ADOBE® FLEX® 4

Features and Migration Guide

Contents

Chapter 1: New SDK Features

The following new features for the SDK are available with Flex 4:

Namespaces

The namespaces for Flex 4 applications are as follows:

```
xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:mx="library://ns.adobe.com/flex/mx"
xmlns:s="library://ns.adobe.com/flex/spark"
```

Each namespace corresponds to a component set. For example, the `xmlns` properties in the following `<s:Application>` tag indicate that tags corresponding to the Spark component set use the prefix *s:*

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

You can still use the 2006 namespace, but you cannot use Flex 4 features (such as the new components and layout schemes) with it.

Flex defines the following Universal Resource Identifiers (URI) for the Flex namespaces:

- `xmlns:fx="http://ns.adobe.com/mxml/2009"`

  The MXML language namespace URI. This namespace includes the top-level ActionScript language elements, such as Object, Number, Boolean, and Array. For a complete list of the top-level elements, see the Top Level package in the ActionScript 3.0 Reference for the Adobe Flash Platform.

  This namespace also includes the tags built into the MXML compiler, such as `<fx:Script>`, `<fx:Declarations>`, and `<fx:Style>` tags. For a list of the compiler elements, see the MXML Only Tags appendix in the ActionScript 3.0 Reference for the Adobe Flash Platform.

  This namespace does not include the MX or Spark component sets.

  The complete list of top-level ActionScript language elements included in this namespace is defined by the frameworks\mxml-2009-manifest.xml manifest file in your Flex SDK installation directory. Note that this file does not list the MXML compiler tags be ca sue they are built into the MXML compiler.

- `xmlns:mx="library://ns.adobe.com/flex/mx"`

  The MX component set namespace URI. This namespace includes all of the components in the Flex mx.* packages, the Flex charting components, and the Flex data visualization components.

  The complete list of elements included in this namespace is defined by the frameworks\mx-manifest.xml manifest file in your Flex SDK installation directory.

- `xmlns:s="library://ns.adobe.com/flex/spark"`

  The Spark component set namespace URI. This namespace includes all of the components in the Flex spark.* packages and the text framework classes in the flashx.* packages.

  This namespace includes the RPC classes for the WebService, HTTPService, and RemoteObject components and additional classes to support the RPC components. These classes are included in the `mx:` namespace, but are provided as a convenience so that you can also reference them by using the `s:` namespace.

  This namespace also includes several graphics, effect, and state classes from the mx.* packages. These classes are included in the `mx:` namespace, but are provided as a convenience so that you can also reference them by using the `s:` namespace.

  The complete list of elements included in this namespace is defined by the frameworks\spark-manifest.xml manifest file in your Flex SDK installation directory.

  The following table lists the classes from the mx.* packages included in this namespace:

Category	Class
RPC classes	mx.messaging.channels.AMFChannel
	mx.rpc.CallResponder
	mx.messaging.ChannelSet
	mx.messaging.Consumer
	mx.messaging.channels.HTTPChannel
	mx.rpc.http.mxml.HTTPService
	mx.messaging.Producer
	mx.rpc.remoting.mxml.RemoteObject
	mx.rpc.remoting.mxml.Operation
	mx.messaging.channels.RTMPChannel
	mx.messaging.channels.SecureAMFChannel
	mx.messaging.channels.SecureStreamingAMFChannel
	mx.messaging.channels.SecureHTTPChannel
	mx.messaging.channels.SecureStreamingHTTPChannel
	mx.messaging.channels.SecureRTMPChannel
	mx.messaging.channels.StreamingAMFChannel
	mx.messaging.channels.StreamingHTTPChannel
	mx.rpc.soap.mxml.WebService
	mx.rpc.soap.mxml.Operation
	mx.data.mxml.DataService
Graphics classes	mx.graphics.BitmapFill
	mx.geom.CompoundTransform
	mx.graphics.GradientEntry
	mx.graphics.LinearGradient
	mx.graphics.LinearGradientStroke
	mx.graphics.RadialGradient
	mx.graphics.RadialGradientStroke
	mx.graphics.SolidColor
	mx.graphics.SolidColorStroke
	mx.graphics.Stroke
	mx.geom.Transform
Effect classes	mx.effects.Parallel
	mx.effects.Sequence
	mx.states.Transition
	mx.effects.Wait
States classes	mx.states.State
	mx.states.AddItems

Components

Flex 4 introduces a new set of components that take advantage of Flex 4 features. The components are known as Spark components.

***Note:** The MX component architecture was included in previous versions of Flex. MX components are still shipped with Flex 4 to support backward compatibility with previous versions of Flex.*

In many cases, the differences between the Spark and MX versions of the Flex components are not visible to you. They mainly concern the component's interaction with the Flex 4 layouts, as well as the architecture of the skins and states.

In some cases, the Spark component is different from the MX component. For example, the Spark Application and the MX Application components are different in several ways: the differences include the background colors and the default layout.

The following table lists the commonly-used components new to Flex 4:

Component	Component
Application	RadioButton
BorderContainer	RadioButtonGroup
Button	RichEditableText
ButtonBar	RichText
CheckBox	Scroller
DataGroup	SkinnableContainer
DataRenderer	SkinnableDataContainer
DropDownList	Spinner
Group	TextArea
HGroup	TextInput
HScrollBar	ToggleButton
HSlider	VGroup
Label	VideoPlayer
List	VScrollBar
NavigatorContent	VSlider
NumericStepper	Window
Panel	WindowedApplication

The components are contained in the spark.* package. See the *ActionScript 3.0 Reference for the Adobe Flash Platform* for more information.

Visual components have additional classes that define their skins. For example, in addition to the Button class, there is also a ButtonSkin class. The skin classes for components are typically in the spark.skins.* package.

You can use Spark components and MX components interchangeably in your applications. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- gumbonents/BasicGumbonentUsage.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
          <s:HorizontalLayout/>
     </s:layout>
     <s:Button label="Click Me"/>
     <mx:Button label="Click Me"/>
</s:Application>
```

Spark components should work within MX containers, and MX components should work within Spark containers.

***Note:** The direct children of an MX navigator container must be MX containers, either layout or navigator containers, or a Spark NavigatorContent container. You cannot directly nest a control or a Spark container other than the Spark NavigatorContent container in an navigator. To use a Spark container other than the NavigatorContent container as the child of a navigator, wrap it in an MX container or in the Spark NavigatorContent container.*

You can incrementally adopt Spark components in your applications, when and if they provide functionality or structure that MX components lack. No MX components have been removed from the Flex 4 framework.

All Spark components target Flash Player 10. If your application uses Spark components, your users must use Flash Player 10.

Component architecture

The biggest change to Flex 4 is the introduction of a new skinning and component architecture called Spark. Many components from the previous versions of Flex are reimplemented using the Spark component architecture.

The Spark component architecture has several goals:

- Define a clean separation between the functional logic of the component and its appearance.

  Spark components consists of two classes to support the separation of logic and appearance: an ActionScript class that defines a component's logic and an MXML skin class that defines its visual appearance. To change the component's logic, subclass the component class and add your own behavior. To change the component's appearance but not its logic, customize the skin.

  Flex 4 has also simplified the process of skinning a component. Instead of defining skins in ActionScript, the Spark architecture lets you define skins in MXML.

- Create a small set of basic components that are easily customized by developers.

  In previous versions of Flex, changing basic functionality of a component often meant reimplementing the component. For example, developers had to create a custom container class just to change the layout algorithm of the container.

  Spark containers use a swappable layout algorithm that lets you select the layout from a set of predefined layouts. Or, you can define your own custom layout without creating an entire custom container.

- Implement the Spark components on the same base classes as the components in the previous versions of Flex.

The new Spark component architecture is based on existing Flex classes, such as mx.core.UIComponent and mx.effects.Effect. Developers upgrading from previous versions of Flex are already familiar with these base classes, which makes the upgrade process simple. This means that you can mix components built with prior versions of Flex with the new Spark components in the same application.

Associated with a Spark component class is a skin class. The skin class manages everything related to the visual appearance of the component, including graphics, layout, data representation, skin parts, and view states. The skinning contract between a skin class and a component class defines the rules that each class must follow so that they can communicate with one another.

Text primitives

Flex text-based controls such as RichText and RichEditableText offer new functionality based on the new text object model in Flash Player 10. The text object model is defined by the Flash Text Engine (FTE) and Text Layout Framework (TLF) features.

TLF uses an ActionScript object model to represent rich text. Concepts like paragraphs, spans, and hyperlinks are not represented as formats that affect the appearance of character runs in a single, central text string. Instead they are represented by runtime-accessible ActionScript objects, with their own properties, methods, and events. The TLF classes are in the flashx.textLayout.* package.

FTE supports low-level text functionality such as rendering individual lines of text. This involves mapping Unicode characters to font glyphs, laying out the glyphs using the Unicode bidirectional text algorithm, determining appropriate line breaks, and rendering the glyphs into pixels. The FTE APIs are defined in the flash.text.engine.* package.

For more information about FTE and TLF, see Building the User Interface.

The new features supported by the Flex text-based components include the following:

- Columns
- Paragraph and character level attributes
- Kerning
- Transforms
- Masks and blend modes
- Whitespace handling
- Margins and indentations
- Direction (left to right and right to left)

The following table describes the new text components in the spark.controls.* package:

Class	Description
Label	The lightest-weight of the text classes. This class is similar to the MX Label class, except that it is completely non-interactive, but supports FTE.
RichText	The middle-weight component of the text primitives. This class is similar to the MX Text class. It can display richly formatted test with character and paragraphic formats, but it is non-interactive. It does not support user interactivity such as scrolling, selection, or editing.
RichEditableText	The heaviest-weight component of the text primitives. This class is similar to the TextArea class, except that it does not define a skin. It does support user interactivity such as scrolling, selection, and editing.

In addition to the text primitive classes described above, the Spark text classes include the TextInput and TextArea classes which have MX equivalents. These classes supporting skinning.

Language tags

Flex 4 introduces new language tags that you can use in your MXML files.

Declarations

Use the `<fx:Declarations>` tag to declare non-default, non-visual properties of the current class. These tags typically include effect, validator, formatter, and data service tags. This tag is also used for MXML-based custom components that declare default properties.

The following example defines two effects in the `<fx:Declarations>` tag:

```
<?xml version="1.0"?>
<!-- behaviors\SparkAnimateProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Animate id="scaleUp"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.0" valueTo="1.5"/>
        </s:Animate>
        <s:Animate id="scaleDown"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.5" valueTo="1.0"/>
        </s:Animate>
    </fx:Declarations>

    <s:Button id="myB1"
        label="Scale Button"
        mouseDown="scaleUp.end(); scaleUp.play();"
        mouseUp="scaleDown.end(); scaleDown.play();"/>
</s:Application>
```

You can declare visual children inside a `<fx:Declarations>` tag, but they are instantiated as if they were non-visual. You must manually add them to the display list with a call to the container's `addElement()` method (for Spark containers) or `addChild()` method (for MX containers).

Definition

Use one or more `<fx:Definition>` tags inside a `<fx:Library>` tag to define graphical children that you can then use in other parts of the application file.

An element in the `<fx:Definition>` tag is not instantiated or added to the display list until it is added as a tag outside of the `<fx:Library>` tag. The `<fx:Definition>` element must define a `name` attribute. You use this attribute as the tag name when instantiating the element. A Library tag can have any number of `<fx:Definition>` tags as children.

The following example defines the MyCircle and MySquare graphics with the Definition tags. It then instantiates several instances of these in the application file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- language/DefinitionExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <fx:Library>
          <fx:Definition name="MySquare">
               <s:Group>
                    <s:Rect width="100%" height="100%">
                         <s:stroke>
                              <s:SolidColorStroke color="red"/>
                         </s:stroke>
                    </s:Rect>
               </s:Group>
          </fx:Definition>
          <fx:Definition name="MyCircle">
               <s:Group>
                    <s:Ellipse width="100%" height="100%">
                         <s:stroke>
                              <s:SolidColorStroke color="blue"/>
                         </s:stroke>
                    </s:Ellipse>
               </s:Group>
          </fx:Definition>
     </fx:Library>
     <mx:Canvas>
          <fx:MySquare x="0" y="0" height="20" width="20"/>
          <fx:MySquare x="25" y="0" height="20" width="20"/>
          <fx:MyCircle x="50" y="0" height="20" width="20"/>
          <fx:MyCircle x="0" y="25" height="20" width="20"/>
          <fx:MySquare x="25" y="25" height="20" width="20"/>
          <fx:MySquare x="50" y="25" height="20" width="20"/>
          <fx:MyCircle x="0" y="50" height="20" width="20"/>
          <fx:MyCircle x="25" y="50" height="20" width="20"/>
          <fx:MySquare x="50" y="50" height="20" width="20"/>
     </mx:Canvas>
</s:Application>
```

Each Definition in the Library tag is compiled into a separate ActionScript class that is a subclass of the type represented by the first node in the definition. In the previous example, the new class is a subclass of mx.graphics.Group. This scope of this class is limited to the document. It should be treated as a private ActionScript class.

For more information about the Library tag, see "Library" on page 9.

Library

Use the `<fx:Library>` tag to define zero or more named graphic `<fx:Definition>` children. The definition itself in a library is not an instance of that graphic, but it lets you reference that definition any number of times in the document as an instance.

The Library tag must be the first child of the document's root tag. You can only have one Library tag per document. The following example defines a single graphic (MyTextGraphic) in the `<fx:Library>` tag, and then uses it three times in the application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- language/LibraryExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Library>
          <fx:Definition name="MyTextGraphic">
               <s:Group>
                    <s:RichText width="75">
                         <s:content>Hello World!</s:content>
                    </s:RichText>
                    <s:Rect width="100%" height="100%">
                         <s:stroke>
                              <s:SolidColorStroke color="red"/>
                         </s:stroke>
                    </s:Rect>
               </s:Group>
          </fx:Definition>
     </fx:Library>

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

     <fx:MyTextGraphic/>
     <fx:MyTextGraphic/>
     <fx:MyTextGraphic/>
</s:Application>
```

For more information, see "Definition" on page 8.

Private

The `<fx:Private>` tag provides meta information about the MXML or FXG document. The tag must be a child of the root document tag, and it must be the last tag in the file.

The compiler ignores all content of the `<fx:Private>` tag, although it must be valid XML. The XML can be empty, contain arbitrary tags, or contain a string of characters.

The following example adds information about the author and date to the MXML file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- language/PrivateExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComps="*">

    <mx:Canvas top="0" bottom="0" left="0" right="0">
          <s:Graphic>
               <s:RichText x="0" y="0">
                    <s:content>Hello World!</s:content>
               </s:RichText>
          </s:Graphic>
     </mx:Canvas>

    <fx:Private>
        <fx:Date>10/22/2008</fx:Date>
        <fx:Author>Nick Danger</fx:Author>
    </fx:Private>
</s:Application>
```

Reparent

The `<fx:Reparent>` language tag lets you specify an alternate parent for a given document node, in the context of a specific state.

For more information, see “States” on page 12.

Two-way data binding

Data binding is the process of tying the data in one object to another object. It provides a convenient way to pass data between the different layers of the application. Data binding requires a source property, a destination property, and a triggering event that indicates when to copy the data from the source to the destination. An object dispatches the triggering event when the source property changes.

In Flex 3, two-way data binding was possible by using a combination of curly braces, `<mx:Binding>` statements, and calls to the `mx.binding.utils.BindingUtils.bindProperty()` method.

Flex 4 introduces some shorthand ways to accomplish this. The two ways to specify a two-way data binding are:

1 Inline declaration using the `@{`*`bindable_property`*`}` syntax

2 With MXML, `<fx:Binding source="a.`*`property`*`" destination="b.`*`property`*`" twoWay="true/>`

In both cases, E4X statements can be used to reference bindable properties.

***Note:** In Flex 4, the namespace prefix of the data binding tag is “fx:”. In Flex 3, it was “mx:”.*

For the `<fx:Binding>` tag, because both the source and the destination must resolve to a bindable property or property chain, neither can be a one-way or two-way inline binding expression.

