
   - Y: **160**
4. Drag three CheckBox controls into the VBox container.

   The VBox container automatically aligns the controls vertically.
5. Select the first CheckBox control in the VBox container and enter **Exact phrase** as the value of its Label property in the Flex Properties view.
6. Select the second CheckBox control and enter **Case sensitive** as the value of its Label property.
7. Select the third CheckBox control and enter **Regular expression** as the value of its Label property.

   The layout should look similar to the following:



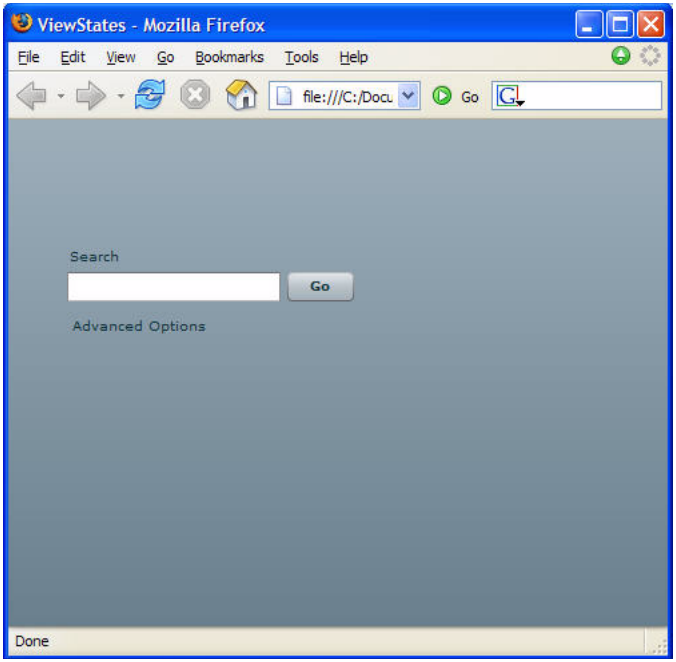
8. Switch to the editor's Source modeby clicking the Code button in the document toolbar.

   Flex Builder inserted the following `<mx:states>` tag after the opening `<mx:Application>` tag:

```
<mx:states>
  <mx:State name="Advanced">
    <mx:AddChild position="lastChild">
      <mx:VBox width="140" height="80" x="20" y="160" id="myVBox">
        <mx:CheckBox label="Regular expression"/>
        <mx:CheckBox label="Case sensitive"/>
        <mx:CheckBox label="Exact phrase"/>
      </mx:VBox>
    </mx:AddChild>
  </mx:State>
</mx:states>
```

9. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar.

   A browser opens and runs the application.

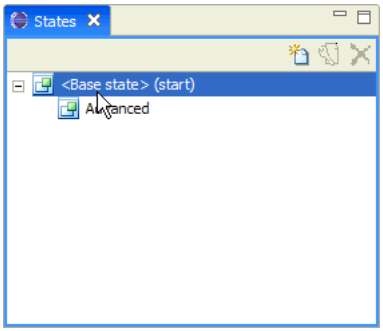



   Your application does not display the controls you inserted in the new view state. Flex applications display the base state by default. You must define how users switch view states, typically by clicking specific controls.

# Define how users switch to the view state

In your application, you want to display the check boxes in the new view state when the user clicks the Link control labelled Advanced Options. When the user clicks the link a second time, you want to hide the check boxes.

1. In the code editor's Design view, select the base state in the list in the States view.

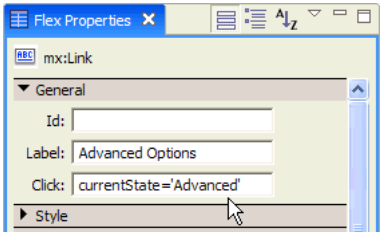



You want to define a click event handler for the Link control that is part of the base state. Therefore, you need to change the focus of the editor's Design mode to the base state.

When you select a state in the States view, Flex Builder displays what that state looks like. Therefore, when you select the base state in this step, Flex Builder hides the three CheckBox controls you defined for the Advanced view state.

2. Select the Link control in the layout and specify the following `click` property in the Flex Properties view:

```
currentState='Advanced'
```




The `click` property specifies that when the user clicks the Link control, the application should switch the current state to the view state called Advanced. This view state displays three additional check boxes.

Next, hide the check boxes when the user clicks the Link control a second time. You can do this by restoring the base state when the user clicks the link in the Advanced view state.

3. In the States view, select the Advanced state.

4. Select the Link control in the layout of the Advanced view state and specify the following `click` property in the Flex Properties view:

   ```
   currentState=''
   ```

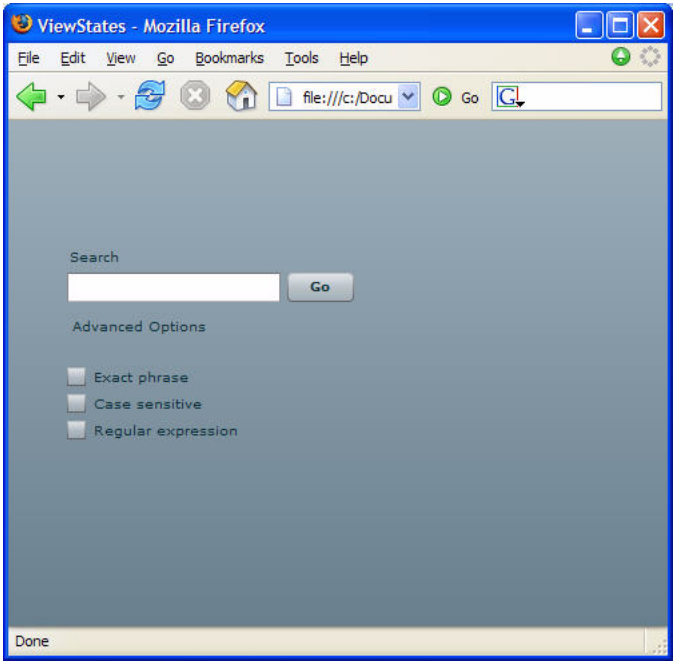
   Specify an empty string (two single quotes,' ') as the value of `currentState`. An empty string specifies the base state, so that when the user clicks the Link control while in the Advanced view state, the base state is restored.

5. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

   A browser opens and runs the application.

6. Click the Link control to view the advanced search options.

   The application displays the three check boxes you defined in the Advanced view state. The check boxes appear to jump into existence.




7. Click the Link control again to restore the base state, which hides the advanced search options.

# Create a transition

When you change the view states in your application, the check boxes immediately appear on the screen. You decide to eliminate this jumpiness by defining a Flex transition that uses the WipeDown and Dissolve effects to make the check boxes appear more gradually.

1. In the editor's Source mode, define a new Transition object and the view state change that triggers it by adding the `<mx:transitions>` tag immediately following the closing `<mx:states>` tag, as shown in the following example:

   ```
   <mx:transitions>
     <mx:Transition id="myTransition" fromState="*" toState="Advanced">
     </mx:Transition>
   </mx:transitions>
   ```

   You define one Transition object called myTransition for your application. You can define more than one transition in the `<mx:transitions>` tag.

   You also specify that you want the transition to be performed when the application changes from any view state (`fromState="*"`) to the view state called Advanced (`toState="Advanced"`). The value `"*"` is a wildcard character specifying any view state.

2. Specify the targeted component for the transition, and how you want the effects to play—simultaneously or sequentially—by entering the following `<mx:Parallel>` tag between the `<mx:Transition>` tags (in bold):

   ```
   <mx:Transition id="myTransition" fromState="*" toState="Advanced">
     <mx:Parallel target="{myVBox}">
     </mx:Parallel>
   </mx:Transition>
   ```

   The targeted component for the transition is the VBox container named myVBox.

   You should define two effects for your transition—a WipeDown effect and a Dissolve effect—and you want them to play simultaneously. Therefore, you use the `<mx:Parallel>` tag. If you wanted the effects to play sequentially, you would use the `<mx:Sequence>` tag instead. In that case, the second effect would not start until the first effect finished playing.

3. Specify the effects to play when the view state changes by entering the following `<mx:WipeDown>` and `<mx:Dissolve>` tags between the `<mx:Parallel>` tags (in bold):

   ```
   <mx:Parallel target="{myVBox}">
     <mx:WipeDown duration="2000"/>
     <mx:Dissolve alphaFrom="0.0" alphaTo="1.0" color="0x99CCCC"
     duration="2000"/>
   </mx:Parallel>
   ```

You want to play two effects, a WipeDown effect that causes the targeted container to appear from top to bottom over a period of 2000 milliseconds, or 2 seconds, and a Dissolve effect that causes an opaque rectangle covering the container to gradually become transparent in 2 seconds.

The completed `<mx:transitions>` tag should look as follows:

```
<mx:transitions>
  <mx:Transition id="myTransition" fromState="*" toState="Advanced">
    <mx:Parallel target="{myVBox}">
      <mx:WipeDown duration="2000"/>
      <mx:Dissolve alphaFrom="0.0" alphaTo="1.0" color="0x99CCCC"
  duration="2000"/>
    </mx:Parallel>
  </mx:Transition>
</mx:transitions>
```

4. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

   A browser opens and runs the application.

5. Click the Link control to view the advanced search options.

   The WipeDown and Dissolve effects play simultaneously, causing the advanced search options to appear gradually from top to bottom.

In this lesson, you used view states and transitions to create a more flexible user interface that provides users with more options on request. To learn more, see Chapter 8, “Adding Interactivity with View States and Transitions,” on page 139 and the following chapters in *Developing Flex Applications*:

- Chapter 25, “Using View States”
- Chapter 26, “Using Transitions”

CHAPTER 13

# Design: Use Behaviors

13

Flex behaviors let you add animation and motion to your application in response to user or programmatic action. A behavior is a combination of a *trigger* paired with an *effect*. A trigger is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. An effect is a visible or audible change to the target component that occurs over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component.

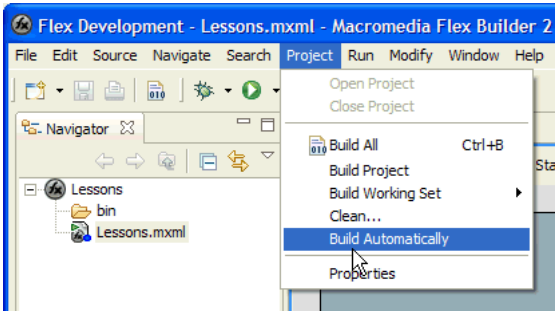
This lesson shows you how to add behaviors to a Flex user interface. It shows you how to use MXML to create behaviors, how to invoke an effect from a different component, and how to combine more than one effect to create a composite effect.

In this lesson, you'll complete the following tasks:

# Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.




# Create a behavior

You decide to create a behavior to make a button glow when a user clicks it. You want the glow to be green, last one and a half seconds, and leave the button a pale green to indicate it was clicked.

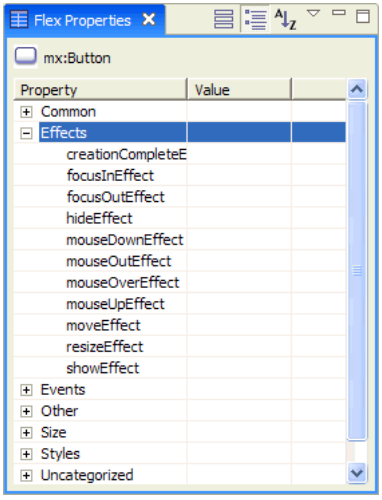
1. In the editor's Source mode, define a Glow effect by entering the following tag after the opening `<mx:Application>` tag:

   ```
   <mx:Glow id="buttonGlow" color="0x99FF66" alphaFrom="1.0" alphaTo="0.3"
     duration="1500"/>
   ```

   The Glow effect starts fully opaque and gradually becomes more transparent—but not fully transparent. A pale glow persists after the effect has played to indicate that the button was clicked.

2. In the editor's Design mode, drag a Button control from the Components panel into the layout, and then set the following Button properties in the Properties view:
   - ID: **myButton**
   - Label: **View**
   - X: **40**
   - Y: **60**

3. In the Properties view, click the View by Category button on the toolbar to view the properties as a table, and then locate the Effects category of properties.




   This category lists the triggers for the Button control.

4. Assign your Glow effect to the mouseUpEffect trigger by entering the effect's ID in curly braces as the value of the trigger, as follows:

   - mouseUpEffect: {**buttonGlow**}

   The curly braces are necessary because you use data binding to assign the effects to their triggers.

   In the editor's Source mode, the `<mx:Button>` tag should look as follows:

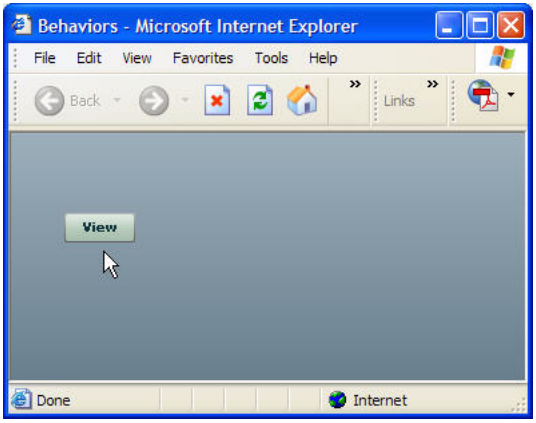
```
<mx:Button x="40" y="60" label="View" id="myButton"
  mouseUpEffect="{buttonGlow}" />
```

5. Save the file.

   Flex Builder compiles the application.

6. Click the Run button in the toolbar.

   A browser opens and runs your small Flex application. Click the View button. It should emit a green glow that gradually diminishes in intensity until it becomes a pale green.




# Invoke an effect from a different component

Instead of component triggers, you can use Flex events to invoke effects. This ability lets you have one component invoke an effect that plays for a different component. For example, you can use a Button control's click event to instruct a TextArea control to play a Fade effect.

When the user clicks your application's View button, you want a Label component to appear with blurry text that gradually comes into focus to reveal a series of numbers.

1. In the editor's Design mode, drag a Label control from the Components panel into the layout below the button, and then set the following Label properties in the Properties view:
   - ID: **myLabel**
   - Text: **4 8 15 16 23 42**
   - X: **40**
   - Y: **100**
2. Switch to the editor's Source mode and define your Blur effect by entering the following tag after the `<mx:Glow>` tag:

```
<mx:Blur id="numbersBlur"
  blurYFrom="10.0" blurYTo="0.0"
  blurXFrom="10.0" blurXTo="0.0"
  duration="2000"/>
```

The tag properties specify the starting and ending amounts of vertical and horizontal blur.

3. In the `<mx:Blur>` tag, specify the Label control called myLabel as the target of the effect (in bold):

```
<mx:Blur id="numbersBlur" target="{myLabel}"
  blurYFrom="10.0" blurYTo="0.0"
  blurXFrom="10.0" blurXTo="0.0"
  duration="2000"/>
```

You want the component called myLabel to play the effect.

4. In the `<mx:Button>` tag, specify the numbersBlur effect as the effect to play during a click event (in bold):

```
<mx:Button id="myButton" x="40" y="60" label="View"
  mouseUpEffect="{buttonGlow}" click="numbersBlur.play();"/>
```

When a user clicks the Button control, the application invokes the effect by calling the effect's play() method.

Because the numbersBlur effect targets the myLabel control, the application applies the effect to the label, not the button.

5. Hide the Label control from the user by setting its `visible` property to false in the `<mx:Label>` tag, as follows (in bold):

```
<mx:Label id="myLabel" x="40" y="100" text="4 8 15 16 23 42"
  visible="false" />
```

You don't want to display the numbers until the user clicks the View button.

6. Make the Label visible only when the user clicks the View button by programmatically setting its `visible` property to true in the button's click property, as follows (in bold):

```
<mx:Button id="myButton" x="40" y="60" label="View"
  mouseUpEffect="{buttonGlow}" click="numbersBlur.play();
  myLabel.visible='true';"/>
```

When the user clicks the button, the blur effect starts playing and the Label becomes visible.

