CHAPTER 27

Using View States

27

View states let you vary the content and appearance of a component or application, typically in response to a user action. For example, the base, or default, view state of the application could be the home page and include a logo, a sidebar, and some welcome content. When the user clicks a button in the sidebar, the application dynamically changes its appearance, meaning its view state, by replacing the main content area with a purchase order form but leaving the logo and sidebar in place.

To use view states, you define the base view state, and one or more additional view states that specify modifications to the base view state. This topic describes how to control component or application display by using view states.

Transitions define how a change of view state looks as it occurs on the screen. For information on transitions, see Chapter 28, “Using Transitions,” on page 1015.

Contents

About view states 983
Create and apply view states 991
Defining view state overrides. 1001
Defining view states in custom components 1017
Using view states with a custom item renderer 1020
Using view states with history management. 1024
Creating your own override classes 1026

About view states

In many rich Internet applications, the interface changes based on the task the user is performing. A simple example is an image that changes when the user rolls the mouse over it. More complex examples include user interfaces whose contents change depending on the user’s progress through a task, such as changing from a browse view to a detail view. View states let you easily implement such applications.

At its simplest, a view state defines a particular view of a component. For example, a product thumbnail could have two view states; a base state with minimal information, and a “rich” state with links for more information or to add the item to a shopping cart, as the following figure shows:

Base view state

Rich view state

To create a view state, you define a base state, and then define a set of changes, or overrides, that modify the base state to define the new view state. Each additional view state can modify the base state by adding or remove child components, by setting style and property values, or by defining state-specific event handlers.

A view state does not have to modify the base state. A view state can specify modifications to any other view state. This lets you create a set of view states that share common functionality while adding the overrides specific to each view state. For more information, see “Basing a view state on another view state” on page 995.

Defining a login interface by using view states

One use of view states is to implement a login and registration form. In this example, the base view state prompts the user to log in, and includes a LinkButton control that lets the user register, if necessary, as the following image shows:

Base view state - Login

Login

Username:

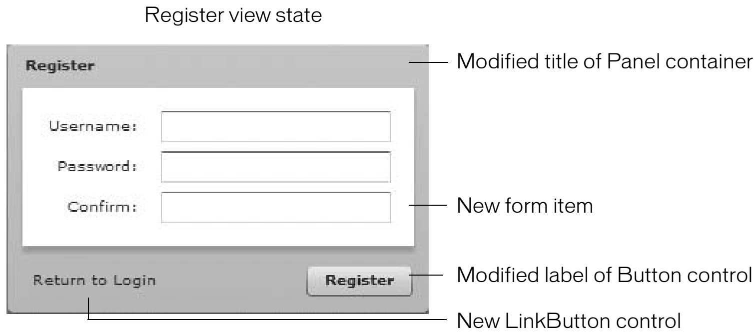
Password:

Need to Register?

Login

LinkButton control

If the user selects the Need to Register link, the form changes view state to display registration information, as the following image shows:



Notice the following changes to the base view state to create this new view state:

- The title of the Panel container is set to Register
- The Form container has a new TextInput control for confirming the password
- The label of the Button control is set to Register
- The LinkButton control has been replaced with a new LinkButton control that lets the user change state back to the base view state

When the user clicks the Return to Login link, the view state changes back to base view state to display the Login form, reversing all of the changes made when changing to the register view state.

The following table shows the classes that you use to define view states:

Class	Description
AddChild and RemoveChild	Adds or removes a child component as part of a change of view state. For more information, see "Adding and removing components by using overrides" on page 1005.
SetEventHandler	Adds an event handler as part of a change of view state. For more information, see "Setting overrides on event handlers" on page 1014.
SetProperty	Sets a component property as part of a change of view state. For more information, see "Setting overrides on component properties" on page 1002.
SetStyle	Sets a style property on a component as part of a change of view state. For more information, see "Setting overrides on component styles" on page 1004.

Example: Login form application

The base view state of an application or component is the default view state. If you omit any view state specifications, you can consider your application to have a singe view state: the base state. To add additional view states to the base view state, you use the `<mx:states>` tag. Within the body of the `<mx:states>` tag, you can add one ore more `<mx:State>` tags, one for each additional view state.

The UIComponent class defines the `currentState` property that you use to set the view state. In the previous example, you use Button controls to set the `currentState` property of the Application object to either "NewButton", the name of the view state specified by the `<mx:State>` tag, or an empty String, "", corresponding to the base view state. When the application starts, the default value of the `currentState` property is ''.

An application or component can set the initial view state to a non-base view state. To use a view state other than the base state as the initial view state, set the `currentState` property to the specific view state. Use code such as the following, for example, to specify that a component is initially in its collapsed state:

```
<myComps:CollapsePanel currentState="collapsed">
```

The following example creates the Login and Register forms shown in “Example: Login form application” on page 986. This application has the following features:

- When the user clicks the Need to Register LinkButton control, the event handler for the `click` event sets the view state to Register.
- The Register state code adds a TextInput control, changes properties of the Panel container and Button control, removes the existing LinkButton controls, and adds a new LinkButton control.

- When the user clicks the Return to Login LinkButton control, the event handler for the `click` event resets the view state to the base view state.

> **NOTE** For an example that adds a transition to animate the change between view states, see "Example: Using transitions with a login form" on page 1016.

```
<?xml version="1.0"?>
<!-- states\LoginExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    verticalAlign="middle">

    <!-- The Application class states property defines the view states.-->
    <mx:states>
        <mx:State name="Register">
            <!-- Add a TextInput control to the form. -->
            <mx:AddChild relativeTo="{loginForm}"
                position="lastChild">
                <mx:FormItem id="confirm" label="Confirm:">
                    <mx:TextInput/>
                </mx:FormItem>
            </mx:AddChild>

           <!-- Set properties on the Panel container and Button control.-->
            <mx:SetProperty target="{loginPanel}"
                name="title" value="Register"/>
            <mx:SetProperty target="{loginButton}"
                name="label" value="Register"/>

            <!-- Remove the existing LinkButton control.-->
            <mx:RemoveChild target="{registerLink}"/>

            <!-- Add a new LinkButton control to change view state
                back to the login form.-->
            <mx:AddChild relativeTo="{spacer1}" position="before">
                <mx:LinkButton label="Return to Login"
                    click="currentState=''"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:Panel id="loginPanel"
        title="Login"
        horizontalScrollPolicy="off"
        verticalScrollPolicy="off">

        <mx:Form id="loginForm">
            <mx:FormItem label="Username:">
                <mx:TextInput/>
            </mx:FormItem>
```

```
            <mx:FormItem label="Password:">
                <mx:TextInput/>
            </mx:FormItem>
        </mx:Form>

        <mx:ControlBar>
      <!-- Use the LinkButton to change to the Register view state.-->
            <mx:LinkButton id="registerLink"
                label="Need to Register?"
                click="currentState='Register'"/>
            <mx:Spacer width="100%" id="spacer1"/>
            <mx:Button label="Login" id="loginButton"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

Example: Controlling layout using view states

In this example, you define an application with three Panel containers and three view states, as the following example shows:

Base view state

One view state

Two view state

To change view state, click on the Panel container that you want to display in the expanded size. For a version of this example that adds a transition to animate the view state change, see "Defining transitions" on page 1018.