In Flex 3, to implement two-way binding, the code would like the following:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:TextInput id="t1" text="{t2.text}"/>
    <mx:TextInput id="t2" text="{t1.text}"/>
</mx:Application>
```

The following uses the shorthand two-way binding syntax in Flex 4:

```
<s:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <s:TextInput id="t1" text="@{t2.text}"/>
    <s:TextInput id="t2"/>
</s:Application>
```

In Flex 3, if you want to set two-way binding using the `<mx:Binding>` tag you need to set it twice, as the following example shows:

```
<mx:Binding source="a.property" destination="b.property"/>
<mx:Binding source="b.property" destination="a.property"/>
```

In Flex 4, this becomes:

```
<fx:Binding source="a.property" destination="b.property" twoWay="true"/>
```

Bindable expressions are supported in the many places, not all or which are appropriate for two-way bindable expressions. The following table shows where two-way binding is supported and not supported:

Expression	**Bindable**	**Two-way Bindable**
Property text	Yes	Yes
Style text	Yes	No
Effect text	Yes	No
`<Model>{`***`bindable_expression`***`}</Model>`	Yes	Yes
`<fx:XML>` top-level* or nodes or attributes	Yes	Yes
`<fx:XMLList>` top-level* or nodes or attributes	Yes	Yes
`<s:request>` for `<s:HttpService>`, `<s:RemoteObject>` and `<s:WebService>`	Yes	No
`<s:arguments>` for `<s:RemoteObject>`	Yes	No

For more information, see Data binding.

FXG

FXG is a declarative syntax for defining graphics in Flex applications. It can also be used as an interchange format with other Adobe tools such as Illustrator. FXG very closely follows the Flash Player 10 rendering model.

FXG defines the following:

- Graphics and text primitives
- Fills, strokes, gradients, and bitmaps
- Support for filters, masks, alphas, and blend modes

In FXG, all graphic elements implement the IGraphicElement interface.

Typically, you reference an FXG file as a standalone component in your application. Flex compiles the FXG tags into low-level Player instructions that are highly optimized.

The following example application uses an FXG file as a component:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- FXG/ArrowExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:comps="comps.*">

        <comps:Arrow/>
</s:Application>
```

The following example FXG file defines the component used in the previous example:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- fxg/comps/ArrowAbsolute.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Path data="
         M 20 0
         C 50 0 50 35 20 35
         L 15 35
         L 15 45
         L 0 32
         L 15 19
         L 15 29
         L 20 29
         C 44 29 44 6 20 6">
         <!-- Define the border color of the arrow. -->
         <stroke>
              <SolidColorStroke color="#888888"/>
         </stroke>
         <!-- Define the fill for the arrow. -->
         <fill>
              <LinearGradient rotation="90">
                   <GradientEntry color="#000000" alpha="0.8"/>
                   <GradientEntry color="#FFFFFF" alpha="0.8"/>
              </LinearGradient>
         </fill>
    </Path>
</Graphic>
```

For information on using FXG, see FXG and MXML Graphics.

States

Flex 4 lets you specify view states using a new inline MXML syntax, rather than the syntax used in Flex 3. In the new syntax, the AddChild, RemoveChild, SetProperty, SetStyle, and SetEventHandler classes have been deprecated and replaced with MXML keywords. This section describes the new states syntax for Flex 4.

For more information, see View states.

Defining states

To add view states, continue to use the `<s:states>` tag in your application, just as you did in Flex 3. In the body of the `<s:states>` tag, you add one or more `<s:State>` tags, one for each additional view state. However, there is no longer a base view state, as defined by the `currentState` property being set to the empty string (""). The default view state is now the first view state defined in the `<s:states>` tag.

In the following example, the base view state is the "default" state:

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

    <s:states>
        <s:State name="default"/>
        <s:State name="addCheckBox"/>
        <s:State name="addTextInput"/>
        <s:State name="addCheckBoxAndButton"/>
    </s:states>
</s:Application>
```

You no longer define view states by specifying the AddChild, RemoveChild, SetProperty, SetStyle, and SetEventHandler classes in the `<s:states>` tag. Instead, you use inline attribute in your MXML application.

In this release, you can no longer use data binding to set the value of the name property in the `<s:State>` tag.

Replacing the AddChild and RemoveChild classes

The AddChild and RemoveChild classes are replaced by the `includeIn` and `excludeFrom` MXML attributes. Use these attributes to specify the set of view state in which a component is included. The `includeIn` attribute takes a comma delimited list of view state names, all of which must have been previously declared within the application's `<s:states>` Array. The `excludeFrom` attribute takes a comma delimited list of view state names where the component in not included. The `excludeFrom` and `includeIn` attributes are mutually exclusive; it is an error to define both on a single MXML tag.

The following example uses view states to add components to the application based on the current state:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- states\NewStatesRemoveChild.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:states>
        <s:State name="default"/>
        <s:State name="addCheckBox"/>
        <s:State name="addTextInput"/>
        <s:State name="addCheckBoxandButton"/>
    </s:states>

    <!-- Included in the addCheckBox and addCheckBoxandButton view states. -->
    <s:CheckBox id="myCB" label="Checkbox"
        includeIn="addCheckBox, addCheckBoxandButton"/>
    <!-- Included in the addTextInput view state. -->
    <s:TextInput id="myTI"
        includeIn="addTextInput"/>
    <!-- Included in the addCheckBoxandButton view state. -->
    <s:TextInput id="myB"
        includeIn="addCheckBoxandButton"/>
    <s:Button label="Add CB"
        click="currentState='addCheckBox'"/>
    <s:Button label="Add TI"
        click="currentState='addTextInput'"/>
    <s:Button label="Add CB and Button"
        click="currentState='addCheckBoxandButton'"/>
    <s:Button label="Restore Default"
        click="currentState='default'"/>
</s:Application>
```

Notice that you no longer use the `AddChild.relativeTo` and `AddChild.position` properties to specify the location of a components when it is added to a view state. Instead, define the component where it should appear in the MXML application.

You can specify the `includeIn` and `excludeFrom` attributes on any MXML object within an MXML document, with the exception of the following tags:

- The root tag of an MXML document, such as the Application tag or the root tag in an MXML component.
- Tags that represent properties of their parent tag. For example, the `label` property of the `<s:Button>` tag.
- Descendants of the `<fx:XML>`, `<fx:XMLList>`, or `<fx:Model>` tags.
- Any language tags declared within the new Flex 4 language namespace such as the `<fx:Script>`, `<fx:Binding>`, `<fx:Metadata>`, and `<fx:Style>` tags.

Replacing the SetProperty, SetStyle, and SetEventHandler classes

You no longer use the SetProperty, SetStyle, and SetEventHandler classes in the `<s:states>` tag to define overrides for a view state. Instead, you use MXML attributes when defining a component in your application.

You define state-specific property values using the dot operator on any writable MXML attribute. The dot notation has the following format:

propertyName.stateName

For example, you can specify the value of the label property of a Button control for the default view state and for the State1 view state, as the following example shows:

```
<s:Button label="Default State" label.State1="New State" label.State2="Newer State"/>
```

The unqualified property, meaning the one that does not use the dot notation to specify a view state, defines the default value.

You use the same syntax for overriding a property, style, or event. The following example sets the enabled property of a Button control based on the current view state:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- states\NewStatesEnabled.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:states>
        <s:State name="default"/>
        <s:State name="addCheckBox"/>
        <s:State name="addTextInput"/>
        <s:State name="addCheckBoxandButton"/>
    </s:states>

    <!-- Included in the addCheckBox and addCheckBoxandButton view states. -->
    <s:CheckBox id="myCB" label="Checkbox"
        includeIn="addCheckBox, addCheckBoxandButton"/>
    <!-- Included in the addTextInput view state. -->
    <s:TextInput id="myTI"
        includeIn="addTextInput"/>
    <!-- Included in the addCheckBoxandButton view state. -->
    <s:TextInput id="myB"
        includeIn="addCheckBoxandButton"/>
    <s:Button label="Add CB"
        click="currentState='addCheckBox'"
        enabled.addCheckBox="false"/>
    <s:Button label="Add TI"
        click="currentState='addTextInput'"
        enabled.addTextInput="false"/>
    <s:Button label="Add CB and Button"
        click="currentState='addCheckBoxandButton'"
        enabled.addCheckBoxandButton="false"/>
    <s:Button label="Restore Default"
        click="currentState='default'"
        enabled.default="false"/>
</s:Application>
```

To clear the value of the property, set the property to the value `@Clear()`, as the following example shows:

```
<Button color="0xFF0000" color.State1="@Clear()"/>
```

For a style property, setting the value to `@Clear()` corresponds to calling the `clearStyle()` method on the property.

Layouts

Layouts in Flex 4 have changed to decouple the layout scheme from the rules of individual components. Layouts in Flex 4 can be defined declaratively and changed or removed at run time. In addition, layouts now support transformations.