The Behaviors.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">

  <mx:Glow id="buttonGlow" color="0x99FF66"
    alphaFrom="1.0" alphaTo="0.3" duration="1500"/>
  <mx:Blur id="numbersBlur" target="{myLabel}"
    blurYFrom="10.0" blurYTo="0.0"
    blurXFrom="10.0" blurXTo="0.0"
    duration="2000"/>
```

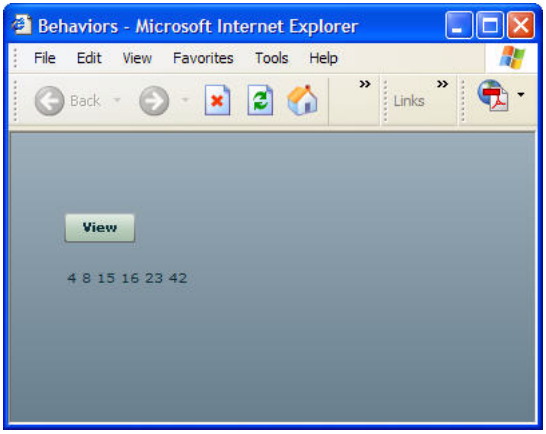
```
    <mx:Button id="myButton" x="40" y="60" label="View"
      mouseUpEffect="{buttonGlow}"
      click="numbersBlur.play(); myLabel.visible='true';"/>

    <mx:Label id="myLabel" x="40" y="100" text="4 8 15 16 23 42"
    visible="false" />

</mx:Application>
```

7. Save the file.

   Flex Builder compiles the application.

8. Click the Run button in the toolbar.

   A browser opens and runs the application. Click the View button. The button emits a green glow while a series of blurred numbers gradually comes into focus.




# Create a composite effect

When the blurry Label component comes into focus when the user clicks the View button, you would also like the component to gradually move down in the layout by 20 pixels. In other words, you would like to combine your Blur effect with a Move effect.

Flex supports combining more than one effect to create a composite effect. You define a composite effect with either the `<mx:Parallel>` or the `<mx:Sequence>` tag, depending on whether you want the combined effects of the composite effect to play in parallel or sequentially. For your application, you want the Blur and Move effects to play in parallel.

1. In the editor's Source mode, start your composite effect by entering the following tag before the <mx:Blur> tag:

   ```
   <mx:Parallel id="BlurMoveShow">
   </mx:Parallel>
   ```

   The name of your parallel composite effect is BlurMoveShow.

2. Select the full <mx:Blur> tag in your code, and then cut and paste it between the opening and closing <mx:Parallel> tags so that it becomes a child tag of the <mx:Parallel> tag.

3. Select the target="{myLabel}" property in the <mx:Blur> tag, and then cut and paste it into the opening <mx:Parallel> tag so that it becomes a property of the <mx:Parallel> tag, as follows (in bold):

   ```
   <mx:Parallel id="BlurMoveShow" target="{myLabel}">
   ```

   You want the composite effect to target the Label control called myLabel.

4. Define your new Move effect by entering the following tag after the <mx:Blur> tag:

   ```
   <mx:Move id="numbersMove" yBy="20" duration="2000" />
   ```

   You want the Label control to move 20 pixels down in 2 seconds.

   The completed <mx:Parallel> tag should look as follows:

   ```
   <mx:Parallel id="BlurMoveShow" target="{myLabel}">
     <mx:Blur id="numbersBlur"
       blurYFrom="10.0" blurYTo="0.0"
       blurXFrom="10.0" blurXTo="0.0"
       duration="2000"/>
     <mx:Move id="numbersMove" yBy="20" duration="2000" />
   </mx:Parallel>
   ```

5. In the <mx:Button> tag, change the effect to play in response to the click event by replacing the numbersBlur effect with the BlurMoveShow composite effect, as follows (in bold):

   ```
   <mx:Button id="myButton" x="40" y="60" label="View"
     mouseUpEffect="{buttonGlow}" click="BlurMoveShow.play();
     myLabel.visible='true';"/>
   ```

6. Save the file.

   Flex Builder compiles the application.

7. Click the Run button in the toolbar.

   A browser opens and runs the application. Click the View button. The button emits a green glow while a series of blurred numbers gradually comes into focus while moving 20 pixels down.

In this lesson, you learned how to use MXML to create behaviors, how to invoke an effect from a different component, and how to combine more than one effect to create a composite effect. To learn more, see Chapter 22, "Using Behaviors" in *Developing Flex Applications*.

*Public Beta 1 Public Beta 1 Public Beta 1 Public*

CHAPTER 14

# Design: Use List-based Form Controls

14

You can use list-based form controls such as a ComboBox, List, or HorizontalList in your Flex applications. After inserting this kind of control, you must populate it with items to display and values to submit for processing. In Flex, the controls are populated by data providers, which are collections of objects similar to arrays.

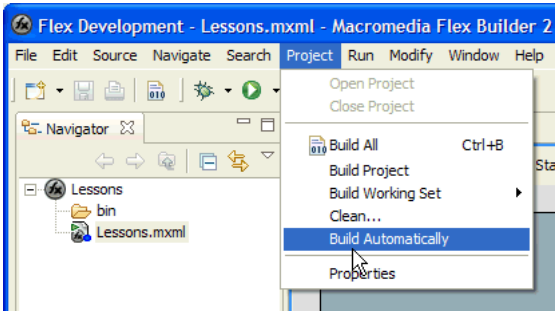
This lesson shows you how to populate list-based form controls with items to display and values to process.

In this lesson, you'll complete the following tasks:

# Set up your project

Before you begin this lesson, perform the following tasks:

- If you haven't already done so, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.




# Insert and position form controls

In this section, you create a simple form containing a ComboBox control and a submit button.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called ListControl.mxml.

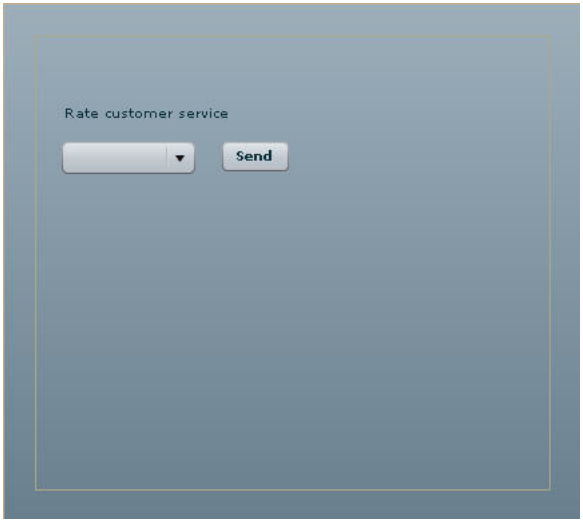
   NOTE: For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the ListControl.mxml file as the default file to be compiled by right-clicking the file in the Navigator view and selecting Application Management > Set As Default Application from the context menu.
3. In the editor's Design mode, add the following controls to the ListControl.mxml file by dragging them from the Components view (Window > Show View > Components):
   - Label
   - ComboBox
   - Button

4. Select the Label control in the layout and set the following Label properties in the Flex Properties view:
   - Text: **Rate customer service**
   - X: **20**
   - Y: **50**
5. Select the ComboBox control and set the following ComboBox properties in the Flex Properties view:
   - ID: **cbxRating**
   - X: **20**
   - Y: **80**

   The ComboBox control doesn't list any items. You populate the list later.
6. Select the Button control and set the following Button properties in the Flex Properties view:
   - Label: **Send**
   - X: **140**
   - Y: **80**

   The layout should look like the following in the editor's Design mode:



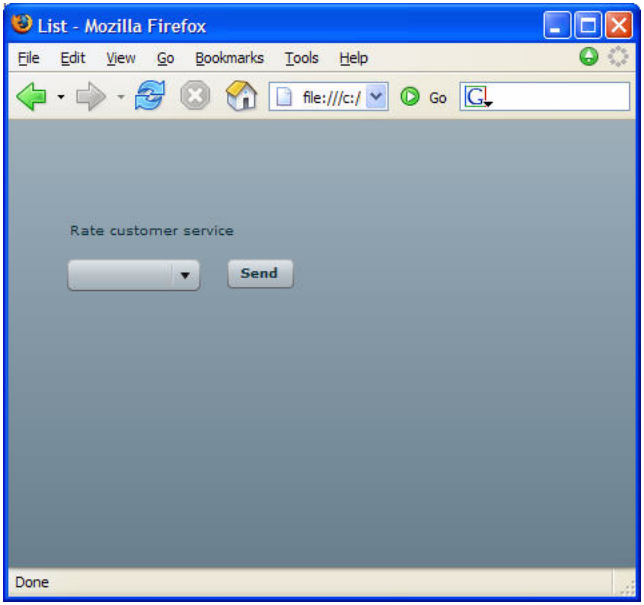
7. Switch to the editor's Source mode by clicking the Code button in the document toolbar.

   The ListControl.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">
  <mx:Label x="20" y="50" text="Rate customer service"/>
  <mx:ComboBox x="20" y="80" id="cbxRating"></mx:ComboBox>
  <mx:Button x="140" y="80" label="Send"/>
</mx:Application>
```

8. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar.

   A browser opens and runs your small Flex application.




NOTE

The browser must have Flash Player 8.5 installed to run the application. You have the option of installing this version of Flash Player in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 8.5, select Window › Preferences › General › Web Browser.

9. Click the ComboBox control in the browser.

   The control doesn't list any items because you haven't defined its data provider yet.

# Populate the list

You populate a list-based form control with the `<mx:dataProvider>` child tag. The `<mx:dataProvider>` tag lets you specify list items in several ways. The simplest method is to specify an array of strings, as follows.

1. In the editor's Source mode, enter the following code between the opening and closing `<mx:ComboBox>` tag:

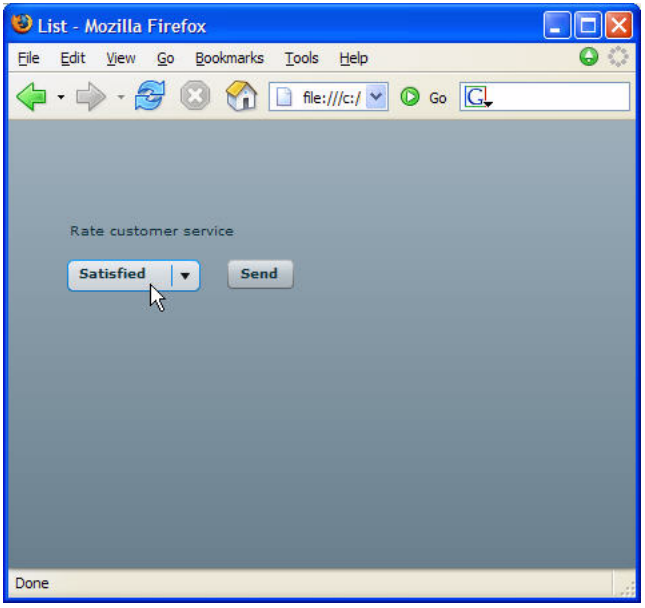
```
<mx:dataProvider>
  <mx:Array>
    <mx:String>Satisfied</mx:String>
    <mx:String>Neutral</mx:String>
    <mx:String>Dissatisfied</mx:String>
  </mx:Array>
</mx:dataProvider>
```