```
<?xml version="1.0"?>
<!-- states/ThreePanel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="400">

    <!-- Define the two view states, in addition to the base state.-->
    <mx:states>
        <mx:State name="One">
            <mx:SetProperty target="{p1}" name="x" value="110"/>
            <mx:SetProperty target="{p1}" name="y" value="0"/>
            <mx:SetProperty target="{p1}" name="width" value="200"/>
            <mx:SetProperty target="{p1}" name="height" value="210"/>
            <mx:SetProperty target="{p2}" name="x" value="0"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="100"/>
            <mx:SetProperty target="{p2}" name="height" value="100"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
        <mx:State name="Two">
            <mx:SetProperty target="{p2}" name="x" value="110"/>
            <mx:SetProperty target="{p2}" name="y" value="0"/>
            <mx:SetProperty target="{p2}" name="width" value="200"/>
            <mx:SetProperty target="{p2}" name="height" value="210"/>
            <mx:SetProperty target="{p3}" name="x" value="0"/>
            <mx:SetProperty target="{p3}" name="y" value="110"/>
            <mx:SetProperty target="{p3}" name="width" value="100"/>
            <mx:SetProperty target="{p3}" name="height" value="100"/>
        </mx:State>
    </mx:states>

    <!-- Define the Canvas container holding the three Panel containers.-->
    <mx:Canvas id="pm" width="100%" height="100%">
        <mx:Panel id="p1" title="One"
                x="0" y="0" width="100" height="100"
                click="currentState='One'">
            <mx:Label fontSize="24" text="One"/>
        </mx:Panel>

        <mx:Panel id="p2" title="Two"
                x="0" y="110" width="100" height="100"
                click="currentState='Two'">
            <mx:Label fontSize="24" text="Two"/>
        </mx:Panel>
```

```
        <mx:Panel id="p3" title="Three"
                x="110" y="0" width="200" height="210"
                click="currentState=''">
            <mx:Label fontSize="24" text="Three"/>
        </mx:Panel>
    </mx:Canvas>
</mx:Application>
```

Comparing view states to navigator containers

View states give you one way to change the appearance of an application or component in response to a user action. You can also use navigator containers, such as the Accordion, Tab Navigator, and ViewStack containers when you perform changes that affect several components.

Your choice of using navigator containers or states depends on your application requirements and user-interface design. For example, if you want to use a tabbed interface, use a TabNavigator container. You might decide to use the Accordion container to let the user navigate through a complex form, rather than using view states to perform this action.

When comparing view states to ViewStack containers, one thing to consider is that you cannot easily share components between the different views of a ViewStack container. That means you will have to recreate a component each time you change views. For example, if you want to show a search component in all views of a View Stack container, you have to define it in each view.

When using view states, you can easily share components across multiple view states by defining the component once, and then including it in each view state. For more information about sharing components among view states, see “Adding and removing components by using overrides” on page 1005.

For more information on navigator containers, see Chapter 16, “Using Navigator Containers,” on page 615.

Additional state-based application techniques

Consider the following additional techniques for structuring and implementing a state-based applications.

- Create cascading view state definitions, where you use the `basedOn` property to explicitly base one view state on another view state. Use this technique when you have multiple view states that all include some common elements. You separate all the common elements into one view state on which you base the other view states. For more information, see “Basing a view state on another view state” on page 995.

- Use view states that replace major sections of the display. For example, a shopping application could use view states to control whether the main part of the display shows a product selector panel or a checkout accordion. For an example, see "Example: Controlling layout using view states" on page 988.
- Use transitions to control the changes between view states; for example, you can add a resize effect to any component that changes size when the view state changes. For more information on transitions, see Chapter 28, "Using Transitions," on page 1015.

Create and apply view states

This section describes the basic concepts for creating and applying view states, and concludes with a simple example that shows how to use these rules.

Creating view states

The properties, styles, event handlers, and child components that you define for an application or component specify its base, or default, view state. Each view state specifies changes to the base view state, or to another view state.

Consider the following when you define a view state.

- You can only define view states at the root of an application or at the root of a custom component; that is, as a property of the `<mx:Application>` tag of an application file, or of the root tag of an MXML component.
- You define states by using the component's `states` property, normally as an `<mx:states>` tag in MXML.
- You populate the `states` property with an Array of one or more State objects, where each State object corresponds to a view state.
- You use the `name` property of the State object to specify its identifier. To change to that view state, you set a component's `currentState` property to the value of the `name` property.

The following MXML code shows this structure:

```
<mx:Application>
  <!-- Define the view states.
       The <mx:states> tag can also be a child tag of
       the root tag of a custom component.
  -->
  <mx:states>
    <mx:State name="State1">
      <mx:AddChild/>
      .
```

```
      <mx:SetStyle/>
      .
      <mx:SetProperty/>
      .
      <mx:SetEventHandler/>
      .
    </mx:State>
    <mx:State name="State2">
      .
      .
    </mx:State>
    .
    .
  </mx:states>
  <!-- Application definition that defines the base view state. -->
  .
  .
</mx:Application>
```

View state overrides

A view state is defined by a set of changes, or *overrides*, to the base view state. You can define the following types of overrides in a view state.

- Setting the following component characteristics:

  Properties, by using the SetProperty class. The following line disables the button1 Button control as part of a view state:

  ```
  <mx:SetProperty target="{button1}" name="enabled" value="false"/>
  ```

  For more information, see "Setting overrides on component properties" on page 1002.

  Styles, by using the SetStyle class. The following line sets the `color` property of the button1 Button control as part of a view state:

  ```
  <mx:SetStyle target="{button1}" name="color" value="0xAAAAAA"/>
  ```

  For more information, see "Setting overrides on component styles" on page 1004.

- Adding or removing child objects, by using the AddChild and RemoveChild classes. For example, the following lines add a Button child control to the v1 VBox control as part of a view state:

  ```
  <mx:AddChild relativeTo="{v1}">
    <mx:Button label="New Button"/>
  </mx:AddChild>
  ```

  For more information, see "Adding and removing components by using overrides" on page 1005.

- Setting or changing event handlers by using the SetEventHandler class. The following line sets the `click` event handler of the button1 Button control as part of a view state:

```
<mx:SetEventHandler target="{button1}" name="click"
  handler="newClickHandler()"/>
```

  For more information, see "Setting overrides on event handlers" on page 1014.
- Using a custom class that you define by implementing the IOverride interface. For information on creating and using custom overrides, see "Creating your own override classes" on page 1026.

Applying view states

You can set the view state of any Flex component, including the Application container, by using the component's `currentState` property. A component can have different view states at different times, but is in only one view state at a time. To specify the base view state, set the `currentState` property to `""`, an empty String.

You can also change a component's view state by calling the `setCurrentState()` method of the UIComponent class. Use this method when you do *not* want to apply a transition that you have defined between two view states. For more information on transitions, see Chapter 28, "Using Transitions," on page 1015.

The following example code lets the user change between two view states by clicking Button controls; button b1 sets the view state to the newButton state, and button b2 sets the view state to the base state.