To specify a layout declaratively, you specify the `<s:layout>` element as a child tag of the `<s:Application>` tag. Within that element, you specify one of the following mx.layout.* classes:

- BasicLayout
- HorizontalLayout
- VerticalLayout
- TileLayout

The following example defines the layout as BasicLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- layouts/BasicLayoutExample.mxml -->
<s:Application
xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:mx="library://ns.adobe.com/flex/mx"
xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
          <!-- BasicLayout uses absolute positioning. -->
          <s:BasicLayout/>
     </s:layout>

     <s:Button x="0" y="0" label="Click Me"/>
     <s:Button x="150" y="0" label="Click Me"/>
</s:Application>
```

A similar example specifies HorizontalLayout. In this case, you do not have to specify coordinates for the individual controls. The contents of the application are laid out as if they were in an HBox container.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- layouts/HorizontalLayoutExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
          <!-- HorizontalLayout positions controls horizontally. -->
          <s:HorizontalLayout/>
     </s:layout>

     <s:Button label="Click Me"/>
     <s:Button label="Click Me"/>
</s:Application>
```

BasicLayout instructs Flex applications to behave with absolute layout. It works in the same way as a MX Canvas container's layout, with the exception that it does not support advanced constraints. BasicLayout supports constraints (`left`, `right`, `top`, `bottom`, `horizontalCenter`, and `verticalCenter`), percent sizing (`percentWidth` and `percentHeight`). Because it is a Flex 4 layout, BasicLayout supports objects with arbitrary 2D transformations (rotation, scale, and skew).

HorizontalLayout and VerticalLayout function like the MX HBox and VBox layouts. They do not support constraints, but support percent sizing and distributing the container width/height between the objects with percent width/height (the same way MX HBox and VBox do it). They have properties like gap and align. As Flex4 layouts, they support objects with arbitrary 2D transformations (rotation, scale, and skew).

The following table describes the alignment properties supported by the HorizontalLayout and VerticalLayout classes:

Property	**Description**
`left/right` `top/bottom`	Each item is positioned at the specified edge.
`middle`	Each item is positioned in the center of the content height/width.
`justify`	Each item's height/width is set to the container's height/width.
`contentJustify`	All items are sized as the biggest item in the container, and the biggest item in the container is sized according to the regular layout rules. The layouts do one pass to determine what would be the height/width of each item. During this pass, the maximum value for the height/width is taken into account. In a second pass, the layouts set the height/width of each item to the calculated maximum value.

Flex 4 also includes the following new layout features:

1 User-settable layout properties were added to the IVisualItem interface and were implemented for GraphicElement (they already exist in UIComponent). You continue to access those properties directly on GraphicElement and UIComponent. For working with run-time objects, you can cast them to IVisualItem and work with those, without checking whether it is GraphicElement or UIComponent.

2 User-settable layout properties were added to the ILayoutItem interface. This includes the properties that are read directly from the layout classes such as the constraint properties and `percentWidth` and `percentHeight`. If you develop custom layouts, use these properties of the ILayoutItem interface rather than interfaces such as the IConstraintClient interface. Use the LayoutItemHelper utility class for working with values and targets of constraints.

3 Constraints `left`, `top`, `right`, `bottom`, `horizontalCenter`, `verticalCenter`, and `baseline` were added to the UIComponent and GraphicElement classes, as well as the IVisualItem interface. On UIComponent, the constraint properties implementations are proxies for style properties. On GraphicElement, they are standard properties.

   In Flex3 all the constraints were styles for UIComponent. In Flex 3, you had to call the `getStyle()` method to inspect the value. In Flex 4, you can access it directly. To maintain compatibility, UIComponent can still access constraints as styles.

4 Existing properties on the UIComponent class such as `width`, `height`, `explicitWidth`, and `measuredWidth` have been changed from post-scale to pre-transform. This makes UIComponent consistent with the behavior of the GraphicElement class.

   For more information, see the migration topic “Layouts” on page 105.

5 New properties `measuredX` and `measuredY` let components report bounds of drawings that are not rooted at the origin (0,0). In addition, the `skipMeasure()` method lets you optimize components that use these properties.

For more information, see About Spark layouts.

Sizing and positioning custom components

Components must implement the ILayout interface to be supported by the Flex 4 layouts. This interface is not designed to be used by application designers and developers.

If your component is based on the UIComponent or GraphicElement classes, you should not have to change much to get full layout support. The following general rules apply when creating custom components:

1. Implement drawing logic in the `updateDisplayList()` method. This is where you put drawing code and optionally perform sizing of the children (for containers that have children).
2. Compute the component's intrinsic/natural size and set the values of the `measuredXXX` properties in the `measure()` method. You can also call the `skipMeasure()` method to let your component opt out of the measurement process.
3. Call the proper combination of invalidation methods `invalidateSize()`, `invalidateDisplayList()`, and `invalidateParentSizeAndDisplayList()` for any new property setters you add to the new class.

   The following table describes the use of these invalidation methods.

Method	Description
`invalidateSize()`	Call this method to indicate that the intrinsic/natural size of the component is changing. This implies that the `measure()` method will be called (if needed, depending on the `skipMeasure()` method, for optimization purposes). Calling this method does not guarantee that `updateDisplayList()` will be called. It will be called only when the changes in the measured bounds result in changes of the actual size! To determine if the measured size will result in actual size changes, the system will run the `measure()` and `updateDisplayList()` methods on the parent, which in turn re-calculates the children actual size.
`invalidateDisplayList()`	Call this method to indicate that a property that affects the drawing code or a property that affects the layout of the children has changed. Calls `updateDisplayList()`.
`invalidateParentSizeAndDisplayList()`	Call this convenience method whenever a user-specified layout property is changing. This guarantees that the parent's `measure()` and `updateDisplayList()` methods will be called.

Effects

The effects architecture for Flex 4 makes effects more flexible and easier to use. One of the most important improvements is that effects now work in parallel. For example, a Move effect and a Resize effect can operate concurrently without interfering with each other.

You can also apply Spark effects to objects other than just Spark components. For example, you can apply Spark effects to:

- Any Spark or MX component
- Any graphical component in the spark.primitives package, such as Rect, Ellipse, and Path
- Any object that contains the styles or properties modified by the effect

The easing API is also simplified. This simplification makes it easier to define custom easing functions to use with Spark effects.

The Spark effects are divided into categories based on their implementation and target type:

Type	**Description**
Property effects	Animate the change of one or more properties of the target.
Transform effects	Animate the change of one or more transform-related properties of the target, such as the scale, rotation, and position. Modify the target in parallel with other transform effects with no interference among the effects.
Pixel-shader effects	Animate the change from one bitmap image to another, where the bitmap image represents the before and after states of the target.
Filter effects	Apply a filter to the target where the effect modifies the properties of the filter, not properties of the target.
3D effects	Modify the 3D transform properties of the target.

All of these effect classes are in the spark.effects.* package.

In Flex 4, effects must be declared in a `<fx:Declarations>` tag.

You do no use triggers, such as `showEffect` and `hideEffect`, with the new Spark effects. Instead, you call the `play()` method of the effect class in response to an event, such as the `show` or `hide` events.

Flex 4 also includes new easing classes to use with the new effects. These easing classes ship in the spark.effects.easing package.

For more information, see Spark effects.

Advanced CSS

CSS in Flex has been enhanced to provide more advanced features. Part of the reason for this is that it plays a more prominent role in applying skins to components.

When discussing CSS in Flex, the subject of a selector is the right-most simple type selector in a potentially-complex selector expression. In the following example, the Button is the subject of the selectors:

```
VBox Panel Button#button12 {
    color: #DDDDDD;
}
VBox.special Button {
    color: #CCCCCC;
}
Button.special {
    color: #BBBBBB;
}
```

For more information about CSS in Flex, see Styles and Themes.

Namespaces in CSS

Some Spark and MX components share the same local name. For example, there is a Spark Button component (in the spark.components.* package) and an MX Button component (in the mx.controls.* package). To distinguish between different components that share the same name, you specify namespaces in your CSS that apply to types. For example, you can specify that a particular selector apply to all components in the Spark namespace only.

If you do not use type selectors in your style sheets, then you are not required to specify namespaces.