   The code for ListControl.mxml should look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">
  <mx:Label x="20" y="50" text="Rate customer service"/>
  <mx:ComboBox x="20" y="80" id="cbxRating">
    <mx:dataProvider>
      <mx:Array>
        <mx:String>Satisfied</mx:String>
        <mx:String>Neutral</mx:String>
        <mx:String>Dissatisfied</mx:String>
      </mx:Array>
    </mx:dataProvider>
  </mx:ComboBox>
  <mx:Button x="140" y="80" label="Send"/>
</mx:Application>
```

2. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar.

   A browser opens and runs the application.




3. Click the ComboBox control to view the list of items.

   If you want to access the value of the selected item in the ComboBox control, you can use the following expression in your code:

   ```
   cbxRating.value
   ```

   In the example, the `value` property of the ComboBox control could contain the string "Satisfied", "Neutral", or "Dissatisfied" depending on the user's selection.

4. To test the control, insert the following tag after the `<mx:Button>` tag in the ListControl.mxml file:

   ```
   <mx:Label x="20" y="140" text="{cbxRating.value}" />
   ```

   The expression inside the curly braces ({ }) is a binding expression that copies the value of the ComboBox control's `value` property, `cbxRating.value`, into the Label control's `text` property. In other words, the `text` property of the Label control is specified by the value of the selected item in the ComboBox control.

5. Save the file, wait until Flex Builder finishes compiling, and run the application.

   Select items in the ComboBox. The Label you inserted displays the string "Satisfied", "Neutral", or "Dissatisfied" depending on your selection.

# Associate values to list items

You may want to associate values with list items in a form control, as with the SELECT form element in HTML. For example, to generate reports and statistics you might want to associate the value of 5 with Satisfied, 3 with Neutral, and 1 with Dissatisfied.

To do this, you populate the ComboBox control with an array of Object components. The `<mx:Object>` tag lets you define a `label` property that contains the string to display in the ComboBox, and a `data` property that contains the data that you want to associate with the label.

1. In the editor's Source mode, replace the three `<mx:String>` tags with the following `<mx:Object>` tags:

   ```
   <mx:Object label="Satisfied" data="5"/>
   <mx:Object label="Neutral" data="3"/>
   <mx:Object label="Dissatisfied" data="1"/>
   ```

   If you want to access the value of the selected item in the ComboBox control, you can use the following expression in your code:

   ```
   cbxRating.value
   ```

   The `value` property contains the value of the selected item. When a data field is specified, the `value` property refers to the data field, not the label field. In the example, the `cbxRating.value` property could contain the values 5, 3, or 1 depending on the user's selection.

2. Save the file, wait until Flex Builder finishes compiling, and then run the application.

   Select items in the ComboBox control in the browser. The testing Label you inserted in the previous section displays the values 5, 3, or 1 depending on your selection.

To submit data for processing, you must write a click event handler for the Button control that calls a remote procedure to process the data. You can use remote-procedure-call (RPC) service components to interact with a server through web services, remote object services (Java objects), or HTTP requests. For more information, see Chapter 52, "Understanding RPC Service Components" and Chapter 53, "Using RPC Components" in *Developing Flex Applications*.

In this lesson, you inserted a list-based form control into your Flex application and provided data to it. To learn more, see Chapter 12, "Using Data-Driven Controls" in *Developing Flex Applications.*

You can also create a simple blog reader that retrieves data from an RSS feed and displays it. For instructions, see "Data: Retrieve and Display Data" on page 163

CHAPTER 15

# 15 Design: Create a Custom Component

You can use custom MXML components in your project to promote code reuse, simplify the process of building complex applications, and let other developers contribute to your project For example, you can encapsulate the layout and logic of a login box in a custom MXML component, and then use the custom component in several locations in your application.

This lesson shows you how to build an MXML component visually with Adobe Flex Builder. The lesson also shows you how to insert the new custom component visually in other MXML files.

NOTE

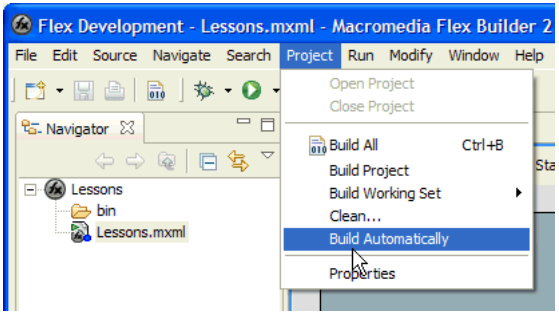
Developing the user authentication logic for a login box component is outside the scope of this lesson.

In this lesson, you'll complete the following tasks:

# Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.




# Create a test file for the custom component

You decide to build a login box as a custom MXML component. Before you start, however, you need to create an MXML application file to test it. An *MXML application file* is an MXML file that contains the `<mx:Application>` root tag. You can't compile and run an MXML component on its own; you must compile and run an MXML application file that uses the component.

In this section, you create an MXML application file to test your custom component.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Main.mxml.

| NOTE | For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project. |
|---|---|

2. Designate the Main.mxml file as the default file to be compiled by right-clicking the file in the Navigator view and selecting Application Management > Set As Default Application from the context menu.

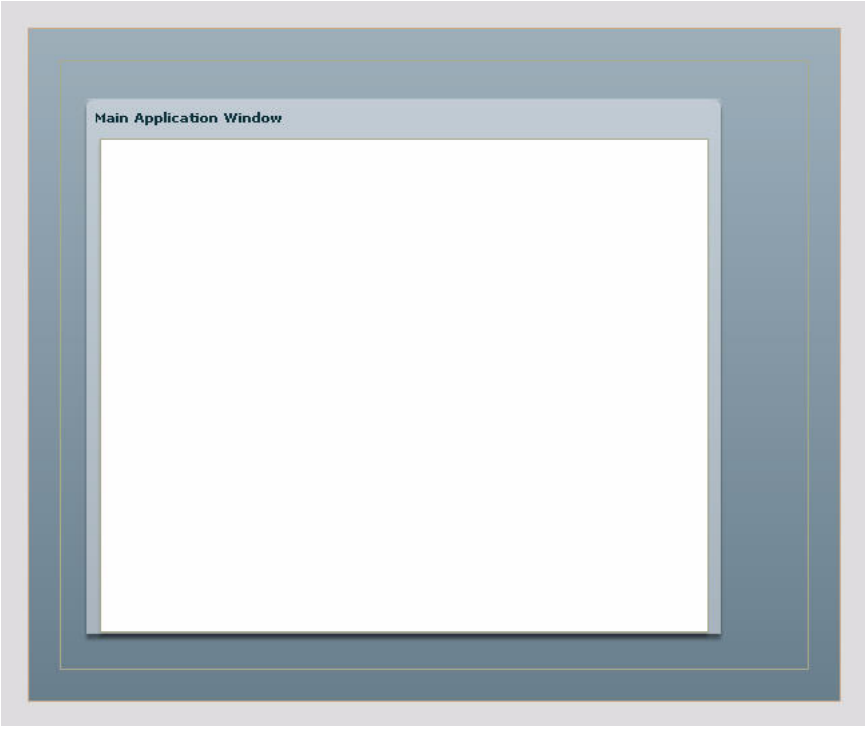
3. In the editor's Design mode, add a Panel container to the Main.mxml file by dragging it from the Components view(Window > Show View > Components).

   The Panel container is listed in the Layout category of components.

4. Select the Panel container in the Main.mxml file and set the following properties in the Flex Properties view:

   - Title: **Main Application Window**
   - Width: **475**
   - Height: **400**
   - X: **20**
   - Y: **30**

   The layout should look similar to the following:




5. Save the file.

Now you can build and test your custom component.

# Create the custom component file

The first step to building a custom MXML component is to create the file. Most custom components are derived from existing components. For your new login box component, you decide to extend the MXML Panel component.

Before you begin, create a subfolder to store the custom component files for your application.

1. In the Navigator view, right-click the Lessons parent folder and select New > Folder from the context menu.

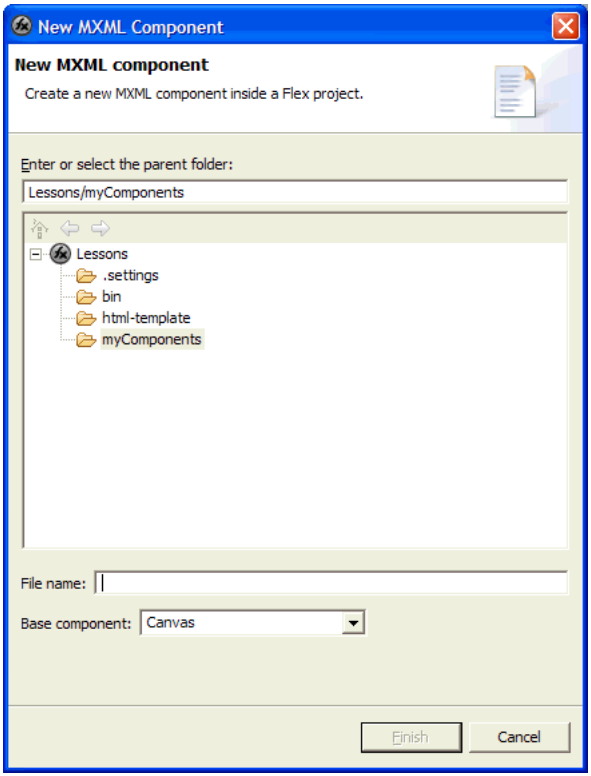
   The New Folder dialog box appears.

2. In the Folder Name text box, enter **myComponents** and click Finish.

   Flex Builder creates a new subfolder called myComponents.

3. With the myComponents folder still selected in the Navigator view, select File > New > MXML Component.

   The New MXML Component dialog box appears with the Lessons/myComponents folder set as the default folder for new custom components.




4. In the File Name text box, enter **LoginBox**.

   The filename also defines the component name.

5. In the Base Components pop-up menu, select Panel.

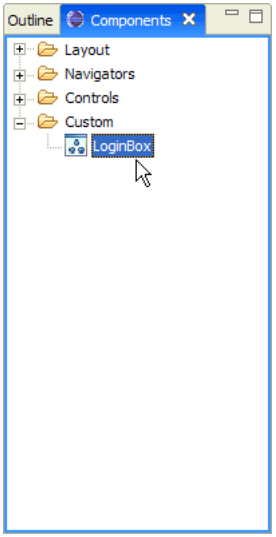
   You want to extend the Panel component.

6. In the Layout pop-up menu, make sure Absolute is selected (it should be the default).

7. Click Finish.

   Flex Builder creates and saves the LoginBox.mxml file in the myComponents folder and opens it in the MXML editor.

   If you switch to the editor's Design mode, the component also appears in the Custom category of the Components view:



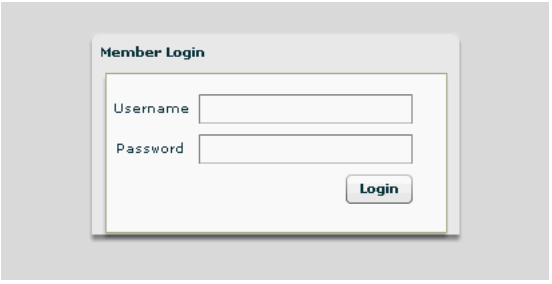

If you save a custom component file in the current project or in the classpath of the current project, Flex Builder displays the component in the Components view so that you can rapidly insert it in your applications.

| NOTE | The Components view only lists visible custom components (components that inherit from UIComponent). For more information, see the *Flex ActionScript and MXML API Reference*. |
|---|---|

# Design the layout of the custom component

The next step is to design the layout of the custom component. For your LoginBox component, you want a layout that includes username and password text boxes, and a submit button.

1. Make sure the LoginBox component is open in the editor's Design mode.
2. Select the Panel and set the following properties in the MXML Properties view:
   - Title: **Member Login**
   - Width: **275**
   - Height: **150**
3. Insert two Label controls in the panel and align them vertically.
4. Insert two TextInput controls to the right of the Label controls and align them vertically.
5. Select the first Label control and enter **Username** as the value of its Text property.
6. Select the second Label control and enter **Password** as the value of its Text property.
7. Select the first TextInput control and enter **txtUID** as the value of is ID property.
8. Select the second TextInput control and enter **txtPwd** as the value of its ID property and **True** as the value of its Is Password property.
9. Insert a Button control below the second TextInput control and enter **Login** as the value of its Label property.
10. Align and fine-tune the position of the controls so that the layout looks as follows:




Your code should look as follows (your coordinate values may vary):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute" title="Member Login" width="275" height="150">
  <mx:Label x="8" y="20" text="Username"/>
  <mx:Label x="14" y="50" text="Password"/>
```

```
    <mx:TextInput x="78" y="18" id="txtUID"/>
    <mx:TextInput x="78" y="48" id="txtPwd" password="true"/>
    <mx:Button x="188" y="78" label="Login"/>
</mx:Panel>
```

11. Save your file.

# Define an event listener for the custom component

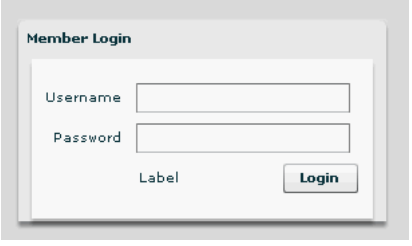
In some cases, you want the custom component to contain logic that can handle user actions. For your LoginBox component, you want the component to validate the username and password when the user clicks the Login button, and then submit the data for authentication.

This section describes how to define a simple event listener for the Login button. An event listener is also known as an event handler. For a lesson on event listeners, see Chapter 19, "Programming: Use an Event Listener," on page 185.

| NOTE | Developing the user authentication logic for the listener is outside the scope of this lesson. |
|---|---|

You also decide to modify a Label control to test that the event listener is being called properly.

1. In the editor's Design mode, insert a Label control in the space to the left of the Login button, as follows:




2. Select the Label control and enter **lblTest** as the value of the Label's ID property, and clear the value of the Text property.
3. Select the Button control and enter **handleLoginEvent**() as the value of the control's Click property.

   When the user clicks the button, you want to call the `handleLoginEvent()` function.

   Next, you write the listener function.

4. Switch to the editor's Source mode and place the insertion point immediately after the opening `<mx:Panel>` tag.
5. Start typing **<mx:Script>** until the full tag is selected in the code hints, press Enter to insert the tag in your code, and then type the closing angle bracket (>) to complete the tag.

   Flex Builder enters an `<mx:Script>` script block that also includes a CDATA construct.

   **NOTE** When using an <mx:Script> script block, you should wrap the contents in a CDATA construct. This prevents the compiler from interpreting the contents of the script block as XML, and allows the ActionScript to be properly generated.

6. Enter the following code in the CDATA construct:

```
private function handleLoginEvent():void {
  lblTest.text = "logging in...";
  //login logic
}
```

   In a real application, the `handleLoginEvent()` function would reference or contain the logic for validating and submitting the login entries for authentication. Developing the logic for the handler is outside the scope of this lesson.

   The keyword `private` sets the scope of the function: it's only available within the component. If you set the scope to public, then the function is available throughout your code.

   The keyword `void` specifies that the function returns nothing. All functions should define a return type.

   The code for the component should look as follows:

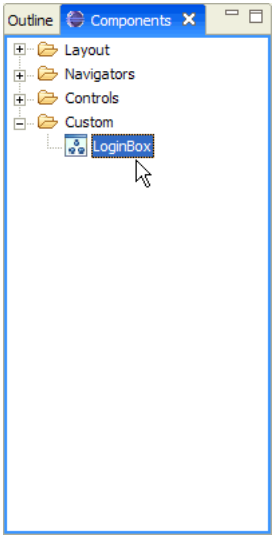
```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute" title="Member Login" width="275" height="150">
<mx:Script>
  <![CDATA[
  private function handleLoginEvent():void {
    lblTest.text = "logging in...";
    //login logic
  }
  ]]>
</mx:Script>
  <mx:Label x="8" y="20" text="Username"/>
  <mx:Label x="14" y="50" text="Password"/>
  <mx:TextInput x="78" y="18" id="txtUID"/>
  <mx:TextInput x="78" y="48" id="txtPwd" password="true"/>
  <mx:Button x="188" y="78" label="Login" click="handleLoginEvent()"/>
  <mx:Label x="78" y="78" id="lblTest"/>
</mx:Panel>
```

7. Save the file.

# Use the custom component

The next step is to add the custom component to your MXML application file, and then to compile and run the application file to test the component.

1. In the editor's Design mode, switch to the Main.mxml file.
2. Locate the LoginBox component in the Custom category of the Components view.

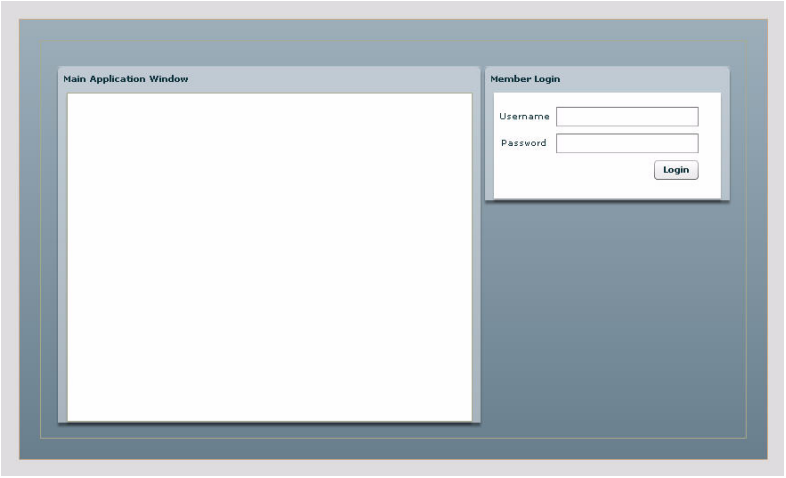



3. Drag the LoginBox component next to the right edge of the Panel in the layout.

   Flex Builder inserts and renders the custom component in your layout like any other component.
4. With the LoginBox component still selected in the layout, set the following properties in the Properties view:
   - X: **500**
   - Y: **30**

   Flex Builder displays the properties of the custom component in the Properties view like any other component.

The layout should look similar to the following:




5. Switch to the editor's Source mode by clicking the Code button in the document toolbar.

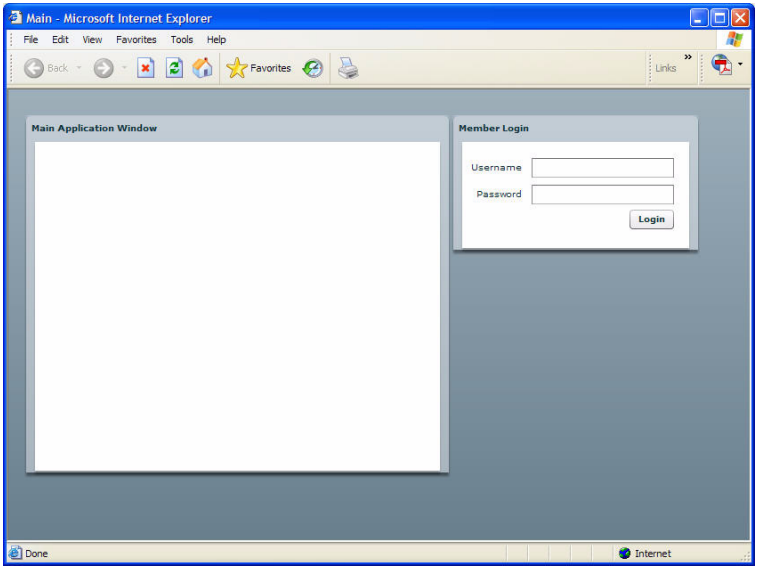
   Flex Builder inserted the following code in your file (in bold):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute" xmlns:ns1="myComponents.*">
  <mx:Panel x="20" y="30" width="475" height="400" layout="absolute"
  title="Main Application Window">
  </mx:Panel>
  <ns1:LoginBox x="500" y="30">
  </ns1:LoginBox>
</mx:Application>
```

   When you dragged the custom component into the MXML file, Flex Builder defined a new namespace called ns1, and then inserted an `<ns1:LoginBox>` tag after the `<mx:Panel>` tag.

6. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar.

   A browser opens and runs the application.




   The application displays the LoginBox component you inserted in the main application file. You can reuse the same component in multiple MXML files.

   Click the Login button to verify that the event listener is being called properly. The string "logging in..." should appear to the left of the Login button.

In this lesson, you created a custom MXML component visually, and then used it in an MXML application file. You designed the component's layout and defined an event listener for a control in the component. To learn more, see Chapter 10, "Creating Custom MXML Components," on page 159 and Chapter 7, "Creating Simple MXML Components" in *Creating and Extending Flex Components*.

CHAPTER 17

# Data: Retrieve and Display Data

17

To provide data to your application, Flex includes components designed specifically for interacting with HTTP servers, web services, or remote object services (Java objects). These components are called remote procedure call (RPC) service components.

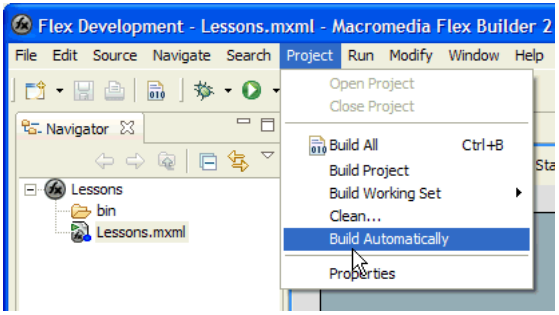
In this lesson, you create a simple blog reader that retrieves recent posts and lets users read the first few lines of the posts. You use an HTTPService component to retrieve data from an RSS feed, and then you bind the data to a Label, DataGrid, TextArea, and Link control.

In this lesson, you'll complete the following tasks:

# Set up your project

Before you begin this lesson, perform the following tasks:

- If you have not already done so, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.




# Review your access to remote data sources

For security reasons, applications running in Flash Player on a client computer can only access remote data sources if one of the following conditions is met:

- Your application's compiled SWF file is in the same domain as the remote data source.
- You use a proxy and your SWF is on the same server as the proxy.

  Flex Enterprise Services provides a complete proxy management system for Flex applications. You can also create a simple proxy service using a web scripting language such as ColdFusion, JSP, PHP, or ASP. For more information on creating your own proxy, see the following TechNote on the Macromedia website at www.macromedia.com/go/16520#proxy.
- A crossdomain.xml (cross-domain policy) file is installed on the web server hosting the data source.

  The crossdomain.xml file permits SWFs in other domains to access the data source. For more information on configuring crossdomain.xml files, see the following TechNote on the Macromedia website at www.macromedia.com/go/14213.

The data sources used in this lesson are located in a domain that has a crossdomain.xml setup. Therefore, Flash Player can access the remote data.

# Insert and position the blog reader controls

In this section, you create the layout of your blog-reader application.

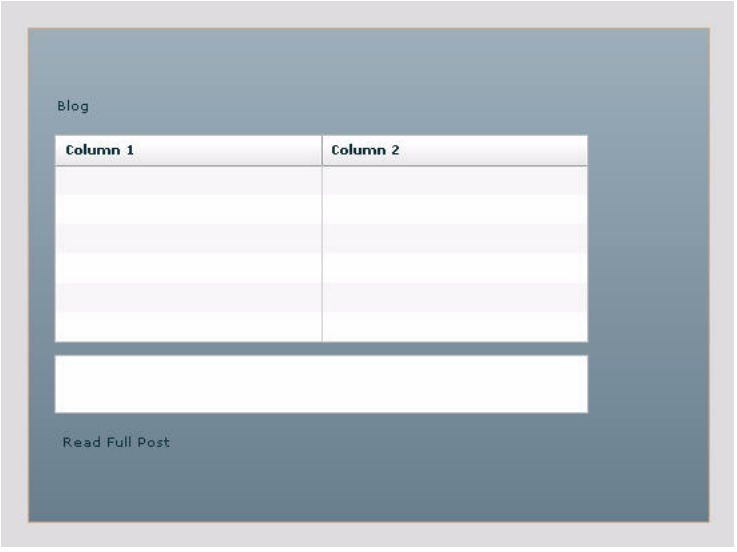
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called BlogReader.mxml.

   NOTE: For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the BlogReader.mxml file as the default file to be compiled by right-clicking the file in the Navigator view and selecting Application Management > Set As Default Application from the context menu.
3. In the editor's Design mode, add the following controls to the BlogReader.mxml file by dragging them from the Components view (Window > Show View > Components):
   - Label
   - DataGrid
   - TextArea
   - Link
4. Use the mouse to arrange the controls in the layout in a vertical, left-aligned column.
5. Select the Label control and set the following properties in the Flex Properties view:
   - Text: **Blog**
   - X: **20**
   - Y: **50**
6. Select the DataGrid control and set the following properties:
   - Id: **dgPosts**
   - X: **20**
   - Y: **80**
   - Width: **400**

7. Select the TextArea control and set the following properties:
   - X: **20**
   - Y: **245**
   - Width: **400**
8. Select the Link control and set the following properties:
   - Label: **Read Full Post**
   - X: **20**
   - Y: **300**

   The layout should look like the following:



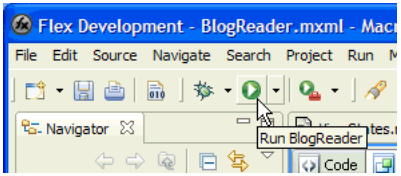

9. Switch to the editor's Source mode by clicking the Code button in the document toolbar.

   The BlogReader.mxml file should contain the following MXML code:

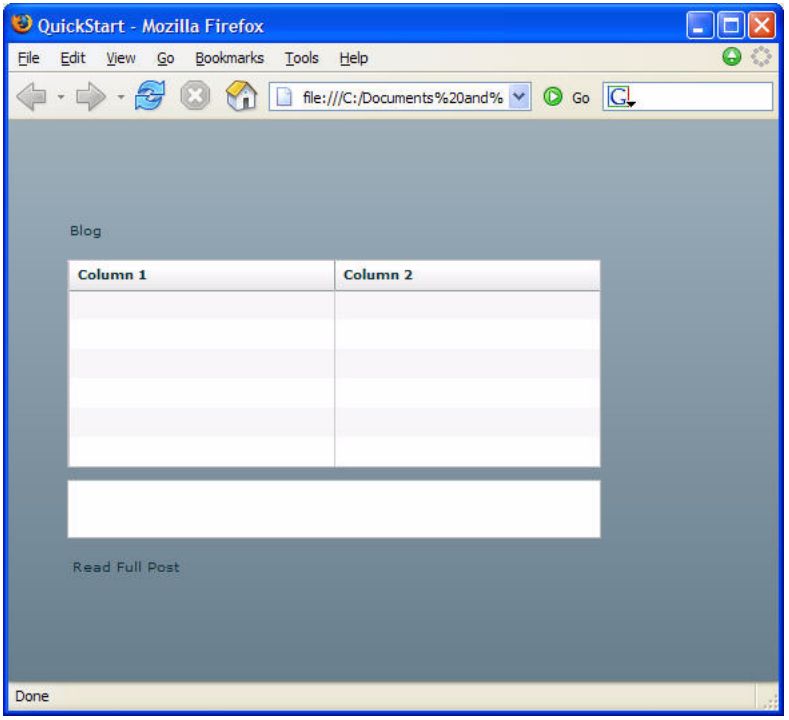
```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">
  <mx:Label x="20" y="50" text="Blog"/>
  <mx:DataGrid x="20" y="80" id="dgPosts" width="400">
    <mx:columns>
      <mx:DataGridColumn headerText="Column 1" columnName="col1"/>
      <mx:DataGridColumn headerText="Column 2" columnName="col2"/>
    </mx:columns>
  </mx:DataGrid>
  <mx:TextArea x="20" y="245" width="400" />
```

```
    <mx:Link x="20" y="300" label="Read Full Post" />
</mx:Application>
```

10. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar to start the application.




A browser opens and runs the application.




NOTE

The browser must have Flash Player 8.5 installed to run the application. You have the option of installing this version of Flash Player in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 8.5, select Window › Preferences › General › Web Browser.

The application doesn't display any blog information yet.

The next step is to retrieve information about recent blog posts. You can use an RPC service component called HTTPService to accomplish this task.

# Insert a HTTPService component

For the blog reader in this lesson, you retrieve posts from Matt Chotin's blog at http://weblogs.macromedia.com/mchotin/ on the Macromedia website. Matt is a Principal Engineer on the Flex team and writes about Flex in his blog.

You can use the HTTPService component to access the blog's XML feed and retrieve information about recent posts. The component lets you send an HTTP GET or POST request, and then retrieve the data returned in the response.

1. In the editor's Source mode, enter the following `<mx:HTTPService>` tag immediately after the opening `<mx:Application>` tag:

   ```
   <mx:HTTPService
     id="feedRequest"
     url="http://weblogs.macromedia.com/mchotin/index.xml"
     useProxy="false"/>
   ```

   The `url` property specifies the location of the requested file, in this case the RSS feed of Matt Chotin's blog. You can find the link on the right-hand side of his blog.

   The `useProxy` property specifies that you don't want to use a proxy on a server. The weblogs.macromedia.com domain where Matt's blog is located has a crossdomain.xml setup, so Flash Player can access the remote data sources on this server, including RSS feeds. For more information, see "Review your access to remote data sources" on page 164.

   The next step is to prompt the application to send a request to the specified URL. You decide to send the request automatically whenever the application starts, as follows.

2. In the `<mx:Application>` tag, add the following `creationComplete` property (in bold):

   ```
   <mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
     layout="absolute" creationComplete="feedRequest.send()" >
   ```

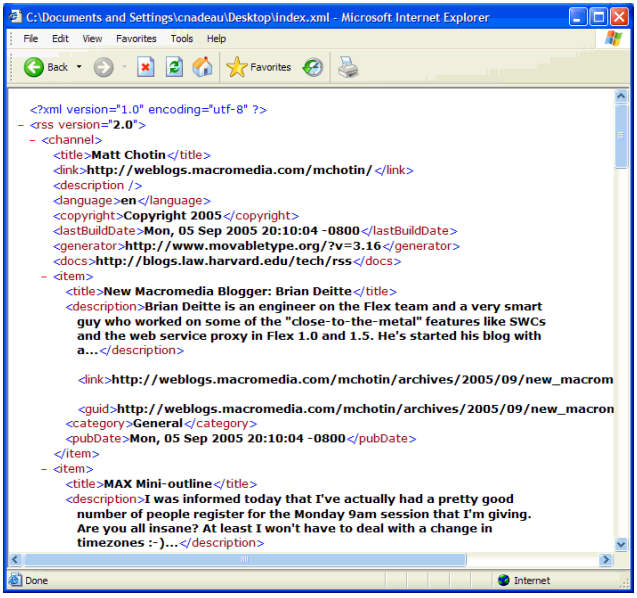
   When your application is finished starting up, the HTTPService component's `send()` method is called. The method makes an HTTP GET or POST request to the specified URL, and an HTTP response is returned. In this case, the RSS feed returns XML data.

   Next, you want to check if the application is retrieving the RSS feed successfully. You can do this by binding data to the Label control, as follows.

3. In the `<mx:Label>` tag, replace the value of the `text` property ("Blog") with the following binding expression (in bold):

```
<mx:Label x="20" y="50" text="{feedRequest.result.rss.channel.title}" />
```

This expression binds the title field to the Label control. The expression reflects the structure of the XML. When XML is returned to a HTTPService component, the component parses it into an ActionScript object named result. The structure of the result object mirrors the structure of the XML document. To check the XML structure, download the RSS feed's XML file (http://weblogs.macromedia.com/mchotin/index.xml) and open it in Internet Explorer.




The general structure of the XML is as follows:

```
<rss>
  <channel>
    <item>
    ...
```

Each node has child nodes containing data, including the "title" child node of the channel node. The result object of the HTTPService component (`feedRequest.result`) reflects this structure:

```
feedRequest.result.rss.channel.title
```

Your code should look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute" creationComplete="feedRequest.send()" >

  <mx:HTTPService
    id="feedRequest"
    url="http://weblogs.macromedia.com/mchotin/index.xml"
    useProxy="false"/>

  <mx:Label x="20" y="50" text="{feedRequest.result.rss.channel.title}"/
  >
  <mx:DataGrid x="20" y="80" id="dgPosts" width="400">
    <mx:columns>
      <mx:DataGridColumn headerText="Column 1" columnName="col1"/>
      <mx:DataGridColumn headerText="Column 2" columnName="col2"/>
    </mx:columns>
  </mx:DataGrid>
  <mx:TextArea x="20" y="245" width="400" />
  <mx:Link x="20" y="300" label="Read Full Post" />
</mx:Application>
```

4. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar to test the application.

   A browser opens and runs the application. The blog's title, Matt Chotin, should appear in the Label control, indicating that the application successfully retrieved data from the RSS feed.

| NOTE | There may be a few seconds delay before the title appears while the application is contacting the server. |
|---|---|

# Populate a DataGrid control

Use the DataGrid control to display the titles of recent posts and the dates they were posted.

1. In the editor's Source mode, enter the following `dataProvider` property in the `<mx:DataGrid>` tag (in bold):

   ```
   <mx:DataGrid x="20" y="80" id="dgPosts" width="400"
     dataProvider="{feedRequest.result.rss.channel.item}" >
   ```

   You want the XML node named item to provide data to the DataGrid control. This node is repeated in the XML, so it will be repeated in the DataGrid.

2. In the first `<mx:DataGridColumn>` tag, enter the following `headerText` and `columnName` property values (in bold):

   ```
   <mx:DataGridColumn headerText="Posts" columnName="title" />
   ```

   You want the first column in the DataGrid control to display the titles of the posts. You do this by identifying the field in the XML that contains the title data, and then entering this field as the value of the `columnName` property. In the XML node specified in the `dataProvider` property (item), the child node called title contains the information you want.

3. In the second `<mx:DataGridColumn>` tag, enter the following `headerText`, `columnName` and `width` property values (in bold):

   ```
   <mx:DataGridColumn headerText="Date" columnName="pubDate" width="150" />
   ```

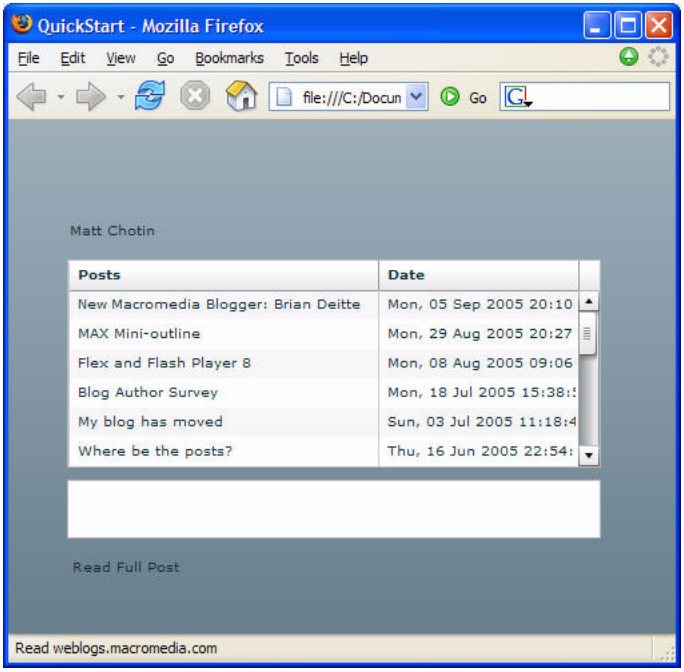
   You want the second column in the DataGrid to display the dates of the posts. In this case, the field that contains the data is called `pubDate`.

   The `<mx:DataGrid>` tag should look as follows:

   ```
   <mx:DataGrid x="20" y="80" id="dgPosts" width="400"
     dataProvider="{feedRequest.result.rss.channel.item}">
     <mx:columns>
       <mx:DataGridColumn headerText="Posts" columnName="title" />
       <mx:DataGridColumn headerText="Date" columnName="pubDate"
     width="150" />
     </mx:columns>
   </mx:DataGrid>
   ```

4. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

   A browser opens and runs the application.




   Blog titles and dates should appear in the DataGrid control, confirming that the application successfully retrieved data from the RSS feed and populated the control.

# Display a selected item

When the user selects a post in the DataGrid control, you want the application to display the first few lines of the post in the TextArea control. In the item node of the XML feed providing data to the DataGrid control, this information is contained in a field called description.

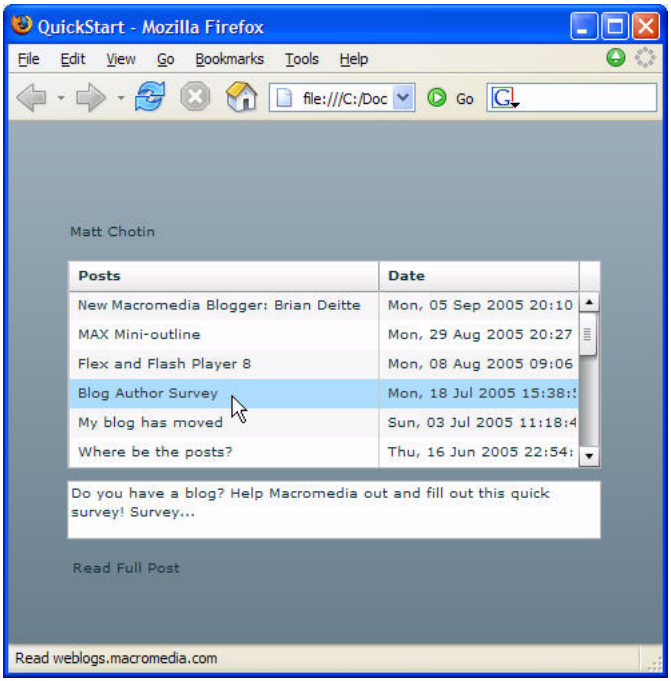
1. In the editor's Source mode, enter the following `htmlText` propertyin the `<mx:TextArea>` tag (in bold):

```
<mx:TextArea x="20" y="245" width="400"
  htmlText="{dgPosts.selectedItem.description}" />
```

   For each selected item in the DataGrid component (named dgPosts), the value of the description field is used for the value of the `htmlText` property. The `htmlText` property lets you display HTML formatted text.

2. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

   A browser opens and runs the application. Click items in the DataGrid control. The first few lines of each post should appear in the TextArea control.




# Create a dynamic link

The RSS feed doesn't provide the full text of the posts, but you still want users to be able to read the posts if they're interested. While the RSS feed doesn't provide the information, it does provide the URLs to individual posts. In the item node of the XML feed, this information is contained in a field called link.

You decide to create a dynamic link that opens a browser and displays the full content of the post selected in the DataGrid.

1. In the editor's Source mode, enter the following `click` propertyin the `<mx:Link>` tag (in bold):

```
<mx:Link x="20" y="300" label="Read Full Post"
  click="navigateToURL(new URLRequest(dgPosts.selectedItem.link));" />
```

   The value of the link field of the selected item in the DataGrid control, `dgPosts.selectedItem.link`, is specified in the argument to the `navigateToURL()` method, which is called when the user clicks the Link control. The `navigateToURL()` method loads a document from the specified URL in a new browser window.

| NOTE | The navigateToURL() method takes a URLRequest object as an argument, which in turn takes a URL string as an argument. |
|---|---|

2. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button.

   A browser opens and runs the application. Click an item in the DataGrid control and then click the Read Full Post link. A new browser window should open and display the blog page with the full post.


In this lesson, you used an HTTPService component to retrieve data from an RSS feed, and then you bound the data to a Label, DataGrid, TextArea, and Link control. To learn more, see the following topics in *Developing Flex Applications:*

- Chapter 52, "Understanding RPC Service Components"
- Chapter 53, "Using RPC Components"

CHAPTER 18

# Data: Use Web Services

18

To provide data to your application, Flex includes components designed specifically for interacting with web services, HTTP servers, or remote object services (Java objects). These components are called remote procedure call (RPC) service components.

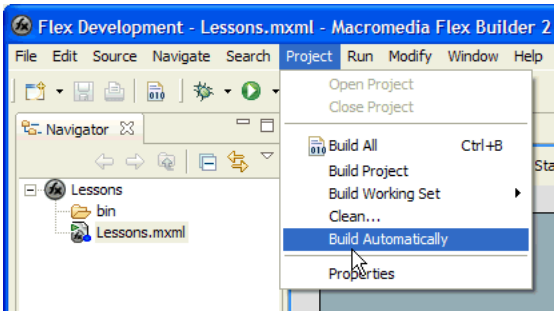
In this lesson, you create a simple reporting application for a blog aggregator that lists the most popular posts in the last 30 days. You decide to let the users determine the number of posts to list. You use a WebService component to retrieve the data from a SOAP-based web service provided by the blog aggregator site, and then you display the data in a DataGrid control.

In this lesson, you'll complete the following tasks:

# Set up your project

Before you begin this lesson, perform the following tasks:

- If you have not already done so, create the QuickStart project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.




# Review your access to remote data sources

For security reasons, applications running in Flash Player on client computers can only access remote data sources if one of the following conditions is met:

- Your application's compiled SWF file is in the same domain as the remote data source.
- You use a proxy and your SWF is on the same server as the proxy.

  Flex Enterprise Services provides a complete proxy management system for Flex applications. You can also create a simple proxy service using a web scripting language such as ColdFusion, JSP, PHP, or ASP. For more information on creating your own proxy, see the following TechNote on the Macromedia website at www.macromedia.com/go/16520#proxy.
- A crossdomain.xml (cross-domain policy) file is installed on the web server hosting the data source.

  The crossdomain.xml file permits SWFs in other domains to access the data source. For more information on configuring crossdomain.xml files, see the following TechNote on the Macromedia website at www.macromedia.com/go/14213.

The data sources used in this lesson are located in a domain that has a crossdomain.xml setup. Therefore, Flash Player can access the remote data.

# Review the API documentation

The MXNA blog aggregator provides a number of web services for developers at http://weblogs.macromedia.com/mxna/Developers.cfm. Before you start building your application, you should review the API documentation for their web services to make sure a method exists that can retrieve the information you want. The API documentation for the web services is located at http://weblogs.macromedia.com/mxna/webservices/mxna2.html.

The documentation describes a method called `getMostPopularPosts`. The method returns a number of posts with the most clicks in the last 30 days. The number of posts returned cannot exceed 50. For each post returned, the following information is provided: postId, clicks, dateTimeAggregated, feedId, feedName, postTitle, postExcerpt, postLink.

The method takes two required numeric parameters:

- **daysBack** specifies the number of days you want to go back.
- **limit** specifies the total number of posts you want returned.

With this information, you can use a Flex component called WebService to consume this web service and retrieve the data you want—a list of the most popular posts in the last 30 days.

# Insert and position controls

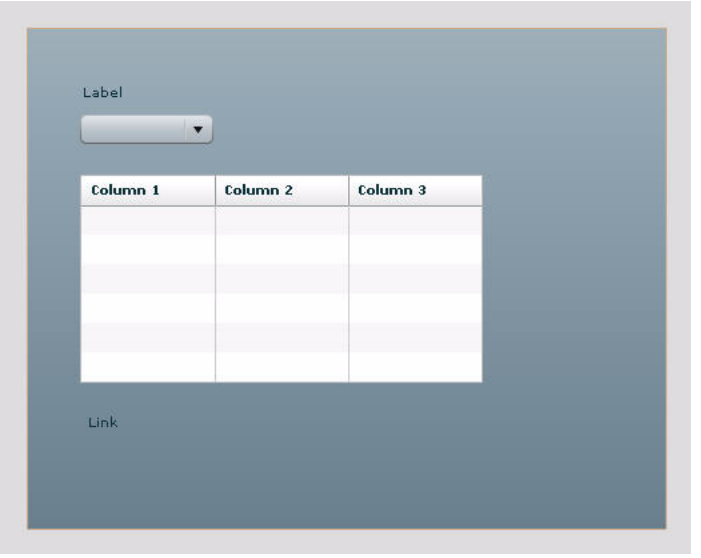
In this section, you create the layout of your reporting application. You decide to use a ComboBox control to let the users set the number of top posts to list, and a DataGrid to display the top posts.

1. With your QuickStart project selected in the Navigator view, select File > New > MXML Application and create an application file called Services.mxml.

   NOTE: For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Services.mxml file as the default file to be compiled by right-clicking the file in the Navigator view and selecting Application Management > Set As Default Application from the context menu.

3. Switch to the editor's Design mode by clicking the Design button in the document toolbar, and then drag the following controls to the Services.mxml file from the Components view:
   - Label
   - ComboBox
   - DataGrid
   - Link
4. Use the mouse to arrange the controls on the Canvas in a vertical, left-aligned column similar to the following:



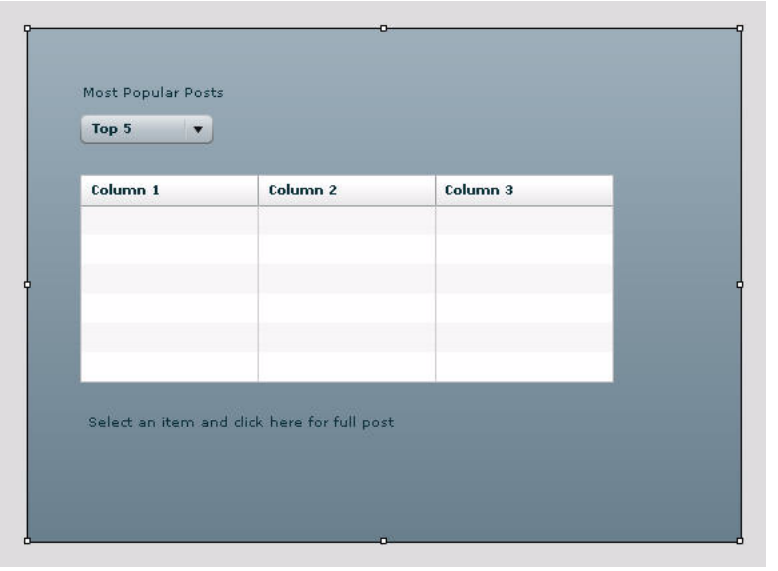

5. Select the Label control and enter **Most Popular Posts** as the value of its Text property in the Flex Properties view.
6. Select the ComboBox control and enter **cbxNumPosts** as the value of its ID property.

   The ComboBox control doesn't list any items. You populate the list next.
7. Switch to the editor's Source mode by clicking the Code button in the document toolbar, and then enter the following code between the opening and closing `<mx:ComboBox>` tag:

```
<mx:Object label="Top 5" data="5"/>
<mx:Object label="Top 10" data="10"/>
<mx:Object label="Top 15" data="15"/>
```

8. Switch back to the editor's Design mode, select the DataGrid component, and specify the following properties in the Flex Properties view:
   - ID: **dgTopPosts**
   - Width: **400**
9. Select the Link control and enter **Select an item and click here for full post** as the value of its Label property.

   The layout should look like the following:




10. Switch to the editor's Source mode.

    The Services.mxml file should contain the following MXML code (your coordinate values may vary):

```
<?xml version="1.0" encoding="utf-8"?>
mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">
  <mx:Label x="40" y="40" text="Most Popular Posts"/>
  <mx:ComboBox x="40" y="65" id="cbxNumPosts">
    <mx:Object label="Top 5" data="5"/>
    <mx:Object label="Top 10" data="10"/>
    <mx:Object label="Top 15" data="15"/>
  </mx:ComboBox>
  <mx:DataGrid x="40" y="110" id="dgTopPosts" width="400">
    <mx:columns>
      <mx:DataGridColumn headerText="Column 1" columnName="col1"/>
      <mx:DataGridColumn headerText="Column 2" columnName="col2"/>
```

```
        <mx:DataGridColumn headerText="Column 3" columnName="col3"/>
      </mx:columns>
    </mx:DataGrid>
    <mx:Link x="40" y="285" label="Select an item and click here for full
    post"/>

</mx:Application>
```

The next step is to insert and configure a Flex component called WebService to your application.

# Insert a WebService component

You use the Flex WebService component to access a SOAP-based web service and retrieve information about recent blog posts.

1. In the editor's Source mode, enter the following `<mx:WebService>` tag immediately after the opening `<mx:Application>` tag:

   ```
   <mx:WebService id="wsBlogAggr"
     wsdl="http://weblogs.macromedia.com/mxna/webservices/mxna2.cfc?wsdl"
     useProxy="false">
   </mx:WebService>
   ```

   The `wsdl` property specifies the location of the WSDL file for the web service. This information is found on the developers page at http://weblogs.macromedia.com/mxna/Developers.cfm#web.

   The `useProxy` property specifies that you don't want to use a proxy on a server. For more information, see "Review your access to remote data sources" on page 176.

2. Specify the parameters to pass to the web service method.

   According to the API documentation, the `getMostPopularPosts` method takes the following required parameters:

   - **daysBack** specifies the number of days you want to go back.
   - **limit** specifies the total number of rows you want returned.

   To specify these parameters, enter the following tags between the opening and closing `<mx:WebService>` tags:

   ```
   <mx:operation name="getMostPopularPosts">
     <mx:request>
       <daysBack>30</daysBack>
       <limit>{cbxNumPosts.value}</limit>
     </mx:request>
   </mx:operation>
   ```

   The `name` property of an `<mx:operation>` tag must match the web service method name.

You use a constant for the value of the `daysBack` parameter, but you bind the value of the `limit` parameter to the value of the selected item in the cbxNumPosts ComboBox control. You want the user to specify the number of posts to list.

The next step is to prompt the application to call the web service method. You decide the method should be called when the ComboBox control changes in response to the user selecting an option.

3. In the `<mx:ComboBox>` tag, add the following `change` property (in bold):

```
<mx:ComboBox x="40" y="76" id="cbxNumPosts"
  change="wsBlogAggr.getMostPopularPosts.send()">
```

When the user selects an option in the ComboBox control, the `getMostPopularPosts` method of the wsBlogAggr WebService component is called. The method's parameters are specified in the WebService component's `<mx:operation>` tag. The `limit` parameter is set at runtime depending on the option the user selects.

The application is ready to call the web service. The next step is to display the data returned by the web service.

# Populate the DataGrid component

You want to use the DataGrid control to display the data returned by the web service. Specically, you want to display the titles of the most popular posts and the number of clicks each has received.

1. In the editor's Source mode, enter the following `dataProvider` property in the `<mx:DataGrid>` tag (in bold):

```
<mx:DataGrid x="40" y="126" id="dgTopPosts" width="400"
  dataProvider="{wsBlogAggr.getMostPopularPosts.result}">
```

You want to display the results of the web service's getMostPopularPosts operation in the DataGrid control.

2. In the first `<mx:DataGridColumn>` tag, enter the following `headerText` and `columnName` property values (in bold):

```
<mx:DataGridColumn headerText="Top Posts" columnName="postTitle" />
```

You want the first column in the DataGrid control to display the titles of the posts. You do this by identifying the field returned by the web service operation that contains the title data, and then entering the field name as the value of the `columnName` property. According to the API documentation for the `getMostPopularPosts` method, the field called `postTitle` contains the information you want.

3. In the second `<mx:DataGridColumn>` tag, enter the following `headerText`, `columnName`, and `width` property values (in bold):

```
<mx:DataGridColumn headerText="Clicks" columnName="clicks" width="75" />
```

You want the second column in the DataGrid control to display the number of clicks for each post during the last 30 days. According to the API documentation, the field that contains the data is called `clicks`.

4. Delete the third `<mx:DataGridColumn>` tag.

You don't need a third column.

The `<mx:DataGrid>` tag should look as follows:

```
<mx:DataGrid x="40" y="110" id="dgTopPosts" width="400"
  dataProvider="{wsBlogAggr.getMostPopularPosts.result}">
  <mx:columns>
    <mx:DataGridColumn headerText="Top Posts" columnName="postTitle"/>
    <mx:DataGridColumn headerText="Clicks" columnName="clicks"
  width="75"/>
  </mx:columns>
</mx:DataGrid>
```

5. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar to test the application.

A browser opens and runs the application. You find a problem in the application's default state: The ComboBox reads Top 5 but the DataGrid does not display any information.

The DataGrid should display the top five posts, but it doesn't because your application hasn't called the web service yet. The application only calls it when the ComboBox changes. Even if you click Top 5 in the ComboBox after the application starts, the call is still not made because the selected item hasn't changed.

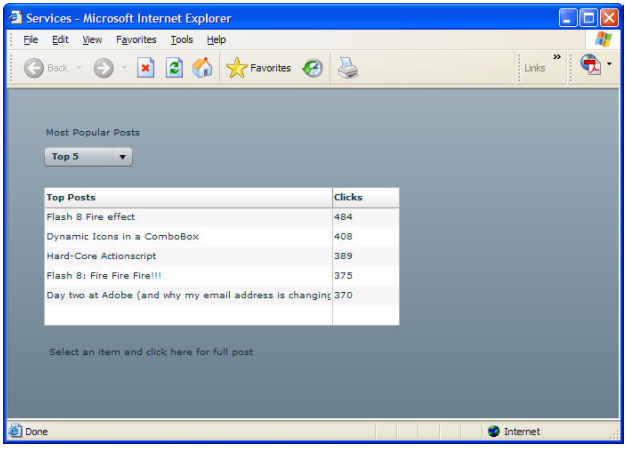
To fix the problem, you decide to also call the web service immediately after the application is created, as follows.

6. In the editor's Source mode, enter the following `creationComplete` property in the opening `<mx:Application>` tag (in bold):

```
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute"
  creationComplete="wsBlogAggr.getMostPopularPosts.send()">
```

7. Save the file and run the application.

   Blog titles and click statistics should appear in the DataGrid control after the application starts, confirming that the application successfully retrieved data from the web service and populated the control.




NOTE

There may be a few seconds delay before the data appears while the application is contacting the server.

Select another option from the ComboBox control to display a longer list of posts.

# Create a dynamic link

The web service doesn't provide the full text of the posts, but you still want users to be able to read the posts if they're interested. While the web service doesn't provide the information, it does provide the URLs to individual posts. According to the API documentation for the `getMostPopularPosts` method (see "Review the API documentation" on page 177), the information is contained in a field called `postLink`.