```
<mx:Button id="b1" label="Add a Button" click="currentState='newButton';"/>
<mx:Button id="b2" label="Remove Added Button" click="currentState='';"/>
```

Example: Creating a simple view state

The following example shows an application with a base view state, and one additional view state named "NewButton":

```
<?xml version="1.0"?>
<!-- states\StatesSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <!-- Define the new view state. -->
        <mx:State name="NewButton">

            <!-- Add a new child control to the VBox. -->
            <mx:AddChild relativeTo="{v1}">
                <mx:Button id="b2" label="New Button"/>
            </mx:AddChild>

            <!-- Disable Button b1. -->
            <mx:SetProperty target="{b1}"
                name="enabled" value="false"/>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1">
        <mx:Button id="b1"/>

        <!-- Define Button control to switch to NewButton view state. -->
        <mx:Button label="Change to NewButton state"
            click="currentState = 'NewButton';"/>

        <!-- Define Button control to switch to default view state. -->
        <mx:Button label="Change to base view state"
            click="currentState = '';"/>
    </mx:VBox>
</mx:Application>
```

In the preceding example, the `<mx:AddChild>` tag adds a child to the v1 VBox control, and the `<mx:SetProperty>` tag disables the b1 Button control. A single view state can modify multiple components. The following example changes the `enabled` property for two Button controls, setting one `false`, and the other `true`:

```
<?xml version="1.0"?>
<!-- states\StatesSimple2Buttons.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewButton">
            <mx:AddChild relativeTo="{v1}">
                <mx:Button id="b3" label="New Button"/>
            </mx:AddChild>

            <!-- Disable Button b1, enable Button b2. -->
            <mx:SetProperty target="{b1}"
                name="enabled" value="false"/>
            <mx:SetProperty target="{b2}"
                name="enabled" value="true"/>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1">
        <mx:Button id="b1"/>
        <mx:Button id="b2" enabled="false"/>

        <!-- Define Button control to switch to NewButton view state. -->
        <mx:Button label="Change to NewButton state"
            click="currentState = 'NewButton';"/>

        <!-- Define Button control to switch to default view state. -->
        <mx:Button label="Change to base view state"
            click="currentState = '';"/>
    </mx:VBox>
</mx:Application>
```

Basing a view state on another view state

By default, a view state specifies a set of overrides that define the changes from the base view state to the new view state. However, you might have a set of view states that build on each other, where the first view state defines changes relative to the base view state, and the second view state defines changes relative to the first view state, rather than to the base view state.

To define the view state relative to another view state rather than to the base view state, use the `State.basedOn` property. When Flex changes to the new view state, it restores the base state, applies any changes from the state determined by the `basedOn` property, and then applies the changes defined in the new state.

In the following example, you define a NewButton view state that adds a Button control to the application, and a NewButton2 view state based on the NewButton state that adds another Button control:

```
<?xml version="1.0"?>
<!-- states\StatesSimpleBasedOn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewButton">
            <mx:AddChild relativeTo="{v1}">
                <mx:Button id="b2" label="New Button B2"/>
            </mx:AddChild>

            <mx:SetProperty target="{b1}"
                name="enabled" value="false"/>
        </mx:State>

        <!-- Define a view state based on the NewButton state. -->
        <mx:State name="NewButton2" basedOn="NewButton">
            <mx:AddChild relativeTo="{v1}">
                <mx:Button id="b3" label="New Button B3"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1">
        <mx:Button id="b1" label="Initial Button"/>

        <mx:Button label="Change to NewButton state"
            click="currentState = 'NewButton';"/>

        <mx:Button label="Change to New Button 2 State"
            click="currentState = 'NewButton2';"/>

        <mx:Button label="Change to base view state"
            click="currentState = '';"/>
    </mx:VBox>
</mx:Application>
```

Using view state events

When a component's `currentState` property changes, the State object for the states being exited and entered dispatch the following events:

enterState Dispatched when a view state is entered, but not fully applied. Dispatched by a State object after it has been entered, and by a component after it returns to the base view state.

exitState Dispatched when a view state is about to be exited. It is dispatched by a State object before it is exited, and by a component before it exits the base view state.

The component on which you modify the `currentState` property to cause the state change dispatches the following events:

currentStateChanging Dispatched when the view state is about to change. It is dispatched by a component after its `currentState` property changes, but before the view state changes. You can use this event to request any data from the server required by the new view state.

currentStateChange Dispatched after the view state has completed changing. It is dispatched by a component after its `currentState` property changed. You can use this event to send data back to a server indicating the user's current view state.

Creating a view state in ActionScript

This section describes how to create a view state in ActionScript. In the State class, the `overrides` property contains an Array of overrides that define the view state. The `overrides` property is the default property for the States class, so you can omit it in MXML. However, you must specify it in ActionScript.

To create a state in ActionScript:

1. Import the classes of the mx.states package.
2. Declare a State variable.
3. Create a function that does the following
   a. Instantiate a new State object
   b. Assign the object to the variable you defined in Step 2.
   c. Set the state name.
   d. Declare variables for the override class objects and assign them to new instances of the classes (such as SetProperty).
   e. Specify the properties of the override instances
   f. Add the override instances to the State object's `overrides` array.
   g. Add the state to the `states` array.

4. Call the function as part of an application or custom component initialization event handler, typically in response to the `initialize` event.

The following example uses ActionScript to create the Login Form that you created in MXML in the section “Example: Login form application” on page 986:

```
<?xml version="1.0"?>
<!-- states\StatesAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    verticalAlign="middle"
    initialize="createState();">

    <mx:Script>
        <![CDATA[

            import mx.states.*;
            import mx.controls.Button;

            // Variables for the ActionScript-defined view state.
            private var newState:State;
            private var changePropPanel:SetProperty;
            private var changePropButton:SetProperty;
            private var newFormItem:AddChild;
            private var newLinkButton:AddChild;
            private var oldLinkButton:RemoveChild;