To specify a namespace in CSS, you declare the namespace with the `@namespace` directive, followed by the namespace's library in quotation marks. The following example defines the Spark namespace and uses the "s" as an identifier:

```
@namespace s "library://ns.adobe.com/flex/spark";
```

The following are valid CSS namespaces:

- `library://ns.adobe.com/flex/spark`
- `library://ns.adobe.com/flex/mx`

After you specify a namespace's identifier, you can use it in CSS selectors. The following example uses the Spark namespace for the Button components and the MX namespace for the Box containers:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/NamespaceIdentifierExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComps="*">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";

        s|Button {
            fontSize:16;
        }

        mx|VBox {
            color:red;
        }
     </fx:Style>

    <mx:VBox>
        <!-- This Spark button is red, and has a fontSize of 16. -->
        <s:Button label="Click Me, Too"/>
    </mx:VBox>
</s:Application>
```

You can also specify that a particular selector apply to mixed nested namespaces. This is common if you are using descendant selectors, as described in "Descendant selector" on page 21.

You can exclude an identifier, in which case the declared namespace becomes the default namespace. The following example uses the default namespace:

```
<Style>
    @namespace "library://ns.adobe.com/flex/spark";
    Button { color: #990000; }
</Style>
```

For custom components that are in the top level package, you can use an "*" for the namespace.

New CSS selectors

The CSS selector syntax has been expanded to include the following additional types of selectors:

- ID
- Descendant
- Pseudo

For selectors that use class names, you can use parent classes rather than the subclass. For example, if you define a class selector for Group, the style applies to all components that are subclasses of Group, such as VGroup and HGroup. This works for all classes below UIComponent in the Flex class hierarchy. UIComponent is considered a “stop class”.

The following sections describe each of these selectors.

For additional information about these selectors, see CSS3 Selectors W3C specification.

ID selector

Flex now supports using an ID selector. Flex applies styles to a component whose id property matches the ID selector in the CSS. To define an ID selector, specify the id property of the component with a pound sign (#) followed by the ID string. The ID selector can be qualified by the type or by itself.

The following example shows two ways to use the ID selector:

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

<s:HorizontalLayout/>
     </s:layout>

     <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        s|Button {fontSize:16;}

        #myButton2 {color:red;}

        s|Button#myButton3 {color:blue;}
     </fx:Style>

     <s:Button id="myButton1" label="Click Me"/>
     <s:Button id="myButton2" label="Click Me, Too"/>
     <s:Button id="myButton3" label="Click Me, Three"/>
</s:Application>
```

Descendant selector

Descendant selectors are applied to components in a document, depending on their relationship to other components in the document. A descendant selector lets you apply styles to a component based on whether they descend (are children, grandchildren, or great grandchildren) from particular types of components.

When a component matches multiple descendant selectors, it adopts the style of the most closely-related ancestor. If there are nested descendant selectors, the component uses the styles defined by the selector that most closely matches its line of ancestors. For example, a Button within a VGroup within a VGroup matches a descendant selector for “VGroup VGroup Button” rather than a descendant selector for “VGroup Button”.

The following example shows four selectors. The first class selector applies to all Button instances. The first descendant selector applies to the second button, because that Button’s parent is a VGroup. The second descendant selector applies to the third Button, because that Button is a child of an HGroup, which is a child of a VGroup. The last descendant select applies to the last Button, because that Button is a child of a VGroup, which is a child of a VGroup.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/CSSDescendantSelectorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComps="*">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        s|Button {fontSize:16;}

        s|VGroup s|Button {color:red;}

        s|VGroup s|HGroup s|Button {color: blue;}

        s|VGroup s|VGroup s|Button {color: green;}
     </fx:Style>

     <!-- This button has a fontSize of 16. -->
     <s:Button label="Click Me"/>

     <s:VGroup>
        <!-- This button is red, and has a fontSize of 16. -->
        <s:Button label="Click Me, Too"/>
      </s:VGroup>

     <s:VGroup>
        <s:HGroup>
            <!-- This button is blue, and has a fontSize of 16. -->
            <s:Button label="Click Me, Also"/>
        </s:HGroup>
     </s:VGroup>

     <s:VGroup>
        <s:VGroup>
            <!-- This button is green, and has a fontSize of 16. -->
            <s:Button label="Click Me, Click Me!"/>
        </s:VGroup>
     </s:VGroup>
</s:Application>
```

Styles are applied only to components that appear in the display list. Descendant selectors are only applied if the ancestors also appear in the display list.

Descendant selectors work with all classes that implement the IStyleClient interface.

A descendant selector applies to a component as long as any parent class of that component matches the descendant selector. The following example shows that two Button controls inherit the style of the Group descendant selector, because their parents (VGroup and HGroup) are both subclasses of Group.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- CSSDescendantSelectorExample2.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComps="*">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";
        s|Group s|Button {color:red;}
     </fx:Style>

    <s:VGroup>
        <!-- This button is red, because VGroup is a subclass of Group. -->
        <s:Button label="Click Me"/>
     </s:VGroup>
     <s:HGroup>
        <!-- This button is also red, because HGroup is also a subclass of Group. -->
        <s:Button label="Click Me, Too"/>
     </s:HGroup>
</s:Application>
```

Pseudo selector

A pseudo selector matches components based on its state.

The following example changes the Button component’s color, depending on whether it is in the up, down, or over state:

```
<?xml version="1.0"?>
<!-- PseudoSelectorExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:custom="*">
    <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";

        s|Button:up {
            baseColor: black;
            color: #FFFFFF;
        }

        s|Button:over {
            baseColor: gray;
            fontWeight: "bold";
            color: #FFFFFF;
        }

        s|Button:down {
            baseColor: blue;
            fontWeight: "bold";
            color: #FFFF66;
        }
    </fx:Style>

    <s:Button label="Click Me" x="10" y="35"/>
</s:Application>
```

To get a list of available states for a component, view the component's entry in the ASDoc Language Reference.

You can apply pseudo selectors to custom states in addition to the default states of a component.

CSS-related APIs

Several APIs have been updated and some new APIs have been added to support the new CSS selector syntax.

The following table describes the new classes added to the Flex framework to support advanced CSS:

Class	**Description**
mx.styles.IAdvancedStyleClient	Matches components based on advanced style selector criteria such as identify, state, or descendant position in the display list. Components that use these types of selectors must implement this interface.
mx.styles.CSSSelector	Represents a chain of selectors each with conditions and ancestors.
mx.styles.CSSSelectorKind	Defines constants used by the `CSSSelector.kind` property.
mx.styles.CSSCondition	Records each type of selector condition.
mx.styles.IStyleManager3	Adds methods and properties to the StyleManager to keep track of multiple CSSStyleDeclarations for each subject, as well as other advanced CSS properties.
mx.styles.CSSConditionKind	Defines constants used by the `CSSCondition.kind` property.

The following table describes changes to the CSSStyleDeclaration class that were added to support advanced CSS:

CSSStyleDeclaration Class Member	**Description**
Constructor	The constructor now takes the `subject` and `selector` arguments. The new signature for the CSSStyleDeclaration constructor is as follows: `CSStylesDeclaration(subject:String, selector:CSSSelector)` For backwards compatibility, if the arguments are `null`, the selector's subject is interpreted as a simple type selector name. If the subject begins with a period, the subject is interpreted as a global class selector.
`isMatch()`	Returns `true` if this style declaration applies to the given component based on a match of the selector chain.

Skinning

The skinning workflow has been greatly simplified for Flex 4. In the Flex 4 skinning model, the skin controls all visual elements of a component, including layout. The new architecture gives developers greater control over what their components look like a structured and tool-friendly way. Previously, MX components that used the Halo theme for their skins defined their look and feel primarily through style properties.

Spark skins can contain multiple elements, such as graphic elements, text, images, and transitions. Skins support states, so that when the state of a component changes, the skin changes as well. Skin states integrate well with transitions so that you can apply effects to one or more parts of the skins without adding much code.

You typically write Spark skin classes in MXML. You do this with MXML graphics tags (or FXG components) to draw the graphic elements, and specify child components (or subcomponents) using MXML or ActionScript.

The base class for Flex 4 skins is the Skin class. The default Spark skins are based on the SparkSkin class, which subclasses the Skin class.

For more information, see Spark Skinning.

HTML wrappers

Flex 4 generates an HTML wrapper based on new templates. The results are different output than previous editions of Flex and Flash Builder.

The templates are in the following locations:

- {*flex_sdk_root*}/templates
- {*flex_builder_install*}/

To generate HTML wrapper output, you can use one of the following methods:

- Flash Builder
- `mxmlc-wrapper` ant task

In previous editions of Flex, the wrapper included a JavaScript file called AC_OETags.js. The method defined in this file output code that embedded the Flex SWF file in HTML. The wrapper was generated by one of six different templates, depending on the combination of features you wanted included (such as express install or deep linking support). These files embedded other files and defined properties that were used to add deep linking and express install support into your HTML wrapper output.

To embed the SWF file in the HTML wrapper, Flex 4 uses a JavaScript file called swfobject.js. This file includes the SWFObject 2. It is designed to be more standardized and easier to work with than the previous template scripts. In addition, SWFObject 2 is maintained in an open source repository. For more information, see the following:

- SWFObject documentation
- SWFObject JavaScript API documentation

The new template builds the parameters to pass to the `swfobject.embedSWF()` method. This method is defined in the swfobject.js. file.

When deploying Flex 4 applications, you must deploy, at a minimum, the following files:

- index.template.html
- swfobject.js

In addition to these files, you must also deploy other files if you enable deep linking or express install.