You decide to create a dynamic link that opens a browser and displays the full content of the post selected in the DataGrid control.

1. In the editor's Source mode, enter the following `click` property in the `<mx:Link>` tag (in bold):

```
<mx:Link x="40" y="285" label="Select an item and click here for full
  post"
  click="navigateToURL(new
  URLRequest(dgTopPosts.selectedItem.postLink));" />
```

The value of the link field of the selected item in the DataGrid control, `dgTopPosts.selectedItem.postLink`, is specified in the argument to the `navigateToURL()` method, which is called when the user clicks the Link control. The `navigateToURL()` method loads a document from the specified URL in a new browser window.

| NOTE | The navigateToURL() method takes a URLRequest object as an argument, which in turn takes a URL string as an argument. |
|---|---|

2. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button.

   A browser opens and runs the application. Click an item in the DataGrid control and then click the Link control. A new browser window should open and display the blog page with the full post.

In this lesson, you used a WebService component to call and pass method parameters to a SOAP-based web service. You then bound the data returned by the web service to a DataGrid and a Link control. To learn more, see the following topics in *Developing Flex Applications:*

- Chapter 52, "Understanding RPC Service Components"
- "Explicit parameter passing with RemoteObject and WebService components"
- "Parameter binding with WebService components"
- "Setting properties for RemoteObject methods or WebService operations"
- "Handling service results"
- "Using features specific to WebService components"

CHAPTER 19

# Programming: Use an Event Listener

19

When developing Flex applications, event handling is one of the most basic and important tasks.

Events let you know when something happens within a Flex application. They can be generated by user devices, such as the mouse and keyboard, or other external input, such as the return of a web service call. Events are also triggered when changes happen in the appearance or life cycle of a component, such as the creation or destruction of a component or when the component is resized.

You can respond to these events in your code by using *event listeners*. Event listeners are the functions or class methods that you write to respond to specific events. They are also referred to as event handlers.

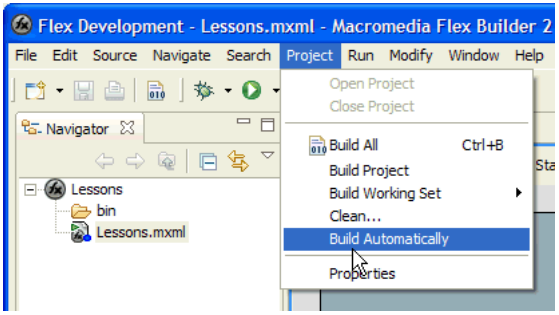
This lesson shows you how to use an event listener. It shows you how to write one for a Button control, and then how to tie the event listener to the Button's click event by using two different methods.

In this lesson, you'll complete the following tasks:

# Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Flex Builder. See "Basic: Create a Project" on page 95.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.




# Create a simple user interface

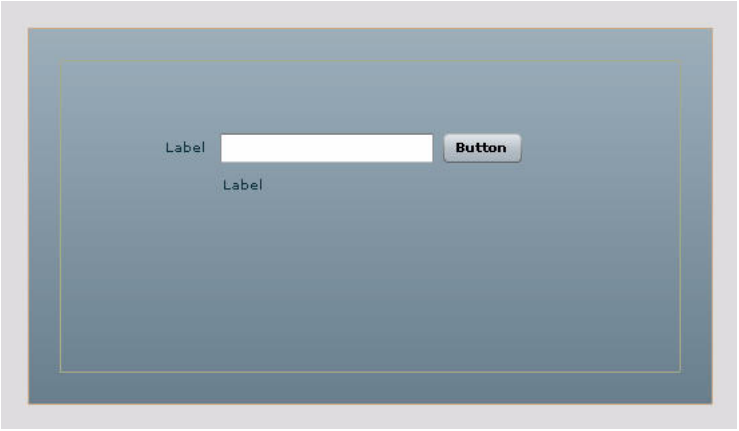
You decide to build a simple currency converter for your online store. You want the user to be able to specify a dollar amount and click a button to get the equivalent amount in yen. The first step is to design a simple user interface.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Events.mxml.

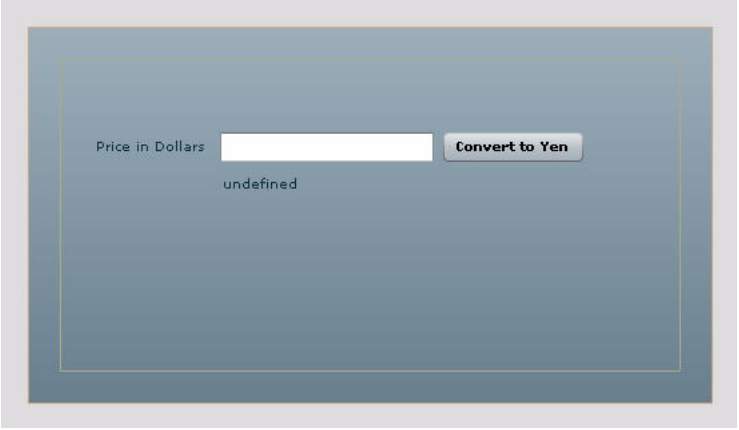
   NOTE: For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Events.mxml file as the default file to be compiled by right-clicking the file in the Navigator view and selecting Application Management > Set As Default Application from the context menu.
3. In the editor's Design mode, add two Label controls, a TextInput control, and a Button control to the Events.mxml file by dragging them from the Components view (Window > Show View > Components).

4. Arrange the controls so that they roughly match the following illustration:




5. Select the first Label control and enter **Price in Dollars** as the value of its Text property in the Flex Properties view.
6. Select the TextInput control and enter **txtPrice** as the value of its ID property.
7. Select the Button control and set the following properties:
   - ID: **btnConvert**
   - Label: **Convert to Yen**
8. Select the second Label control (below the TextInput control) and do the following:
   - Clear the value of its Text property.
   - Enter **lblResults** as the value of its ID property.
9. Fine-tune the position of the controls so that the layout looks as follows:



**10.** Switch to the editor's Source modeand examine the code generated by Flex Builder.

Your code should look as follows (your coordinate values may vary):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute">
  <mx:Label x="31" y="60" text="Price in Dollars"/>
  <mx:Label x="126" y="88" id="lblResults"/>
  <mx:TextInput x="126" y="58" id="txtPrice"/>
  <mx:Button x="294" y="58" label="Convert to Yen" id="btnConvert"/>
</mx:Application>
```

**11.** Save your file.

# Write an event listener

Next, you write an event listener for the Convert to Yen button. You want the listener to consist of an ActionScript function that can calculate and display a specified dollar price in Yen.

**1.** Switch to the editor's Source mode and place the insertion point immediately after the opening `<mx:Application>` tag.

**2.** Start typing **<mx:Script>** until the full tag is selected in the code hints, hit Enter to insert the tag in your code, and then type the closing angle bracket (>) to complete the tag.

Flex Builder enters an `<mx:Script>` script block that also includes a CDATA construct.

| NOTE | When using an `<mx:Script>` script block, you should wrap the contents in a CDATA construct. This prevents the compiler from interpreting the contents of the script block as XML, and allows the ActionScript to be properly generated. |
|---|---|

**3.** Enter or paste the following code in the CDATA construct:

```
public function convertCurrency():void {
  var rate:Number = 120;
  var price:Number = Number(txtPrice.text);
  if (isNaN(price)) {
    lblResults.text = "Please enter a valid price.";
  } else {
    price = price * rate;
    lblResults.text = "Price in Yen: " + String(price);
  }
}
```

The keyword `public` sets the scope of the function. It's available throughout your code.

The keyword `void` specifies that the function returns nothing. All functions should define a return type.

The price entered by the user, `txtPrice.text`, is cast as a Number and then validated to make sure the user entered a number. If the price is a number, the calculation is performed and the result is cast back to a String for display in the lblResults control.

In a real application, the value of the rate variable would be set at runtime by calling a web service and retrieving the current exchange rate. As well, the result would be formatted as a currency.

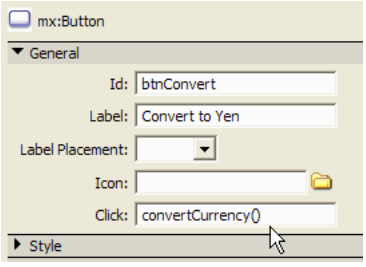
4. Save the file.

# Register the event listener with MXML

After writing the event listener, you want to call it when the user clicks the Convert to Yen button. When called, the listener performs the currency calculation and displays the results. One way to register the listener is to specify it as the value of the `click` property in the `<mx:Button>` tag.

You can also use ActionScript to register the listener with the Button control, as described in the section, "Register the event listener with ActionScript" on page 190.

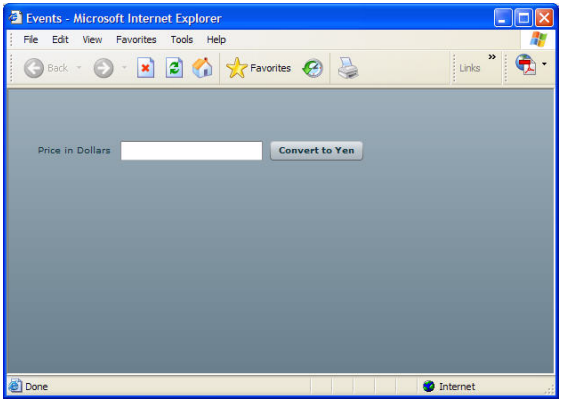
1. In the editor's Design mode, select the Button control and enter **convertCurrency**() as the value of the control's Click property.




When the user clicks the button, the button calls the `convertCurrency()` function.

2. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar.

   A browser opens and runs the application.




3. Enter a price and click the Convert to Yen button.

   The Label control below the TextInput control displays the price in yen.

# Register the event listener with ActionScript

You can use ActionScript to register an event listener with the Button control. The ActionScript code connects the listener to a specific control event, such as a mouse click. When the control dispatches the event, your listener gets called.

1. Switch to the editor's Source mode.
2. Delete the `click` property and its value in the `<mx:Button>` tag.
3. Declare an Event object in the signature of your convertCurrency event listener as follows (in bold):

   ```
   public function convertCurrency(e:Event):void {
   ...
   ```

   When a listener function is invoked, Flex implicitly creates an Event object for you and passes it to the listener function. Therefore, it is best practice to declare an Event object in the signature of your listener function. Accordingly, you declare an object of type Event called e in the signature of the convertCurrency function.

4. Enter the following function immediately before the convertCurrency function in the `<mx:Script>` tag:

```
public function createListener():void {
  btnConvert.addEventListener(MouseEvent.CLICK, convertCurrency);
}
```

The statement in this function instructs the btnConvert object to call the `convertCurrency()` function when it "hears" the mouse click event. For more information on the `addEventListener` method, click the method name in the code and press F1.

The script block should look as follows:

```
<mx:Script>
  <![CDATA[
  public function createListener():void {
    btnConvert.addEventListener(MouseEvent.CLICK, convertCurrency);
  }

  public function convertCurrency(e:Event):void {
    var rate:Number = 120;
    var price:Number = Number(txtPrice.text);
    if (isNaN(price)) {
        lblResults.text = "Please enter a valid price.";
    } else {
      price = price * rate;
      lblResults.text = "Price in Yen: " + String(price);
    }
  }
  ]]>
</mx:Script>
```

5. In the `<mx:Application>` tag, enter the following property so that the `createListener()` function is called and the event listener is registered immediately after the application is created:

```
creationComplete="createListener()"
```

The `<mx:Application>` tag should look as follows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute" creationComplete="createListener()">
```

6. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar.
7. Enter a price and click the Convert to Yen button.

The Label control below the TextInput control displays the price in yen.

In this lesson, you wrote an event listener for a Button control, and then registered the listener using two different methods. In the first method, you registered it by specifying it in the `click` property in the Button control's MXML tag. In the second method, you registered the listener by writing an ActionScript function that connected it to the Button's click event. To learn more, see Chapter 5, "Using Events" in *Developing Flex Applications*.

CHAPTER 21

# Enterprise: Use the Data Service

21

The Adobe Flex Data Service feature is a Flex Enterprise Services feature that spans the client, network, and server tiers to provide distributed data in Flex applications. This tutorial provides two lessons on using the Data Service. The first lesson uses the ActionScript object Data Service adapter, which persists data in server memory and is useful for applications that require transient distributed data that is not persisted to a data store.

The second lesson uses the Java Data Service adapter for working with data that is persisted to a data store. The Java adapter passes data changes to methods available on an arbitrary Java class, referred to as the Java assembler. This adapter lets Java developers employ the Data Transfer Object (DTO) design pattern.

This tutorial provides the following lessons for building distributed applications that each use the Data Service, which is part of Flex Enterprise Services:



## Before you begin

Before you begin this tutorial, perform the following tasks:

- Ensure that you have installed the Flex 2 Beta 1 release and that you can run the applications in the samples web application.
- Ensure that a tutorials directory was created in the samples web application when you unzipped the tutorials.zip file on the Beta site. The directory should contain two MXML files named completed1.mxml and completed2.mxml. If you don't have a tutorials directory, the use folder names option may not have been enabled when you extracted files from the zip file.

# Build a distributed application with the ActionScript object adapter

In this lesson, you create a simple distributed application for entering and displaying notes that are shared among all clients. The application uses the ActionScript object Data Service adapter to distribute text data to all clients. The ActionScript object adapter persists data in server memory and is useful for applications that require transient data that is not persisted to a permanent data store. Changes to data in one client are sent to the server-side data service and automatically propagated to other clients.

In this lesson, you'll complete the following tasks:



## Configure a Data Service destination

In this section of the lesson, you define a server-side Data Service destination that uses the ActionScript object adapter. This destination persists data in server memory and distributes data to client applications.

1. In a text editor, open the flex-data-services.xml file located in the WEB_INF/flex directory of the samples web application.
2. Directly above the text `<destination id="contact">`, make sure the following destination definition exists. Create the destination definition if it isn't there, and save the file.

```
<destination id="notes">
  <adapter ref="actionscript"/>
  <properties>
    <metadata>
      <identity property="noteId"/>
    </metadata>
```

```
    </properties>
</destination>
```

3. If you modified the flex-data-services.xml file, restart the samples server if it is running.

## Create a new MXML file

In your favorite MXML editor or text editor, create a text file that contains the following text and save it as tutorial1.mxml file in the tutorials directory of the samples web application:

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml"
height="100%" width="100%">

</mx:Application>
```

## Create the user interface

In this section, you create the TextArea control that will display editable text in the application.

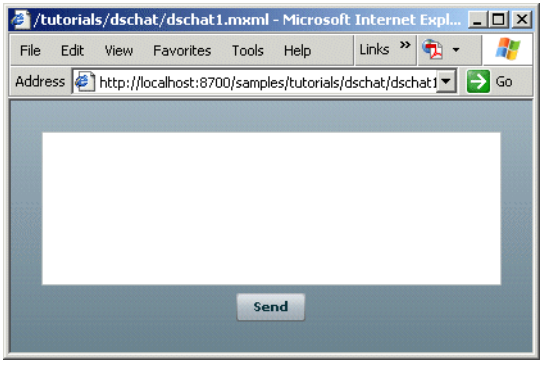
1. Create a TextArea control by adding the following MXML code after the beginning `<mx:Application>` tag in the tutorial1.mxml file.

   ```
   <mx:TextArea id="log" width="100%" height="100%"/>
   <mx:Button label="Send"/>
   ```

2. Save the file, and open the following URL in a browser window:

   http://localhost:*port*/samples/tutorials/tutorial1.mxml

   The following application appears in the browser window:



## Import the required ActionScript classes

In this section, you create a script block and import a set of classes that you will use within the script block.

1. Create a script block for ActionScript code directly below the `<mx:Application>` tag in the tutorial1.mxml file:

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

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

2. Directly below the `<![CDATA[` tag, add the following ActionScript import statements to import the mx.data.DataService and mx.collections.ArrayCollection classes:

```
import mx.data.DataService;
import mx.data.events.*;
import mx.rpc.AsyncToken;
import mx.rpc.events.*;
import mx.messaging.events.*;
import mx.utils.ObjectProxy;
```

## Create variables

In this section, you declare variables for objects that you will use within the script block.

Directly below the import statements in the script block, add the following variable declarations:

```
private var ds:DataService;
public var noteObj:Object = new Object();
public var getToken:AsyncToken;
public var noteProxy:ObjectProxy;
```

## Initialize the application

In this section, you write an event listener method that creates a DataService component and calls the DataService component's `getItem()` method, which retrieves the current note text.

1. Add the following method declaration directly under the variable declarations to create an event listener:

```
public function initApp() {

}
```

2. Add the boldface text to the `initApp()` method to create a DataService component when the `initApp()` method is called. The `ds` DataService component connects to the server-side `notes` Data Service destination, which is specified in its one argument.