            // Variables for the new components.
            private var fi:FormItem;
            private var ti:TextInput;
            private var lb:LinkButton;

            // Initialization method to
            // create a view state in ActionScript.
            private function createState():void {

                // Configure the AddChild class for the TextInput.
                fi = new FormItem();
                fi.label = "Confirm";
                ti = new TextInput();
                fi.addChild(ti);
                newFormItem = new AddChild();
                newFormItem.relativeTo = loginForm;
                newFormItem.target = fi;

                // Configure the Panel SetProperty class.
                changePropPanel = new SetProperty();
                changePropPanel.name ="title";
                changePropPanel.value = "Register";
                changePropPanel.target = loginPanel;

                // Configure the Button SetProperty class.
                changePropButton = new SetProperty();
                changePropButton.name ="label";
```

```
            changePropButton.value = "Register";
            changePropButton.target = loginButton;

            // Configure the RemoveChild class.
            oldLinkButton = new RemoveChild();
            oldLinkButton.target = registerLink;

            // Configure the AddChild class for the LinkButton.
            lb = new LinkButton();
            lb.label = "Return to Login";
            lb.addEventListener('click', newLBClick);
            newLinkButton = new AddChild();
            newLinkButton.relativeTo = spacer1;
            newLinkButton.position = "before";
            newLinkButton.target = lb;

            // Create an instance of the State class and
            // populate it with the view state overrides.
            newState = new State();
            newState.name = "Register";
            newState.overrides.push(newFormItem);
            newState.overrides.push(changePropPanel);
            newState.overrides.push(changePropButton);
            newState.overrides.push(oldLinkButton);
            newState.overrides.push(newLinkButton);

            // Add the view state to the states Array.
            states = new Array();
            states.push(newState);
        }

        private function newLBClick(event:Event):void {
            currentState="";
        }
    ]]>
</mx:Script>

<mx:Panel id="loginPanel"
    title="Login"
    horizontalScrollPolicy="off"
    verticalScrollPolicy="off">

    <mx:Form id="loginForm">
        <mx:FormItem label="Username:">
            <mx:TextInput/>
        </mx:FormItem>
        <mx:FormItem label="Password:">
            <mx:TextInput/>
        </mx:FormItem>
    </mx:Form>
```

```
        <mx:ControlBar>
      <!-- Use the LinkButton to change to the Register view state.-->
            <mx:LinkButton id="registerLink"
                label="Need to Register?"
                click="currentState='Register'"/>
            <mx:Spacer width="100%" id="spacer1"/>
            <mx:Button label="Login" id="loginButton"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

Defining view state overrides

When you define a view state, you specify the changes, or overrides, from the base state to the new view state, or from any other view state to the new view state. As part of the view state, you can define the following overrides:

- Set the value of object properties
- Set the value of component styles
- Add or remove components
- Change the event handler assigned to an event

This section describes these actions in more detail.

If you require overrides in addition to the ones supplied by Flex, you can define your own custom override class. For more information, see “Creating your own override classes” on page 1026.

Setting overrides on component properties

You use the SetProperty class to set a property value that is in effect during a specific view state. For example, the following code sets properties of a Panel container and of a Button control when you switch to the Register view state:

```
<?xml version="1.0"?>
<!-- states\StatesSetProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="Register">
            <mx:SetProperty
                target="{loginPanel}"
                name="title" value="Register"/>
            <mx:SetProperty
                target="{loginButton}"
                name="label" value="Register"/>
        </mx:State>
    </mx:states>

    <mx:Panel id="loginPanel"
        title="Login"
        horizontalScrollPolicy="off"
        verticalScrollPolicy="off">

        <mx:Form id="loginForm">
            <mx:Button label="Login" id="loginButton"/>
        </mx:Form>

        <mx:ControlBar width="100%">
            <mx:Button label="Change State"
              click="currentState =
                currentState=='Register' ? '':'Register';"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

You can use data binding to specify information to the `value` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- states\StatesSetProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define a variable in ActionScript.
            [Bindable]
            public var registerValue:String="Register";
        ]]>
    </mx:Script>

    <mx:states>
        <mx:State name="Register">
            <mx:SetProperty
                target="{loginPanel}"
                name="title" value="{registerValue}"/>
            <mx:SetProperty
                target="{loginButton}"
                name="label" value="{registerValue}"/>
        </mx:State>
    </mx:states>

    <mx:Panel id="loginPanel"
        title="Login"
        horizontalScrollPolicy="off"
        verticalScrollPolicy="off">

        <mx:Form id="loginForm">
            <mx:Button label="Login" id="loginButton"/>
        </mx:Form>

        <mx:ControlBar width="100%">
            <mx:Button label="Change State"
              click="currentState =
                currentState=='Register' ? '':'Register';"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

When you switch to the Register state, the `value` property of the SetProperty class is determined by the current value of the registerValue variable. However, this binding occurs only when you switch to the Register view state; if you modify the value of the registerValue variable after switching to the Register view state, the `title` property of the target component is not updated.

Setting overrides on component styles

You use the SetStyle class to set a style property value that is in effect only during a specific view state. In the following example, you change the background color of the Panel container, and the text color of the Button control as part of changing to the Register view state:

```
<?xml version="1.0"?>
<!-- states\StatesSetStyle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="Register">
            <mx:SetProperty
                target="{loginPanel}"
                name="title" value="Register"/>
            <mx:SetStyle
                target="{loginPanel}"
                name="backgroundColor" value="0x00FFFF"/>

            <mx:SetProperty
                target="{loginButton}"
                name="label" value="Register"/>
            <mx:SetStyle
                target="{loginButton}"
                name="color" value="0xFFFF00"/>
        </mx:State>
    </mx:states>

    <mx:Panel id="loginPanel"
        title="Login"
        horizontalScrollPolicy="off"
        verticalScrollPolicy="off">

        <mx:Form id="loginForm">
            <mx:Button label="Login" id="loginButton"/>
        </mx:Form>

        <mx:ControlBar width="100%">
            <mx:Button label="Change State"
              click="currentState =
                currentState=='Register' ? '':'Register';"/>
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

Adding and removing components by using overrides

You use the AddChild and RemoveChild classes to add and remove components as part of a switch of view state. When you remove a component by using the RemoveChild class, you remove the component from the application's display list, which means that it no longer appears on the screen. Even though the component is no longer visible, the component still exists and you can access it from within your application. For more information, see "Removing a child component" on page 1005.

When you add a component as part of a change of view state by using the AddChild class, you can either create the component before the first switch to the view state, or create it at the time of the first switch. If you create the component before the first switch, you can access the component from within your application even though you have not yet switched view states. If you create the component when you perform the first switch to the view state, you cannot access it until you perform that first switch.