The new HTML template includes the following new tokens:

- `${useBrowserHistory}`
- `${expressInstallSwf}`

For deep linking, you must deploy the following files in addition to the files listed above:

- history/history.css
- history/history.js
- history/historyFrame.html

For express install, you must deploy the following file in addition to the files listed above:

- playerInstall.swf

Typically, you copy and paste the logic from the output template into your HTML wrapper, so you will not usually deploy the entire HTML wrapper (index.template.html) as generated.

Embedding `flashVars` variables in your SWF file uses new syntax. To embed `flashVars` variables in your SWF file, you add the logic directly to the properties that are passed to the `swfobject.embedSWF()` method. This method is defined in the HTML template.

To add `flashVars` variables to your template, you attach dynamic properties to the `flashvars` object in the HTML template. That object is passed as one of the parameters to the `swfobject.embedSWF()` method.

The following example adds `firstName` and `lastName` as dynamic properties to `flashvars` object. It then passes this object to the `swfobject.embedSWF()` method:

```
<script type="text/javascript" src="swfobject.js"></script>
<script type="text/javascript">
     <!-- For version detection, set to min. required Flash Player version, or 0 (or 0.0.0),
for no version detection. -->
     var swfVersionStr = "${version_major}.${version_minor}.${version_revision}";
    <!-- To use express install, set to playerProductInstall.swf, otherwise the empty string.
-->
     var xiSwfUrlStr = "";

     var flashvars = {};
     flashvars.firstName = "Nick";
     flashvars.lastName = "Danger";

     var params = {};
     params.quality = "high";
     params.bgcolor = "${bgcolor}";
     params.allowscriptaccess = "sameDomain";
     var attributes = {};
     attributes.id = "${application}";
     attributes.name = "${application}";
     attributes.align = "middle";
     swfobject.embedSWF(
          "${swf}.swf", "flashContent",
          "${width}", "${height}",
          swfVersionStr, xiSwfUrlStr,
          flashvars, params, attributes
     );

     <!-- JavaScript enabled so display the flashContent div in case it is not replaced with
a swf object. -->
     swfobject.createCSS("#flashContent", "display:block");
</script>
```

Deferred instantiation

In Flex 4, the contents of a component are treated like any other property and are initialized in the component's constructor. Since there is no defined order of property initialization, there is no control over the timing of the content initialization in relation to other properties. During normal, non-deferred instantiation, `createDeferredContent()` is called from `createChildren()`, or by calling any of the item APIs such as `getItemAt()` and `addItem()`. The `contentCreationComplete` event is sent when the content has been created.

Groups do not support deferred creation. When you create a Group, VGroup, or HGroup, all children of the those classes are created. You cannot defer their creation.

Flex 4 applications do not support queued instantiation.

SkinnableContainer is a new base class for all skinnable components that have content. For example, Panel. This is mentioned in the section on the new component architecture.

The SkinnableContainer class has a method called `createDeferredContent()`. When the value of the `creationPolicy` property is `auto` or `all`, this function is called automatically by the Flex framework. When the value of the `creationPolicy` property is `none`, this method must be called to initialize the content property.

The following example sets the value of the `creationPolicy` property of the Container to `none`. The result is that the contents of the container, in this case a CheckBox, an RadioButton, and a Label, are not created when the application starts up. You can explicitly create the deferred content by clicking the Button control.

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

     <mx:Panel title="Create Deferred Content Example"
        height="75%" width="75%" layout="horizontal"
        paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10">

         <s:Button label="Create Content" click="cont.createDeferredContent()"/>

         <s:Panel id="cont" creationPolicy="none">
             <s:CheckBox label="Check Box" y="20"/>
             <s:RadioButton label="Radio Button" y="40"/>
             <mx:Label text="Label" y="60"/>
         </s:Panel>
     </mx:Panel>
</s:Application>
```

The IDeferredContentOwner interface is also new for Flex 4. It is implemented by any classes that support deferred instantiation. The only Flex 4 class that implements this interface is SkinnableContainer.

Two ways to use the `SkinnableContainer.createDeferredContent()` method are:

1 If you extend a container and want to walk the content children immediately after their creation.

2 If you create a custom component that has content and the component does not extend SkinnableContainer, you can manually control the content creation. You can implement the IDeferredContentOwner interface for the component. You can use the `createDeferredContent()` method to initialize the component's content.

The following example extends the Container class and overrides the `createDeferredContent()` method. After calling the `super.createDeferredContent()` method, it displays an Alert window.

```
package {

import mx.components.FxContainer;
import mx.components.FxButton;
import mx.events.FlexEvent;
import mx.events.CloseEvent;
import mx.controls.Alert;

public class MyCustomIC extends FxContainer {
    public function MyCustomIC() {
        super();
    }

    override public function createDeferredContent():void {
        super.createDeferredContent();
       Alert.show("Hello from createDeferredContent", "Message", 1, this, alertClickHandler);
    }

    private function alertClickHandler(event:CloseEvent):void {
    }
}
}
```

The following example is a little more complicated than the previous example. It is a custom component, myComp/MyFxCont.as. It calls a method of the loading Application from the `createDeferredContent()` method. This ensures that the content children are created before the method is called.

```
package myComp
{
    import mx.containers.Canvas;
    import mx.core.Application;
    import mx.core.FlexGlobals;

    public class MyFxCont extends Canvas
    {
        public function MyFxCont()
        {
            super();
        }

        override public function createDeferredContent():void
        {
            super.createDeferredContent();

            FlexGlobals.topLevelApplication.updatePrice(void);
        }
    }
}
```

The following example is the application that loads the MyFxCont component. This application implements the `updatePrice()` method. After the component and its children are created, the component calls the `updatePrice()` method.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- deferred_instantiation/MainContentChildrenApp.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComp="myComp.*"
    backgroundColor="0xFFFFFF">

     <fx:Script>
          <![CDATA[
               import mx.utils.ObjectUtil;

               public var totalPrice:Number = 23000;
               public var hasCd:Boolean = false;
               public var hasLeather:Boolean = false;
               public var hasSunroof:Boolean = false;

               public function updatePrice(event:MouseEvent):void
               {
                    if(cd.selected && !hasCd)
                    {
                         totalPrice += 500;
                         hasCd = true;
                    }

                    if(!cd.selected && hasCd)
                    {
                         totalPrice -= 500;
                         hasCd = false;
                    }

                    if(leather.selected && !hasLeather)
                    {
                         totalPrice += 1075;
                         hasLeather = true;
                    }

                    if(!leather.selected && hasLeather)
                    {
                         totalPrice -= 1075;
                         hasLeather = false;
                    }

                    if(sunroof.selected && !hasSunroof)
                    {
                         totalPrice += 1599;
                         hasSunroof = true;
                    }

                    if(!sunroof.selected && hasSunroof)
                    {
                         totalPrice -= 1599;
                         hasSunroof = false;
                    }

                    total.text = usdFormatter.format(totalPrice);
```

```
                }
            ]]>
        </fx:Script>

        <fx:Declarations>
            <mx:CurrencyFormatter id="usdFormatter" precision="2" currencySymbol="$"
                decimalSeparatorFrom="." decimalSeparatorTo="." useNegativeSign="true"
                useThousandsSeparator="true" alignSymbol="left"/>
    </fx:Declarations>

        <s:Panel width="75%" height="75%" title="Example 1">
            <s:layout>
                <s:VerticalLayout/>
            </s:layout>
            <s:VGroup>
                <s:Label text="Honda Accord Ex Base Price: $23,000" fontSize="16"/>
                <s:Label text="Select options for your car:"/>
                <s:CheckBox id="cd" label="CD Player $500"
                    click="updatePrice(event);"/>
                <s:CheckBox id="leather" label="Leather Seats ($1075)"
                    selected="true" click="updatePrice(event);"/>
                <s:CheckBox id="sunroof" label="Sun Roof $1599"
                    click="updatePrice(event);"/>
            </s:VGroup>

            <mx:Spacer height="10"/>
            <s:Label text="Total Price:" fontSize="16"/>

            <myComp:MyFxCont id="myFxC">
                <s:Label id="total" text="$23,000.00" fontSize="14" />
            </myComp:MyFxCont>

        </s:Panel>
</s:Application>
```

For more information about deferred instantiation in Flex, see Improving Startup Performance

DataGroup

The DataGroup class defines a container for displaying data objects using an item renderer. All other Flex containers, such as Group, require that their children implement the IVisualItem interface. The UIComponent class implements the IVisualItem interface so that you can use any Flex component as a child of a container.