```
public function initApp()
{
  ds = new DataService("Notes");

}
```

3. Add the boldface text to the `initApp()` method. This code adds a Result event listener to the DataService component and sets initial values for the noteObj object's `noteId` and `noteText` properties. It also sets the value of the getToken object to the AsyncToken object returned by the DataService component's `getItem()` method.

   The ActionScript object adapter uses the `noteId` property as a unique identifier for each noteObj object, and the `noteText` property contains the note text.

```
public function initApp()
{
  ds = new DataService("notes");
  ds.addEventListener(ResultEvent.RESULT, resultHandler);
  ds.autoCommit = false;
  noteObj.noteId = 1;
  noteObj.noteText = "This is a test note.";
  getToken = ds.getItem(noteObj, noteObj);
}
```

4. Add the boldface text to the `<mx:Application>` tag to call the `initApp()` method when the contact application is initialized:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml"
  creationComplete="initApp()">
```

## Send notes

In this section, you write an event listener method that sends text to the notes destination when you click the Send button. you then assign that method to the click event of the Send button.

1. Add the following code below the `initApp()` method in the script block:

```
private function sendMessage() {
  noteProxy.noteText = log.text;
  ds.commit();
}
```

2. Add the boldface text to the `<mx:Button>` tag to call the `sendMessage()` method when the Send button is clicked:

```
<mx:Button label="Send" click="sendMessage();"/>
```

## Handle returned data

In this section, you write an event listener method that handles data that the Data Service returns to the client.

Add the following method just below the `initApp()` method:

```
public function resultHandler(event:ResultEvent) {
  if (event.call == getToken)
    {
      noteProxy = ObjectProxy(event.result);
      noteProxy.addEventListener(ObjectEvent.CHANGE, changeHandler);
      log.text = noteProxy.noteText;
  }
}
```

In the `resultHandler()` method, `event.call` is the AsyncToken that the `getItem()` method returns. Because the noteObj object is an anonymous object, the Data Service wraps it in an ObjectProxy object and returns that object in the result event. Anonymous objects do not implement the object change event, but ObjectProxy objects do. This is why noteProxy is used instead of noteObj in the following code:

```
log.text = noteProxy.noteText
```

## Handle data changes

In this section, you write an event listener method that handles changes to text in the `log` TextArea control.

Add the following method just below the `resultHandler()` method. When the text in the `log` TextArea control changes, `noteProxy.noteText` is set to the new value.

```
public function changeHandler(event:ObjectEvent) {
  log.text = noteProxy.noteText;
}
```

## Verify that your code is correct

Your code should match the following code example. Verify that the content is correct and save the tutorial1.mxml file.

```
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml"
 height="100%" width="100%"
 creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.data.DataService;
      import mx.data.events.*;
```

```
        import mx.rpc.AsyncToken;
        import mx.rpc.events.*;
        import mx.messaging.events.*;
        import mx.utils.ObjectProxy;

        private var ds:DataService;
        public var noteObj:Object = new Object();
        public var getToken:AsyncToken;
        public var noteProxy:ObjectProxy;

        private function initApp() {
          ds = new DataService("notes");
          ds.addEventListener(ResultEvent.RESULT, resultHandler);
          ds.autoCommit = false;

          noteObj.noteId = 1;
          noteObj.noteText = "This is a test note.";
          getToken = ds.getItem(noteObj, noteObj);
        }

        private function sendMessage() {
          noteProxy.noteText = log.text;
          ds.commit();
        }

        public function resultHandler(event:ResultEvent) {
          if (event.call == getToken)
          {
            noteProxy = ObjectProxy(event.result);
            noteProxy.addEventListener(ObjectEvent.CHANGE,
              changeHandler);
            log.text = noteProxy.noteText;
          }
        }

        public function changeHandler(event:ObjectEvent) {
          log.text = noteProxy.noteText;
        }

      ]]>
    </mx:Script>
    <mx:TextArea id="log" width="100%" height="100%"/>
    <mx:Button label="Send" click="sendMessage();"/>
</mx:Application>
```

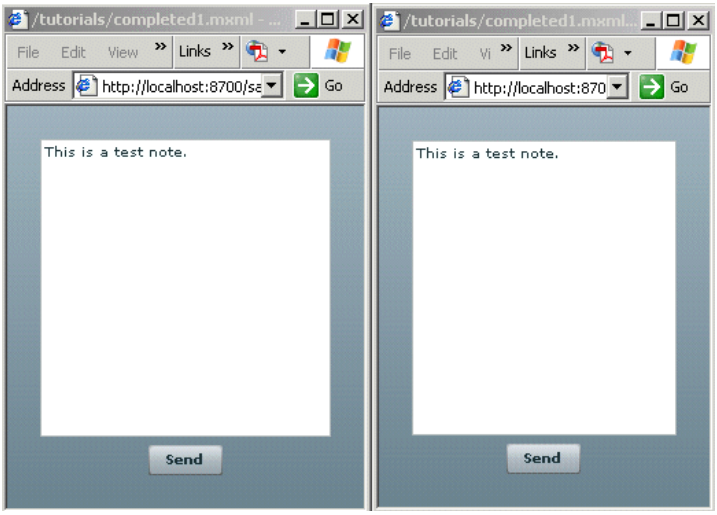
## Run the completed notes application

In this section, you run the completed notes application in two browser windows to see automatic updates in one window when data is changed in the other.

1. Open the following URL in two browser windows:

   http://localhost:*port*/samples/tutorials/tutorial1.mxml

2. Make sure both instances of the application look like those in the following example. Resize the browser windows so that you can see both instances of the application at the same time.




3. In one of the browser windows, type a new line of text in the text area and click the Send button. Your change should update the application that is running in the other browser window.
4. In the second window, type additional text and click the Send button. The change should be appended to the text already displayed in the application that is running in the other browser window.

In this lesson, you used the Data Service feature with the ActionScript object adapter to create a distributed data application that automatically synchronized data among multiple clients and a server-side data resource. You built the client-side part of the application, and took a look at the server-side components of the application. To learn more, see the following topics in *Developing Flex Applications:*

- Understanding the Flex Data Service Feature
- Distributing Data in Flex Applications
- Configuring the Data Service

# Build a distributed application with the Java adapter

In this lesson, you create a simple contact application that automatically retrieves contact information from a database and displays it in a DataGrid component. Changes to data in one client are sent to the server-side data service and automatically propagated to other clients.

After building and running the client application, you learn how the server-side Data Service enables the distribution and synchronization of data in the application.

In this tutorial, you'll complete the following tasks:



## Create a new MXML file

In your MXML editor, create a text file that contains the following text and save it as tutorial2.mxml file in the tutorials directory of the samples web application:

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml">

</mx:Application>
```

## Create the user interface

In this section, you create the editable DataGrid control that will display editable contact information in the contact application.

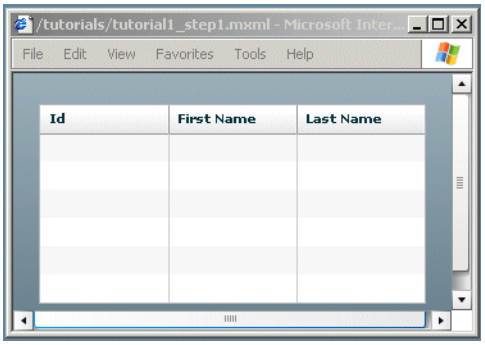
1. Create a three-column editable DataGrid control by adding the following MXML code after the beginning `<mx:Application>` tag in the tutorial2.mxml file. Set the `editable` property of the first column to `false`.

```
<mx:DataGrid id="dg" editable="true">
  <mx:columns>
    <mx:DataGridColumn columnName="contactId" headerText="Id"
  editable="false"/>
    <mx:DataGridColumn columnName="firstName" headerText="First Name"/>
    <mx:DataGridColumn columnName="lastName" headerText="Last Name"/>
  </mx:columns>
</mx:DataGrid>
```

2. Save the file, and open the following URL in a browser window:

   http://localhost:*port*/samples/tutorials/tutorial2.mxml

   The browser window should display the following application:




## Import the required ActionScript classes

In this section, you import the mx.collections.ArrayCollection and mx.data.DataService classes so that you can create a DataService component and an ArrayCollection object. Later in the tutorial, you will create a DataService component that requests data from a server-side Data Service destination and fills an ArrayCollection object with that data.

1. Create a script block for ActionScript code directly above the `<mx:DataGrid>` tag in the tutorial2.mxml file:

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

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

2. Directly below the `<![CDATA[` tag, add the following ActionScript import statements to import the mx.data.DataService and mx.collections.ArrayCollection classes:

```
import mx.data.DataService;
import mx.collections.ArrayCollection;
```

## Create variables

In this section, you add variables for the ArrayCollection and DataService objects to the script block.

1. Directly below the import statements in the script block, add the following variable declarations for the `ds` and `contacts` variables:

```
public var ds:DataService;
public var contacts:ArrayCollection;
```

2. Add a `[Bindable]` metadata tag directly above the `contacts` variable declaration:

```
[Bindable]
public var contacts:ArrayCollection;
```

   The `[Bindable]` metadata tag indicates to the MXML compiler that `contacts` is a bindable property. By making it a bindable property, you can bind it into the `dataProvider` property of the DataGrid control and display data in the DataGrid control.

## Bind the ArrayCollection object to the DataGrid

In this section, you bind the `contact` ArrayCollection object to the DataGrid control's `dataProvider` property to fill the DataGrid control with data from the ArrayCollection object.

1. Add the boldface text to the `<mx:DataGrid>` tag:

```
<mx:DataGrid id="dg" dataProvider="{contacts}" editable="true">
```

2. Save the tutorial2.mxml file.

## Fill the ArrayCollection object with data

In this section, you write an event listener method that creates a DataService object and an ArrayCollection object, and calls the DataService's `fill()` method to fill the ArrayCollection with data.

1. Add the following method declaration directly under the variable declarations to create an event listener:

```
public function initApp() {

}
```

2. Add the boldface text to the initApp() method to create a DataService component and an ArrayCollection object when the `initApp()` method is called. The `ds` DataService connects to the server-side `contact` Data Service destination, which is specified in its one argument.

```
public function initApp()
{
  contacts = new ArrayCollection();
  ds = new DataService("contact");
  ds.fill(contacts);
}
```

3. Add the boldface text to the `initApp()` method to call the DataService object's `fill()` method when the `initApp()` method is called. The `fill()` method requests data from the server-side Data Service destination and fills the contacts ArrayCollection with that data. The DataService object manages all updates, additions, and deletions to the data in the ArrayCollection.

```
public function initApp()
{
  contacts = new ArrayCollection();
  ds = new DataService("contact");
  ds.fill(contacts);
}
```

4. Add the boldface text to the `<mx:Application>` tag to call the `initApp()` method when the contact application is initialized:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml"
  creationComplete="initApp()">
```

## Verify that your code is correct

Your code should match the following code example. Verify that the content is correct and save the tutorial2.mxml file.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" >

  <mx:Script>
    <![CDATA[
      import mx.data.DataService;
      import mx.collections.ArrayCollection;
      public var ds:DataService;

      [Bindable]
      public var contacts:ArrayCollection;

      public function initApp(){
        contacts = new ArrayCollection();
        ds = new DataService("contact");
        ds.fill(contacts);
      }
    ]]>
  </mx:Script>

    <mx:DataGrid id="dg" dataProvider="{contacts}" editable="true">
      <mx:columns>
        <mx:DataGridColumn columnName="contactId" headerText="Id"
          editable="false"/>
        <mx:DataGridColumn columnName="firstName" headerText="First Name"
        />
        <mx:DataGridColumn columnName="lastName" headerText="Last Name"/>
      </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

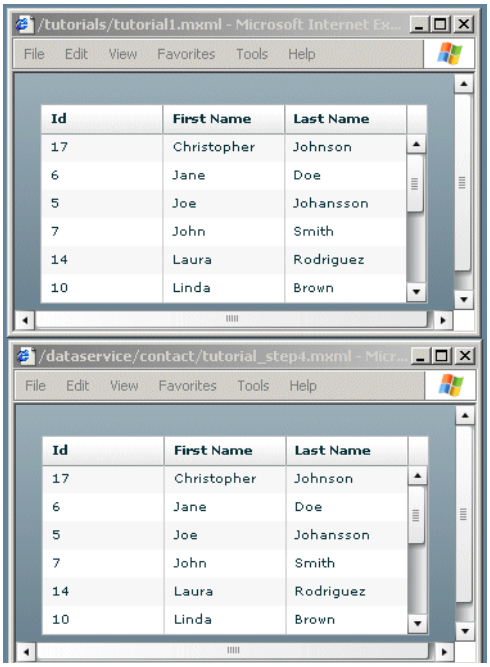
## Run the completed contact application

In this section, you run the completed contact application in two browser windows to see automatic updates in one window when data changes in the other.

1. Open the following URL in two browser windows:

   http://localhost:*port*/samples/tutorials/tutorial2.mxml

2. Ensure both instances of the application look like the following example. Resize the browser windows so that you can see both instances of the application at the same time.




3. In one of the browser windows, click a DataGrid cell in the First Name column, and change the name.
4. Tab to the Last Name column of the DataGrid. Your change should automatically update the application that is running in the other browser window. After each change, the DataService component's `commit()` method is automatically called to send the changed data to the server-side Data Service.

The following sections explain some of the server-side functionality that enables data distribution and synchronization in the tutorial2 application.

## View the server-side Data Service destination

The tutorial2.mxml file contains a DataService component that takes the name of a server-side Data Service destination named `contact` as an argument in its constructor. This reference to the contact destination is the only thing the client application needs to communicate with the destination. The `contact` destination is defined in the flex-data-services.xml file in the WEB_INF/flex directory of the samples web application. The following figure shows the XML code that defines the destination:

```
<destination id="contact">
  <adapter ref="java-dao" />
  <properties>
    <metadata>
      <identity property="contactId"/>
    </metadata>

    <network>
      <session-timeout>0</session-timeout>
      <paging enabled="false" size="10" />
      <throttle-inbound policy="ERROR" max-frequency="500"/>
      <throttle-outbound policy="REPLACE" max-frequency="500"/>
    </network>

    <server>
      <assembler>
        <class>samples.contact.ContactAssembler</class>
        <singleton>true</singleton>
      </assembler>

    <fill-method>
      <name>loadContacts</name>
    </fill-method>
    <fill-method>
      <name>loadContacts</name>
      <params>java.lang.String</params>
    </fill-method>

    <sync-method>
      <name>syncContacts</name>
      </sync-method>
    </server>
  </properties>
</destination>
```

Reference to a Data Service adapter configuration; this destination uses the Java object adapter

Network -related settings

Java class that carries out data fill and synchronization operations between client and server.

Mapping to method on adapter class to get data.

Mapping to method on adapter class that synchronizes multipe versions of data.

The previous image calls out the following elements of the `contact` destination definition:

| Destination section | Description |
|---|---|
| adapter | References the Java object adapter configuration that the contact destination uses. This is also defined in the flex-data-service.xml file. The Java object adapter lets you interact with a Java object to obtain data and commit data to a server-side data resource. |
| network | Contains settings for how data messages are passed between the server-side Data Service and the client-side DataService component. For example, the paging element specifies whether data sent from the server to the client is chunked into smaller subsets of data instead of being sent all at once. |
| assembler | Specifies the assembler class, which is the Java class that passes data between a server-side data resource and the client-side DataService component.<br>The assembler class is a custom class that you write. It is usually an implementation of the Transfer Object Assembler design pattern, which is described at http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html. |
| fill-method | Specifies a mapping to a fill method that is invoked when the client-side `DataService.fill()` method is called to fill an ArrayCollection object. You can implement any number of methods as fill methods, but they must be differentiated by the types of their parameters. Each of these methods can accept an arbitrary number of parameters of varying types. Based on the parameters that the client-side `DataService.fill()` method provides, the appropriate fill method on the assembler class is invoked. |
| sync-method | Specifies a method that accepts a list of data changes to allow synchronization of data among multiple clients and the backend data resource.<br>A sync method takes a single argument, which is a standard java.util.List implementation that contains objects of type flex.data.ChangeObject. Each ChangeObject object in the list contains methods to access the application-specific changed Object instance, as well as convenience methods for describing the type of change and for accessing the changed data members. |

## View the assembler class

The contact destination uses the Java adapter. This is one of several types of Data Service adapters that Flex Enterprise Services provides. As previously noted, the contact destination specifies an assembler class, which is a custom Java class that gets data from a data resource and handles the synchronization of data among clients and the data resource.