Regardless of when you create a component by using the AddChild class, the component remains in memory after you switch out of the view state that creates it. Therefore, after the first switch to a view state, you can always access the component even if that view state is no longer the current view state. For more information, see "Controlling when to create added children" on page 1007.

Removing a child component

You use the RemoveChild class to remove a child component from a container as part of a change of view state. When you remove a child, you can use the `target` property to specify the component to remove. Removing a child does not delete it, so you can redisplay it later without recreating it.

For an example using the RemoveChild class, see "Example: Login form application" on page 986.

Adding a child component

You use the AddChild class to add a child component to a container as part of a change of view state. When you add a child, you can use the `relativeTo` property to specify the container relative to which you are placing the new child; the default value is the application or custom component that defines the view state. For example, to add a Button control to a HBox container, you specify the container as the value of the `relativeTo` property.

You can use the `position` property to specify the child's location within the container. Valid values are `before`, `after`, `firstChild`, and `lastChild`. The default value is `lastChild`.

The following example adds a Button control as the first child of an HBox container named h1:

```
<?xml version="1.0"?>
<!-- states\StatesAddRelative.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewButton">
            <mx:AddChild relativeTo="{h1}" position="firstChild">
                <mx:Button id="buttonNew" label="New Button"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:HBox id="h1">
        <mx:Button label="Change State"
            click=
             "currentState = currentState=='NewButton' ? '':'NewButton';"/>
    </mx:HBox>
</mx:Application>
```

You can add multiple child components, or a container that contains multiple child components, as the following example shows:

```
<?xml version="1.0"?>
<!-- states\StatesAddRelativeMultiple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewVBox">
            <mx:AddChild relativeTo="{v1}">
                <mx:VBox id="v2">
                    <mx:Button id="buttonNew1" label="New Button"/>
                    <mx:Button id="buttonNew2" label="New Button"/>
                    <mx:Button id="buttonNew3" label="New Button"/>
                </mx:VBox>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1">
      <mx:Button label="Change State"
        click=
          "currentState = currentState=='NewVBox' ? '':'NewVBox';"/>
    </mx:VBox>
</mx:Application>
```

In the previous example, you use a single `<mx:AddChild>` tag to add the VBox container and its children. The following example performs the same action, but uses multiple `<mx:AddChild>` tags:

```
<?xml version="1.0"?>
<!-- states\StatesAddRelativeMultipleAlt.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewVBox">
            <mx:AddChild relativeTo="{v1}">
                <mx:VBox id="v2"/>
            </mx:AddChild>
            <mx:AddChild relativeTo="{v2}">
                <mx:Button id="buttonNew1" label="New Button"/>
            </mx:AddChild>
            <mx:AddChild relativeTo="{v2}">
                <mx:Button id="buttonNew2" label="New Button"/>
            </mx:AddChild>
            <mx:AddChild relativeTo="{v2}">
                <mx:Button id="buttonNew3" label="New Button"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1">
      <mx:Button label="Change State"
        click=
         "currentState = currentState=='NewVBox' ? '':'NewVBox';"/>
    </mx:VBox>
</mx:Application>
```

Controlling when to create added children

By default Flex creates container children when they are first required as part of a change of view state. However, if a child requires a long time to create, users might see a delay when the view state changes. Therefore, you can choose to create the child before the state changes to improve your application's apparent speed and responsiveness.

Flex lets you choose among three ways of creating a child component added by a view state:

- Automatically create it when it is first needed by a change of view state.
- Automatically create it when the application first loads.
- Explicitly create the child when you want it.

The specification of when the child is created is called its *creation policy.* For more general information on creation policies and controlling how children are created, see Chapter 6, "Improving Startup Performance," in *Building and Deploying Flex 2 Applications*.

The AddChild class has two mutually exclusive properties that you use to specify the child to add and the child's creation policy:

`target` Always creates the child when the applications starts. You cannot change the creation policy when using this property. When you use this property, you can access the component before the first change to the view state that defines it.

`targetFactory` Lets you control the creation policy of the child.

About the target property

The `target` property specifies the component to add. When you use this property, Flex creates the child when the applications starts. You cannot use any other creation policy with this property.

One use of the `target` property is to modify components that already exist in your application. For example, if you want to move a component from one parent container to another, you use the RemoveChild class to remove the child from its parent container, and then use the AddChild class to add the component to a different container. Since the component already exists in the application, you do not have to worry about its creation policy, as the following example shows to move a Button control from one VBox container to another:

```
<?xml version="1.0"?>
<!-- states\StatesAddRemove.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:states>
        <mx:State name="NewParent">
            <mx:RemoveChild target="{button1}"/>
            <mx:AddChild target="{button1}" relativeTo="{v2}"/>
        </mx:State>
    </mx:states>

    <mx:VBox id="v1" borderStyle="solid">
        <mx:Label text = "VBox v1"/>
        <mx:Button id="button1"/>
    </mx:VBox>

    <mx:VBox id="v2" borderStyle="solid">
        <mx:Label text = "VBox v2"/>
        <mx:Button label="Change Parent"
            click="currentState =
                currentState=='NewParent' ? '':'NewParent';"/>
    </mx:VBox>
</mx:Application>
```

About the targetFactory property

The `targetFactory` property is the default property of the AddChild class, so if you specify the child by using an MXML tag in the `<mx:AddChild>` tag body, as shown in the following code example, Flex automatically uses the `targetFactory` property and creates the required factory class:

```
<mx:AddChild relativeTo="{v1}">
  <mx:Button id="b0" label="New Button"/>
</mx:AddChild>
```

Because `targetFactory` property is the default property of the AddChild class, the previous example is equivalent to the following:

```
<mx:AddChild relativeTo="{v1}">
  <mx:targetFactory>
    <mx:Button id="b0" label="New Button"/>
  </mx:targetFactory>
</mx:AddChild>
```

You can specify either of the following items to the `targetFactory` property:

- A Flex component, (that is, any class that is a subclass of the UIComponent class), such as the Button control. If you use a Flex component, the Flex compiler automatically wraps the component in a factory class.
- A factory class that implements the IDeferredInstance interface and creates the child instance or instances. "Example: Adding a Button control using the IDeferredInstance interface" on page 1013.

When you use the `targetFactory` property to specify the child, you can optionally use the `creationPolicy` property to specify the creation policy. The `creationPolicy` property supports the following values:

`auto` Creates the child instance when it is first added by a change to the view state. This is the default value. In ActionScript, specify this property by using the `mx.core.ContainerCreationPolicy.AUTO` constant.

`all` Creates the child instance at application startup. In ActionScript, specify this property by using the `ContainerCreationPolicy.ALL` constant. This is equivelant to using the `target` property.

`none` Requires your application to call the `AddChild.createInstance()` method to create an instance of the child. In ActionScript, specify this property by using the `mx.core.ContainerCreationPolicy.NONE` constant.