The DataGroup class can take any object as a child, even objects that do not implement the IVisualItem interface. The DataGroup class then converts its children to visual children for display. For example, you can use the DisplayObject class to display a String, Object, Boolean, or other data type, as the following example shows:

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

    <s:DataGroup itemRenderer="MyItemRenderer">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayCollection>
            <fx:Object firstName="Dave" lastName="Davis"/>
            <fx:Object firstName="Jim" lastName="Jackson"/>
            <fx:Object firstName="Mary" lastName="Jones"/>
            <fx:Object firstName="Ellen" lastName="Smith"/>
            <fx:Object firstName="Amelia" lastName="Sanson"/>
        </mx:ArrayCollection>
    </s:DataGroup>
</s:Application>
```

In this example, you use the DataGroup class to display a group of Objects. The DataGroup class uses the VerticalLayout class in this example to arrange the Objects in a column.

The DataGroup class itself does not define any visual elements. For example, if you require a border around a DataGroup class, you must add it yourself. The same is true for scrollbars. Scrolling is handled by the Scroller component.

To control how the children in the DataGroup are displayed, you typically specify an item renderer, or a function that returns an item renderer, to the DataGroup class. However, if the DataGroup class contains children that implement IVisualItem, or are of type mx.graphics.graphicsClasses.GraphicElement or of type flash.display.DisplayObject, you do need an item renderer.

Notice in the previous example that the DataGroup specifies an item renderer to display each child. The item renderer for this example, MyItemRenderer.mxml, is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- groups\MyItemRenderer.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
    <s:states>
        <s:State name="normal"/>
        <s:State name="hovered"/>
    </s:states>

    <s:Label text="{data.lastName}, {data.firstName}"/>
</s:ItemRenderer>
```

The item renderer in this example displays the last name, a comma, and the first name for each child Object.

Controlling the display of children using item renderers

The DataGroup class creates visual items for its children using the follow rules:

1 If the `itemRendererFunction` property is defined, call it to obtain the renderer factory and instantiate it. If it returns `null`, go to rule 2.

2 If the `itemRenderer` property is defined, use the `itemRenderer` to display the item.

3 If the item is of type mx.graphics.graphicsClasses.GraphicElement, create the display object for it and use it directly.

4 If the item is of type flash.display.DisplayObject, use it directly.

The `itemRendererFunction` property takes a function with the following signature:

```
function itemRendererFunction(item:Object):IFactory
```

where `item` is an item from the data provider, and IFactory is the name of an item renderer or `null`.

For more information, see The Spark DataGroup and Spark SkinnableDataContainer containers .

ASDoc

The ASDoc command-line tool parses one or more ActionScript class definitions and MXML files to generate API language reference documentation for all public and protected methods and properties, and for certain metadata tags. This release of Flex adds new features to the ASDoc command-line tool, including support for ASDoc comments in MXML files.

For more information, see ASDoc.

MXML file support

The previous release of the ASDoc command-line tool processed MXML files as input, but did not support ASDoc comments in the MXML file. In this release, you can now use the following syntax to specify an ASDoc comment in an MXML file:

```
<!--- asdoc comment -->
```

The comment must contain three dashes following the opening `<!` characters, and end with two dashes before the closing `>` character, as the following example shows:

```
<?xml version="1.0"?>
<!-- asdoc\MyVBox.mxml -->
<!---
    The class level comment for the component.
    This tag supports all ASDoc tags,
    and does not require a CDATA block.
-->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <!---
        Comment for button
     -->
    <s:Button id="myButton" label="This button has a comment"/>
</mx:VBox>
```

In this example, the first comment is a standard XML comment that is ignored by ASDoc. The second comment precedes the root tag of the component and uses the three dashes to identify it as an ASDoc comment. An ASDoc comment on the root tag is equivalent to the ASDoc comment before an ActionScript class definition. Therefore, the comment appears at the top of the output ASDoc HTML file.

All MXML elements in the file correspond to public properties of the component. The comment before the Button control defines the ASDoc comment for the public property named myButton of type mx.controls.Button.

You can use any ASDoc tags in these comments, including the `@see`, `@copy`, `@param`, `@return`, and other ASDoc comments.

Specify the input MXML file to the compiler in the same way that you specify an ActionScript file. For example, you can use the -doc-sources option of the compiler to process this file:

```
>asdoc -doc-sources C:\myApp\myMXMLFiles\MyVBox.mxml -output framework-asdoc
```

The ASDoc command-line tool only processes elements of an MXML file that contain an id attribute. If the MXML element has an id attribute but no comment, the elements appears in the ASDoc output with a blank comment. An MXML element with no `id` attribute is ignored, even if it is preceded by an ASDoc comment, as the following example shows:

```
<?xml version="1.0"?>
<!-- asdoc\MyVBoxID.mxml -->
<!---
    The class level comment for the component.
    This tag supports all ASDoc tags,
    and does not require a CDATA block.

    @see mx.container.VBox
-->
<mx:VBox xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <!---
        Comment for first button appears in the output.
     -->
    <s:Button id="myButton" label="This button has a comment"/>

    <s:Button id="myButton2"
        label="Has id but no comment so appears in output"/>

    <!---
        Comment for button with no id is ignored by ASDoc.
     -->
    <s:Button label="This button has no id"/>
</mx:VBox>
```

You can insert ASDoc comments for metadata tags in the `<fx:Script>` and `<fx:Metadata>` blocks in an MXML file. Comments before `<fx:Definition>`, `<fx:Library>`, and `<fx:Private>` tags are ignored. Also comments inside a private block are ignored.

New option to the ASDoc command-line tool

By default, the ASDoc command-line tool halts processing and outputs a failure message when an ASDoc comment in an input file contains invalid HTML code. The tool writes error information to the validation_errors.log file.

This release of the ASDoc command-line tool adds the `-lenient` option that specifies to complete the compilation even when an HTML error occurs. The ASDoc comment that included the error is omitted from the output and the tool writes error information to the validation_errors.log file, but the remainder of the file or files is processed normally.

Conditional compilation

Setting the values of constants on the command line overrides the values of those parameters if they are set in the configuration files.

For example, if you have the following definitions in your flex-config.xml file:

```
<define append="true">
    <name>config::LOCAL</name>
    <value>false</value>
</define>
<define append="true">
    <name>config::DEBUG</name>
    <value>false</value>
</define>
```

You can now override the value of those constants on the command line. Previously, the command line definitions did not override the value of the constants in the configuration files

To override a value of a constant on the command line, use the following syntax:

```
-define+=namespace::property_name,property_value
```

In this example, you could change the value of the `DEBUG` constant to `true` and `LOCAL` to `false` by using the following:

```
mxmlc -define+=config::DEBUG,true -define+=config::LOCAL,false MyFile.mxml
```

In addition, previously, if a single constant was defined on the command line, all constants in the configuration file were ignored. Now, constants in configuration files are not ignored if constants are defined on the command line, as long as you use "+=" to specify the constants on the command line. For example:

```
mxmlc -define+=test::RECORD,true MyFile.mxml
```

In this case, the value of the `test::RECORD` constant is available, as well as any constants defined in the application's configuration files.

Chapter 2: New Flash Builder Features

The following new features for Flash Builder are available with the Adobe MAX Gumbo Preview:

Flex Package Explorer

This release introduces the Flex Package Explorer, which replaces the Flex Navigator.

All projects in a workspace are displayed in the Package Explorer. The Package Explorer provides a tree view of projects from both a physical view and logical (flat) view. Using the Package Explorer, you manage projects by adding and deleting resources (folders and files), importing and linking to external resources, and moving resources to other projects in the workspace.

Highlights of the Flex Package Explorer include:

- You can display ActionScript packages in either a hierarchical or flat presentation.

  Use the Package Explorer's menu to specify the package presentation.

- Project libraries are represented in two top-level nodes, one node for the Flex SDK and the other for referenced libraries.

  You can expand a library's contents and open editors to view attachments.

- Error and warning badges on Package Explorer nodes notify you of problems within a package.

- You can limit which projects and resources are visible.

  You can create a working set (a collection of resources), create display filters, and sort resources by name and type. These options are available from the Package Explorer menus.
- You can expand ActionScript, MXML, and CSS files and see a tree view of their contents.

Project Configuration

In this release, Flash Builder provides the following enhanced functionality for configuring Flash Builder projects:

- Changing the server options of an existing web application or desktop application project
- Changing a web application project to a desktop application project (runs in Adobe AIR)
- Specifying server compile options

Changing server options of existing projects

At times you may find that the original server configuration for a project does not meet your current needs. You can reconfigure the server configuration for a web application or desktop application from the Project Properties window.

In the Project Properties window select the Flex Server option to add or change the server options for the project:

- Select None to remove the server configuration from a project.

  Removing the server configuration from a project removes any added SWCs on the library path for that server type.
- Select a server type to change or add the server configuration of a project

  All the server options for the selected server configuration are available.