An assembler must have a zero-argument constructor. The assembler for this application is instantiated as a singleton, which means there is only one instance of the class for the entire web application. The `singleton` element in the assembler section of the destination specifies that the assembler is a singleton. If you do not set the `singleton` value to `true`, a new instance of the assembler is created for each client operation.

The destination specifies the methods of the assembler class that are invoked to get data and synchronize multiple versions of data. In addition to those methods, the assembler class also implements methods for getting individual data items, and creating, updating, and deleting data items; these methods are implementations of methods in the flex.data.ChangeObject interface.

The following example shows the source code of the contact application's assembler class. This class delegates the actual calls to a SQL database, to a data access object (DAO) called ContactDAO. The source code for the ContactAssembler class and the ContacDAO class are in the WEB_INF/src/samples/contact directory. The compiled classes are in the WEB_INF/classes/samples/contact directory.

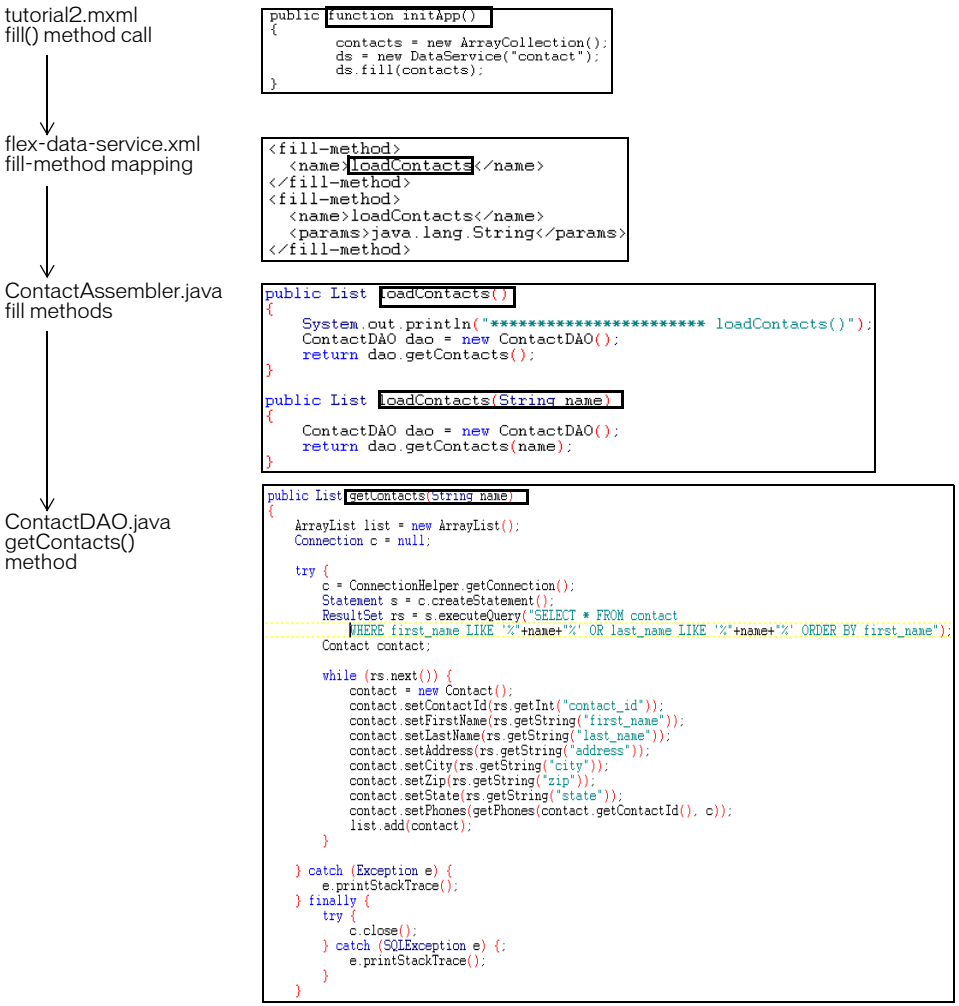
### View the fill methods

A fill method of the assembler class is called as a result of the client-side DataService's `fill()` method being called. The following code example shows the two methods that are specified as fill methods in the contact destination definition. The methods have the same name but different signatures, yet both return a List object that contains Contact objects. One method takes no arguments, while the other takes a name of type String as an argument. One of the methods is called, depending on whether or not the `DataService.fill()` request from the client specifies an argument of type String.

```
import flex.data.ChangeObject;
public class ContactAssembler {
...
  public List loadContacts() {
    System.out.println("********************** loadContacts()");
    ContactDAO dao = new ContactDAO();
    return dao.getContacts();
  }

  public List loadContacts(String name) {
```

```
    ContactDAO dao = new ContactDAO();
    return dao.getContacts(name);
  }
...
```

The following figure shows the flow of a `fill()` method call from the client-side contact application:



## View the sync method

The sync method of an assembler class lets you handle data changes sent from client-side DataService components. A sync method accepts one input parameter, which is a java.util.List object that contains a list of data changes of type flex.data.ChangeObject. The list of changes can include new data items, updates, and deletions.

Depending on whether a change is an add, update, or delete, the sync method calls the class's `doCreate()`, `doUpdate()`, or `doDelete()` method. The `doCreate()`, `doUpdate()`, and `doDelete()` methods are implementations of methods in the flex.data.ChangeObject interface. These methods call methods on the ContactDAO object, which interacts with a SQL database.

The following example shows the Java source code for the ContactAssember class's sync method:

```
import flex.data.ChangeObject;
public class ContactAssembler {
...
  public List syncContacts(List changes) {
    Iterator iterator = changes.iterator();
    ChangeObject co;
    while (iterator.hasNext()) {
      co = (ChangeObject) iterator.next();
      if (co.isCreate()) {
        co = doCreate(co);
      }
      else if (co.isUpdate()) {
        doUpdate(co);
      }
      else if (co.isDelete()) {
        doDelete(co);
      }
    }
    return changes;
   }
```

The following example shows the assembler class's `doCreate()`, `doUpdate()`, `doDelete()` methods:

```
private ChangeObject doCreate(ChangeObject co) {
  ContactDAO dao = new ContactDAO();
  Contact contact = dao.create((Contact) co.getNewVersion());
  co.setNewVersion(contact);
  return co;
}

private void doUpdate(ChangeObject co) {
  ContactDAO dao = new ContactDAO();
  try
```

```
    {
      dao.update((Contact) co.getNewVersion(), (Contact)
    co.getPreviousVersion());
    }
    catch (ItemNotFoundException e)
    {
      System.err.println("*** Throwing DataSyncException when trying to
    update contact id=" + ((Contact) co.getNewVersion()).getContactId() );
      throw new DataSyncException(co);
    }
}

private void doDelete(ChangeObject co) {
    ContactDAO dao = new ContactDAO();
    try {
      dao.delete((Contact) co.getNewVersion(), (Contact)
    co.getPreviousVersion());
    }
      catch (ItemNotFoundException e) {
      System.err.println("*** Throwing DataSyncException when trying to
    delete contact id=" + ((Contact) co.getNewVersion()).getContactId() );
      throw new DataSyncException(co);
    }
}
```

In this lesson, you used the Data Service feature with the Java adapter to create a distributed data application that automatically synchronized data among multiple clients and a server-side data resource. You built the client-side part of the application, and took a look at the server-side components of the application. To learn more, see the following topics in *Developing Flex Applications:*

- Chapter 58, “Understanding the Flex Data Service feature”
- Chapter 55, “Distributing Data in Flex Applications”
- Chapter 60, “Configuring the Data Service”

CHAPTER 23

# Enterprise: Use ColdFusion Event Gateway Adapter

23

This tutorial shows you how to create a Flex application to send a message to a ColdFusion application. The sample application does not take advantage of capabilities that are unique to Adobe Flex, instead, it describes the communication with ColdFusion applications that the ColdFusion Event Gateway Adapter enables.

To show the capabilities of the ColdFusion Event Gateway Adapter and the Flex Messaging event gateway, the sample application lets you enter information in a form in a Flex application. The Flex application sends the information through the ColdFusion Event Gateway Adapter and Flex Messaging event gateway to the ColdFusion application. The ColdFusion application then sends an e-mail message that contains the message to the recipient specified in the Flex application.

In this tutorial, you'll complete the following tasks:



## Set up your development environment

The ColdFusion Event Gateway Adapter lets you create applications in which Flex Enterprise Services 2 and Macromedia ColdFusion MX 7.1 from Adobe communicate. Flex Enterprise Services 2 includes the ColdFusion Event Gateway Adapter. ColdFusion MX 7.1 includes the Flex Messaging event gateway.

To complete this tutorial, you must have the following products installed:

- Flex Enterprise Services 2
- ColdFusion MX 7.1

## Start Flex Enterprise Services 2 and ColdFusion MX 7.1

To set up your development environment, you must start Flex Enterprise Services 2 and ColdFusion MX 7.1. This tutorial assumes that both Flex Enterprise Services 2 and ColdFusion are running on localhost (127.0.0.1) on your local computer. Because of the way the Remote Method Invocation (RM)I registry is created and maintained, Adobe recommends that you start Flex Enterprise Services 2, and then start ColdFusion.

| NOTE | The example ColdFusion application uses the cfmail tag. You must set up an e-mail server in the ColdFusion MX Administrator before testing the application. |
|---|---|

## Enable the ColdFusion Event Gateway Adapter

To ensure that Flex Enterprise Services 2 recognizes the ColdFusion Event Gateway Adapter, you edit the flex-message-service.xml file, which is located in the C:\fes2\jrun4\server\default\samples\WEB-INF\flex directory. Uncomment the ColdFusion Event Gateway Adapter definition and the Flex Messaging event gateway definition.

1. Open flex-message-service.xml in a text editor.
2. Add a close comment (-->) at the end of the line that starts with `<!-- description of message service configuration`, so the line appears as follows:

   ```
   <!-- description of message service configuration -->
   ```

3. Add an open comment (<!--) to the line that starts with `//multiple destinations may be specified`, so the line appears as follows:

   ```
   <!-- //multiple destinations may be specified
   ```

4. Save the file.

## Create an instance of the Flex Messaging event gateway

To be able to communicate with the ColdFusion application through the Flex Event Gateway, you must create an instance of the gateway.

1. Create a blank file handleemail.cfc in the C:\CFusionMX7\wwwroot\flexgatewayexamples directory. (The flexgatewayexamples directory does not already exist.)
2. Start the ColdFusion MX Administrator.
3. Select Event Gateways > Gateway Instances.

4. Enter Flex2CF2 as the Gateway ID.
5. Select Flex Messaging - Flex as the Gateway Type.
6. Specify C:\CFusionMX7\wwwroot\flexgatewayexamples\handleemail.cfc as the CFC Path.
7. Specify *{cf.rootdir}*/gateway/config/flex-test.cfg as the Configuration File.
8. Select Manual as the Startup Mode.
9. Click Add Gateway Instance.

# Create the Flex application

The Flex application in this tutorial is a simple form in which you specify the elements of an e-mail message, including the recipient, the sender, the subject, and the message body.

1. Create a blank file and enter the following code:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2005/mxml" xmlns="*"
  layout="absolute"
   creationComplete="initApp()">

   <mx:Script>
    <![CDATA[
      import mx.messaging.events.*;
      import mx.messaging.Producer;
      import mx.messaging.messages.Message;
      import mx.messaging.AsyncMessage;

      public var pro:mx.messaging.Producer;

      private function initApp() {
        pro = new mx.messaging.Producer();
        pro.destination = "gateway1";
        pro.resendAttempts = 5;
        pro.resendInterval = 5000;
      }

      public function sendMessage() {
        private var msg:Message = new AsyncMessage();

        msg.headers.gatewayid = "Flex2CF2";
        msg.body = new Object();
        msg.body.emailto = emailto.text;
        msg.body.emailfrom = emailfrom.text;
        msg.body.emailsubject = emailsubject.text;
        msg.body.emailmessage = emailmessage.text;
```

```
            pro.send(msg);
          }

        ]]>
      </mx:Script>

    <mx:TextInput x="103" y="13" width="291" id="emailto" editable="true"/>
    <mx:TextInput x="103" y="43" width="291" id="emailfrom" editable="true"/
      >
    <mx:TextInput x="103" y="73" width="291" id="emailsubject"
      editable="true"/>
    <mx:TextArea x="103" y="102" width="291" height="236" id="emailmessage"
      editable="true"/>

    <mx:Label x="63" y="15" text="To:" textAlign="right"/>
    <mx:Label x="37" y="103" text="Message:" textAlign="right"/>
    <mx:Label x="52" y="45" text="From:"/>
    <mx:Label x="37" y="75" text="Subject:"/>

    <mx:Button x="402" y="13" label="Send" id="emailsend"
      click="sendMessage();"/>

</mx:Application>
```

2. Save the file as flexemail2cf.mxml. in the C:\fes2\jrun4\servers\default\samples\dataservice\waitlist folder.

# Create the ColdFusion application

The ColdFusion application puts the information received from the Flex application in a structure. It then sends an e-mail message by using elements of the structure.

A ColdFusion application can handle data sent from a Flex application in either the header or the body of the message. The sample Flex application sends the data in the body of the message. To create the ColdFusion application, you create a ColdFusion component.

1. Create a blank file and enter the following code:

```
<cfcomponent displayname="Send e-mail from Flex application"
  hint="Handles message from Flex">

<!--- Handle the event from Flex. --->
<cffunction name="onIncomingMessage" returntype="any">
  <cfargument name="event" type="struct" required="true">

  <!--- Get the structure that holds the message object sent from Flex
  <cfset messagebody = event.data.body>

  <!--- Populate the structure. --->
```

```
    <cfset mailfrom="#messagebody.emailfrom#">
    <cfset mailto="#messagebody.emailto#">
    <cfset mailsubject="#messagebody.emailsubject#">
    <cfset mailmessage ="#messagebody.emailmessage#">

    <!--- Send the e-mail. --->
    <cfmail from="#mailfrom#"
      to="#mailto#"
      subject="#mailsubject#">
      <cfoutput>#mailmessage#</cfoutput>
    </cfmail>
  </cffunction>
  </cfcomponent>
```

2. Save the file handleemail.cfc as C:\CFusionMX7\wwwroot\flexgatewayexamples.

# Test the application

To test the sample application, you must set up the testing environment, run the Flex application, and then view your e-mail client to ensure that the application sent the e-mail message successfully.

## Set up the testing environment

Before testing the sample application, do the following:

- Ensure that Flex Enterprise Services 2 is running.
- Ensure that ColdFusion is running.

| TIP | To view the messages, start ColdFusion in a console by goingto the CFusionMX7\bin directory and entering `cfstart`. |
|---|---|

- Start the Flex2CF2 Flex Event Gateway instance.

### To start the Flex2CF2 Flex Event Gateway instance:

1. Start the ColdFusion MX Administrator.
2. Select Event Gateways >Gateway Instances.
3. Click the Start button next to the Flex2CF2 gateway instance.

## Run the application

To run the Flex application, you browse to the MXML file.

1. Open the http://localhost:8700/samples/dataservice/waitlist/flexmail2cf.mxml file in a browser.
2. Enter a valid e-mail address in the To text box. Ensure that the e-mail address is one whose incoming e-mail you can check.
3. Enter the name of the sender in the From text box.
4. Enter the subject in the Subject text box.
5. Enter the message in the Message text area.
6. Click Send.

## Check e-mail messages

To ensure that the application executed successfully, check the e-mail messages of the recipient specified in the Flex application. There should be an e-mail message tsent from the sender, with the subject and body hat you specified in the Flex application.