You can call the `createInstance()` method with any creation policy, but if the child has already been created, it does nothing. For example, if you have a `creationPolicy` value of `auto`, but need to access a child before the view state is entered, you can call the `createInstance()` method to ensure that the child has been created.

The following example adds a child and has Flex create a child when the application first starts. The application does not display the child until the view state with the AddChild tag is activated.

```
<mx:AddChild relativeTo="{v1}" position="lastChild" creationPolicy="all">
  <mx:Button id="b0" label="New Button"/>
</mx:AddChild>
```

Example: Using different creation policies

The following example lets you change between a base view state, a view state that adds a Button control created at application startup, and a view state that adds a Button control created using the `AddChild.createInstance()` method:

```
<?xml version="1.0"?>
<!-- states\StatesCreationPolicy.mxml  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initButton();">

    <mx:Script>
        <![CDATA[

            // Because the cpAll view state creates the Button control
            // at application startup, you can access the control to
            // set the label before the first switch
            // to the cpAll view state.
            public function initButton():void {
                newButton.label="cpAll Button";
            }
        ]]>
    </mx:Script>

    <mx:states>
        <!-- Create the Button control at application startup. -->
        <mx:State name="cpAll">
            <mx:AddChild relativeTo="{myPanel}" creationPolicy="all">
                <mx:Button id="newButton"/>
            </mx:AddChild>
        </mx:State>

        <!-- Create the Button control when you want to create it. -->
        <mx:State name="cpNone">
            <mx:AddChild id="noCP"
                relativeTo="{myPanel}" creationPolicy="none">
                <mx:Button label="cpNone button"/>
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <mx:Panel id="myPanel"
        title="Static and dynamic states"
        width="300" height="150">

        <!-- Change to the cpAll view state. -->
        <mx:Button label="Change to cpAll state"
           click="currentState = currentState == 'cpAll' ? '' : 'cpAll';"/>

        <!-- Create the Button control for the noCP state that
```

```
            uses creationPolicy=none.
            If you do not click this button before changing to the
            cpNone view state, the view state does nothing
            because the Button control does not exist. -->
        <mx:Button label="Explicitly create a button control"
            click="noCP.createInstance();"/>

        <!-- Change to the cpNone view state. -->
        <mx:Button label="Change to noCP state"
         click="currentState = currentState == 'cpNone' ? '' : 'cpNone';"/>

    </mx:Panel>
</mx:Application>
```

Example: Adding a Button control using the IDeferredInstance interface

The following example uses DeferredInstanceFromFunction, a subclass of DeferredInstance, to create the child instance for the AddChild class:

```
<?xml version="1.0"?>
<!-- states\StatesDefInstan.mxml  -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.core.*;
            import mx.states.*;

            [Bindable]
            public var defInst:DeferredInstanceFromFunction =
                new DeferredInstanceFromFunction(createMyButton);

            // Function to create a new Button control.
            public function createMyButton():Object {
                var newButton:Button = new Button();
                newButton.label = "New Dynamic Button";
                return newButton;
            }
        ]]>
    </mx:Script>

    <mx:states>
        <mx:State name="deferredButton">
            <mx:AddChild relativeTo="{myPanel}" targetFactory="{defInst}"/>
        </mx:State>
    </mx:states>

    <mx:Panel id="myPanel"
        title="Static and dynamic states"
        width="300" height="150">

        <mx:Button id="myButton2"
            label="Toggle Deferred Button"
            click="currentState =
               currentState == 'deferredButton' ? '' : 'deferredButton';"/>
    </mx:Panel>
</mx:Application>
```

Setting overrides on event handlers

Flex lets you define state-specific event handlers in your application. A state-specific event handler is only active during a specific view state. For example, you might define a Button control that uses one event handler in the base view state, but uses a different event handler when you change view state.

You can set state-specific event handlers in either of two ways:

- In ActionScript or MXML, use the `handlerFunction` property of the SetEventHandler class to specify the event handler.
- In MXML use the `handler` event of the SetEventHandler class to specify the event handler.

When using the SetEventHandler class, any existing event handler that is defined in MXML is removed from the target component. The following example defines a view state that adds an event handler to a Button control:

```
<mx:states>
  <mx:State name="State1">
  <mx:SetEventHandler target="{b1}"
    name="click" handler="trace('goodbye');"/>
</mx:states>
...
<mx:Button id="b1" click="trace('hello');"/>
```

In this example, the `trace('hello');` handler is removed when the `trace('goodbye');` handler is added as part of the State 1 view state. However, if you add an event handler in ActionScript by calling the `addEventListener()` method, the event handler is not removed by the SetEventHandler class.

Using the handlerFunction property

You can use the `handlerFunction` property in ActionScript or MXML. It lets you specify an event handler that must take a single argument of type flash.events.Event. The following example code sets an event handler as part of a view state:

```
<mx:SetEventHandler target="{myButton1}"
  name="click" handlerFunction="handler2"/>
```

You use the `SetEventHandler.name` property to specify the event name for the associated event handler. Notice that you only specify the name of the event handler to the `handlerFunction` property because that event handler is required to take a single argument of type flash.events.Event. Therefore, the function handler2 must have the following declaration:

```
private function handler2(event:Event):void
{
```

```
    ...
}
```

Using the handler event type

The `handler` event has the following advantages over the `handlerFunction` property:

- You can specify ActionScript code directly in the `<mx:SetEventHandler>` tag, without having to define a separate function for the event handler. This technique is useful for very simple event handlers, such as when you want to pop up an Alert dialog box.
- The event handler can take multiple arguments, of any type. If you use the `handlerFunction` property, the handler function can take only an Event object as an argument.

You can use the `handler` event in MXML only.

You use the `SetEventHandler.name` property to specify the event name for the associated event handler. The following example shows how to use the `handler` property to specify the event handler code directly in the `<mx:SetEventHandler>` tag for the `click` event of a Button control:

```
<mx:SetEventHandler target="{myButton1}"
  name="click" handler="Alert.show('Hello World.')"/>
```

The following example shows how you specify a handler function that takes two arguments; the Event object and a user ID String variable:

```
<mx:SetEventHandler target="{myButton1}"
  name="click" handler="myAlert(event, userID)"/>
```

The function myAlert must have the following declaration:

```
private function myAlert(eventObj:Event, idString:String):void
{
    ...
}
```

Example: Setting event handlers

The following example creates the base view state and three additional view states. Each state uses the SetEventHandler class to define a different event handler for the `click` event for the Button control named b1:

```
<?xml version="1.0"?>
<!-- states\StatesEventHandlersSimple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            // Base view state click event handler.
            private function handlerBase():void {
              ta1.text = "Default click handler for the base view state";
            }

            // View state 1 click event handler.
            private function handlerState1(event:Event):void {
              ta1.text =
                "Use the handlerFunction property to specify the function.";
            }

            // View state 3 click event handler.
            private function handlerState3(theString:String):void {
              ta1.text =
                "Use the handler property to specify the function" +
                "\nArgument String: " + theString;
            }
        ]]>
    </mx:Script>

    <mx:states>
        <mx:State name="State1">
            <mx:SetEventHandler target="{b1}"
                name="click"
                handlerFunction="handlerState1"/>
            <mx:SetProperty target="{b1}"
                name="label" value="Click Me: State 1"/>
        </mx:State>

        <mx:State name="State2">
            <mx:SetEventHandler target="{b1}"
                name="click"
                handler="ta1.text='Specify an inline event listener';"/>
            <mx:SetProperty target="{b1}"
                name="label" value="Click Me: State 2"/>
        </mx:State>
```

```
        <mx:State name="State3">
            <mx:SetEventHandler target="{b1}"
              name="click"
              handler="handlerState3('Event listener arg');"/>
            <mx:SetProperty target="{b1}"
                name="label" value="Click Me: State 3"/>
        </mx:State>
    </mx:states>

    <mx:Button id="b1"
        label="Click Me: Base State"
        click="handlerBase();"/>

    <mx:TextArea id="ta1" height="100" width ="50%"/>

    <mx:VBox>
        <mx:Button
            label="Set state 1, use handlerFunction property"
            click="currentState='State1';"/>
        <mx:Button
            label="Set state 2, specify inline handler"
            click="currentState='State2';"/>
        <mx:Button
            label="Set state 3, specify handler function"
            click="currentState='State3';"/>
        <mx:Button
            label="Set base state"
            click="currentState='';"/>
    </mx:VBox>
</mx:Application>
```

Defining view states in custom components

If one or more view states apply to a single component or set of components, you define a custom component that comprises this component or components, and specify the view states in the component definition, rather than at the Application tag level. For example, if a view state adds a Button control to an HBox container, consider making the HBox a custom component and defining a state in the component that adds the button. You should consider doing this particularly if you have multiple states that do nothing but modify the HBox container.

Similarly, if a custom component has multiple view states, define the view states in the component code, not in the main application. For example, if a custom TitleWindow component has a collapsed view state and a expanded view state, you should define these view states in the panel MXML component, not in the main application file.

This way, you segregate the view state definitions into the specific components to which they apply. A item renderer is one example of good use of a multiple view state MXML component.

The following example shows a custom component for a TitleWindow component that has two view states, collapsed and expanded:

```
<?xml version="1.0"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
    close="checkCollapse();" headerHeight="40" showCloseButton="true">

    <mx:Script>
        <![CDATA[

            // Skins for the close button when the
            // TitleWindow container is collapsed.
            [Embed(source="closeButtonUpSkin.jpg")]
            [Bindable]
            public var closeBUp:Class;

            [Embed(source="closeButtonDownSkin.jpg")]
            [Bindable]
            public var closeBDown:Class;

            [Embed(source="closeButtonOverSkin.jpg")]
            [Bindable]
            public var closeBOver:Class;

            private function checkCollapse():void {
                currentState =
                    currentState == "collapsed" ? "" : "collapsed";
            }
    ]]>
    </mx:Script>

    <mx:states>
        <mx:State name="collapsed">
            <mx:SetProperty
                name="height"
                value="{getStyle('headerHeight')}"/>
            <mx:SetStyle
                name="closeButtonUpSkin"
                value="{closeBUp}"/>
            <mx:SetStyle
                name="closeButtonDownSkin"
                value="{closeBDown}"/>
            <mx:SetStyle
                name="closeButtonOverSkin"
                value="{closeBOver}"/>
        </mx:State>
    </mx:states>
</mx:TitleWindow>
```

This example replaces the default close icon for a TitleWindow when the component is in the collapsed state.

You can then use this component in an application, as the following example shows:

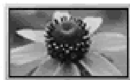
```
<?xml version="1.0"?>
<!-- states\StatesTitleWindowMain.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:MyComp="myComponents.*">

    <MyComp:StateTitleWindow id="myP" title="My Application">
        <mx:HBox width="100%">
            <mx:Button/>
            <mx:TextArea/>
        </mx:HBox>

        <mx:ControlBar width="100%">
            <mx:Label text="Quantity"/>
            <mx:NumericStepper/>
            <!-- Use Spacer to push Button control to the right. -->
            <mx:Spacer width="100%"/>
            <mx:Button label="Add to Cart"/>
        </mx:ControlBar>
    </MyComp:StateTitleWindow>
</mx:Application>
```

Using view states with a custom item renderer

A shopping application that displays multiple items on a page might have a custom thumbnail item renderer with two view states. In the base view state, the item cell might look the following image:



When the user rolls the mouse over the item, the view state changes: the thumbnail no longer has the availability and rating information, but now has buttons that let the user get more information or add the item to the wish list or cart. In the new state, the cell also has a border and a drop shadow, as the following image shows:

In this example, the application item renderer's two view states have different child components and have different component styles. The summary state, for example, includes an availability label and a star rating image, and has no border. The rolled-over state replaces the label and rating components with three buttons, and has an outset border.

For information on item renderers, see Chapter 21, "Using Item Renderers and Item Editors," on page 819.

Example: Using view states with a custom item renderer

The following code shows an application that uses a custom item renderer to display catalog items. When the user moves the mouse over an item, the item renderer changes to a state where the picture is slightly enlarged, the price appears in the cell, and the application's text box shows a message about the item. All changes are made by the item renderer's state, including the change in the parent application.

```
<?xml version="1.0"?>
<!-- states\StatesRendererMain.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="450" height="500">
    <mx:Script>
        <![CDATA[
            import mx.controls.listClasses.*;
            import mx.controls.Image;
            import mx.collections.ArrayCollection;

            //The data to display in the TileList control.
            [Bindable]
            public var catArray:ArrayCollection = new ArrayCollection([
                { name: "USB Watch",
                  data: "1",
                  price: "129.99",
                  image: "assets/usbwatch.jpg",
                  description: "So, you need to tell the time of course" },
                { name: "007 Digital Camera",
                  data: "1",
                  price: "99.99",
                  image: "assets/007camera.jpg",
                  description: "Just like 007 used" },
                { name: "2-Way Radio Watch",
                  data: "1",
                  price: "49.99",
                  image: "assets/radiowatch.jpg",
                  description: "Better than Dick Tracy's" },
                { name: "USB Desk Fan",
                  data: "1",
                  price: "19.99",
                  image: "assets/usbfan.jpg",
                  description: "Computer-powered cool!!!"},
            ]);
        ]]>
    </mx:Script>

    <mx:TileList id="myList"
        dataProvider="{catArray}"
        columnWidth="150"
        rowHeight="150"
        width="310"
```

```
        height="310"
        itemRenderer="myComponents.ImageComp"/>

    <!-- The item renderer fills in the TextArea control. -->
    <mx:HBox>
        <mx:Label text="Description"/>
        <mx:TextArea id="t1"
            width="200" height="100"
            wordWrap="true"/>
    </mx:HBox>
</mx:Application>
```

The following code defines the item renderer, in the file imagecomp.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- states\myComponents\ImageComp.mxml
    When the mouse pointer goes over a cell,
        this component changes its state to showdesc.
    When the mouse pointer goes out of the cell,
        the component returns to the base state. -->

<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    horizontalAlign="center"
    verticalAlign="top"
    mouseOver="currentState='showdesc'"
    mouseOut="currentState=''">

    <!-- In the base state, display the image and the label. -->
    <mx:Image id="img1"
        source="{data.image}"
        width="75"
        height="75"/>
    <mx:Label text="{data.name}"/>

    <mx:states>
        <mx:State name="showdesc">
          <!-- In the showdesc state, add the price, make the image bigger,
            and put the description in the parent application's TextArea.-->
            <mx:AddChild>
                <mx:Text text="{data.price}"/>
            </mx:AddChild>
            <mx:SetProperty target="{img1}" name="width" value="85"/>
            <mx:SetProperty target="{img1}" name="height" value="85"/>
            <mx:SetProperty target="{parentApplication.t1}" name="text"
                value="{data.description}"/>
        </mx:State>
    </mx:states>
</mx:VBox>
```

Using view states with history management

The Flex History Manager lets users navigate through a Flex application by using the web browser's back and forward navigation commands. The History Manager can track when the application enters a state so that users can use the browser to navigate between states, such as states that correspond to different stages in an application process.

To enable history management to track application states, your application must do the following:

- Implement the IHistoryState interface by defining `loadState` and `saveState` methods.
- Register the application with the History Manager.
- Each time the state changes, call the `HistoryManager.save` method.

The following code shows how you can code a search interface so that the History Manager keeps track of your search. For more information on history management, including another example of tracking states, see Chapter 32, "Using the History Manager," on page 1111.

```
<?xml version="1.0"?>
<!-- states\StatesHistoryManager.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    implements="mx.managers.IHistoryManagerClient"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
        import mx.managers.HistoryManager;

        /////////////////////////
        // IHistoryState methods
        /////////////////////////
        // Restore the state and searchString value.
        public function loadState(state:Object):void {
            if (state) {
                currentState = state.currentState;
                searchString = searchInput.text = state.searchString;
            }
            else {
                currentState = '';
            }
        }
        // Save the current state and the searchString value.
        public function saveState():Object {
            var state:Object = {};
            state.currentState = currentState;
            state.searchString = searchString;
            return state;
        }

        /////////////////////////
        // App-specific scripts
        /////////////////////////
        // The search string value.
        [Bindable]
        public var searchString:String;

        // Register the application with the history manager
        // when the application is created.
        public function initApp():void {
            HistoryManager.register(this);
        }

        // The method for doing the search.
        // For the sake of simplicity it doesn't do any searching.
```

```
        // It does change the state to display the results box,
        // and save the new state in the history manager.
        public function doSearch():void {
            currentState = "results";
            searchString = searchInput.text;
            HistoryManager.save();
        }
        // Method to revert the state to the base state.
        // Saves the new state in the history manager.
        public function reset():void {
            currentState = '';
            searchInput.text = "";
            searchString = "";
            HistoryManager.save();
        }
        ]]>
    </mx:Script>

    <mx:states>
        <!-- The state for displaying the search results -->
        <mx:State name="results">
            <mx:SetProperty target="{p}" name="width" value="100%" />
            <mx:SetProperty target="{p}" name="height" value="100%" />
            <mx:SetProperty target="{p}" name="title" value="Results" />
            <mx:AddChild relativeTo="{searchFields}">
                <mx:Button label="Reset" click="reset()" />
            </mx:AddChild>
            <mx:AddChild relativeTo="{p}">
                <mx:Label text="Search results for {searchString}" />
            </mx:AddChild>
        </mx:State>
    </mx:states>

    <!-- In the base state, just show a panel
        with a search text input and button. -->
    <mx:Panel id="p" title="Search" resizeEffect="Resize">
        <mx:HBox id="searchFields" defaultButton="{b}">
            <mx:TextInput id="searchInput" />
            <mx:Button id="b" label="Go" click="doSearch();" />
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```

Creating your own override classes

Flex lets you create a view state by specifying overrides to add or remove child components, to set style and property values, or to set state-specific event handlers. However, your application may require you to define your own overrides in addition to the ones supplied by Flex.

You can define custom overrides by creating a class that implements the IOverride interface. You can then use your override class in the same way that you use AddChild, RemoveChild, SetProperty, and the other override classes. For example, you can create a class that adds a bitmap filter, such as a filter that blurs an object.

The IOverride interface contains the following methods:

Method	Description
`initialize()`	Initializes the override.
`apply()`	Saves the original value and applies the override.
`remove()`	Restores the original value.

The following example shows an AddBlur override class that applies the Flash BlurFilter class to blur the target component.

```
package myOverrides
{
    import flash.display.*;
    import flash.filters.*;
    import mx.core.*;
    import mx.states.*;

    /* State override that adds a Blur effect to a component. */
    public class AddBlur implements IOverride
    {
        /* Constructor. */
        public function AddBlur(
            target:DisplayObject = null)
        {
            this.target = target;
        }

        /* The object to blur. */
        public var target:DisplayObject;

        /* The initialize() method is empty for this example. */
        public function initialize():void   {
        }

        /* The apply() method adds a BlurFilter to the filters array. */
        public function apply(parent:UIComponent):void {
            var obj:DisplayObject = target ? target : parent;
            var filters:Array = obj.filters;

            filters.push(new BlurFilter());
            obj.filters = filters;
        }

        /* The remove() method removes the BlurFilter
           from the filters array. */
        public function remove(parent:UIComponent):void {
            var obj:DisplayObject = target ? target : parent;
            var filters:Array = obj.filters;

            filters.pop();
            obj.filters = filters;
        }
    }
}
```

You can then use your custom override class in an application, as the following example shows:

```
<?xml version="1.0"?>
<!-- states\AddBlurApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:myOverride="myOverrides.*">

    <mx:states>
        <mx:State name="State1">
            <myOverride:AddBlur target="{b1}"/>
        </mx:State>
    </mx:states>

    <mx:Button id="b1"
        label="Click Me: Base State"/>

    <mx:Button
        label="Set state 1"
        click="currentState='State1';"/>

    <mx:Button
        label="Set base state"
        click="currentState='';"/>
</mx:Application>
```