  Changing the server type of a project can result in errors in existing code that relies on the original server type. You need to investigate and correct any resulting errors in your code.

Changing Flex projects to Adobe AIR projects

You can change the application type of a Flex project from Web (runs in Flash Player) to Desktop (runs in Adobe AIR). The following changes are made during the conversion:

- An AIR descriptor file is created for each application in the project.
- The launch configurations for the project are updated to properly launch in the AIR runtime.
- Settings for HTML wrapper are removed.
- Custom Flash Player settings are removed.
- The library path is modified to include airglobal.swc instead of playerglobal.swc.

During conversion, you can specify whether to change the base Application tags to WindowedApplication tags for each application in the project. If you choose to convert these tags, this is the only change to application code that occurs during conversion. You should inspect the attributes to the base tags after the conversion to make sure the application runs as intended in Adobe AIR.

To change a web application project to a desktop application

***Note:** This procedure cannot be undone.*

1 Select the project that you want to convert.

  The project should be a Flex project with the Web application type (runs in Flash Player)

2 From the context menu for the project select:

  Add/Change Project Type > Convert to Desktop/Adobe AIR project.

3 In the Convert to Desktop/Adobe AIR Project dialog, specify whether to rewrite code:

  - Convert Application Tags to WindowedApplication Tags

    For existing applications in the project, all Application tags are rewritten to WindowedApplication tags. No other change to your code occurs. Inspect attributes to the base tags to make sure the application runs as intended in Adobe AIR.

    New applications you create in the project are desktop applications and can run in Adobe AIR.

  - Do Not Rewrite Any Code

    No changes are made to your code. You must edit any applications in the project before they can run in Adobe AIR.

    New applications you create in the project are desktop applications and can run in Adobe AIR.

Specifying an SDK for a project

When creating a new Flex project, you can specify which Flex SDK to use. However, you can later modify the SDK settings by selecting Project > Properties > Flex Compiler > Use a specific SDK.

If you want to compile your project against a version of the Flex SDK that is not available in your Flash Builder installation, you can download the SDK and add it to your installation. For example, if you want to match the exact SDK that is installed on your server, extract the SDK from the server and then add the SDK to Flash Builder using Project > Properties > Flex Compiler > Configure Flex SDKs.

Importing projects that use remote server compilation

Importing projects that use server compile is not supported. You can import a project that specifies server compilation, however the project is imported with errors in the Problems View. The error provides a link with information on how to convert a server compile project to a tool compile project.

Flash Builder command line build

Flash Builder provides the Ant task <fb.exportReleaseBuild>. Use this task to implement command line builds that synchronize a developer's individual build settings with the nightly build. Alternatively, you can use the <mxmlc> task in custom scripts for nightly builds.

<fb.exportReleaseBuild> task

Using the <fb.exportReleaseBuild> task ensures that the nightly build's settings exactly match the settings used by developers during their daily work.

For example, if a developer changes the library path of a Flex project, the new library path is written to the Flash Builder project. When the nightly build machine runs <fb.exportReleaseBuild>, that task loads the Flash Builder project and all of its settings.

Another advantage of using <fb.exportReleaseBuild> is that it automatically takes care of additional “housekeeping” tasks normally included in a Flash Builder build, such as:

- Automatically compile associated library projects
- Copy assets such as JPEGs, and so on, into the output directory
- Copy the HTML template, including macro substitution based on the compilation results (such as width and height)

***Note:** The <fb.exportReleaseBuild> task requires you to install Flash Builder on the nightly build machine.*

<mxmlc> task

If you write a custom script that uses the <mxmlc> task (for example, an Ant script), then you do not need to install Flash Builder on the build machine. However, the build machine is required to have the Flex SDK available. Thus, the build machine can be on a Linux platform.

However, the disadvantage of this approach is that you have two sets of build settings to synchronize: One in Flash Builder, used by developers during their daily work, and another on your nightly build machine.

<fb.exportReleaseBuild> usage

1 Install Flash Builder on a build machine.

2 Write `build.xml` with fb.exportReleaseBuild as a target. For example:

```
<?xml version="1.0"?>
<project default="main">
    <target name="main">
        <fb.exportReleaseBuild project="MyProject" />
    </target>
</project>
```

`build.xml` specifies to run a command line build of your Flex project, using the settings saved in your project files. See “<fb.exportReleaseBuild> usage” on page 39 for details on available parameters.

3 Create a nightly build script that tells Eclipse to look for a build file and execute its target.

The following examples specify `build.xml` as a build file, which executes MyTarget.

If your nightly build script is on a Macintosh platform, you could run the following script:

```
WORKSPACE="$HOME/Documents/Adobe Flash Builder"

# works with either FlashBuilder.app or Eclipse.app
"/Applications/Adobe Flash Builder/FlashBuilder.app/Contents/MacOS/FlashBuilder" \
    --launcher.suppressErrors \
    -noSplash \
    -application org.eclipse.ant.core.antRunner \
    -data "$WORKSPACE" \
    -file "$(pwd)/build.xml" MyTarget
```

If your nightly build is on a Windows platform, you could run the following batch file:

```
  set WORKSPACE=%HOMEPATH%\Adobe Flash Builder

  REM works with either FlashBuilderC.exe or eclipsec.exe
  "C:\Program Files\Adobe\Adobe Flash Builder\FlashBuilderC.exe" ^
      --launcher.suppressErrors ^
      -noSplash ^
      -application org.eclipse.ant.core.antRunner ^
      -data "%WORKSPACE%" ^
      -file "%cd%\build.xml" MyTarget
```

fb.running Ant property

The fb.running Ant property has a value of true when Flash Builder is running. You can use this property when running scripts inside Flash Builder. For example:

```
<target name="myFlashBuilderTasks" if="fb.running">
    <fb.exportReleaseBuild ... />
</target>
```

Eclipse Ant Tasks

Eclipse provides several Ant tasks you can incorporate as targets in your build script. For example:

- eclipse.incrementalBuild
- elicpse.refreshLocal
- eclipse.convertpath

Consult the Eclipse documentation for more information on these scripts.

Parameters for fb.exportReleaseBuild task

Attribute	**Description**	**Required?**	**Default Value**
project	The project to build. Specify the name of a project in your Flash Builder workspace, without a path. For example, "MyFlexProject."	Yes	n/a
application	The name of the application to compile. You can specify just the application name with no path or extension (for example: app1). To avoid ambiguity in naming, you can specify full path, relative to the project root (for example: src/app1.mxml).To compile all applications, specify '*', or omit this attribute. When running against an AIR project, you can only specify a single application. The '*' value is not allowed.	No	The default application of the project.
publishsource	Whether to publish the source of the application, allowing the user to view source files using the context menu View Source.	No	false
locale	Specifies the locale, for example, en-US. This value is passed to the compiler using the compiler's -locale flag. If specified, this overrides any locale that has been specified in Flash Builder's Additional Compiler Arguments field.	No	n/a
destdir	The output folder. The folder can be a relative path or an absolute path. If you specify a relative path, it is relative to the root of the project. If compiling an AIR project, the folder is a temporary directory that is deleted after the .air file has been created.	No	bin-release

Attribute	**Description**	**Required?**	**Default Value**
failonerror	Indicates whether compilation errors cause the build to fail.	No	true
verbose	The <fb.exportReleaseBuild> task outputs additional information. For example, it lists the files that were packaged into the AIR file and how long each step of the process took.	No	false
package	For AIR projects only: Indicates whether to package the result into a .air or .airi file. If true, a .air or .airi file is created, and the temporary output directory (bin-release by default, set by the destdir attribute) is deleted. If false, a .air or .airi file is not created, and the intermediate directory remains intact after compilation.	No	true
destfile	For AIR projects only: The filename of the .air or .airi file to create. You can specify a relative path or absolute path. If you specify a relative path, the path is relative to the root of the project.	No	appname.air or appname.airi (in the project root)
certificate	For AIR projects only: The path to the certificate used to sign the AIR file.	No	If omitted, an unsigned .airi file is generated, which can be signed later.
password	For AIR projects only: The password for the certificate that is used to sign the AIR file. If this argument is omitted, an error message displays. **Caution**: Specifying a literal value for a password can compromise security.	No	n/a
timestamp	For AIR projects only: Indicates whether the generated AIR file includes a timestamp.	No	false

Export Release Build wizard

When you run the Export Release Build wizard (Project > Export Release Build), the settings you make in the wizard are saved in the .actionScriptProperties file. A command line build that uses fb.exportReleaseBuild task picks up the settings from the wizard. The Export Release Build wizard saves the following settings:

- View Source

  The source files you specify for View Source are saved. If you specify the publishsource parameter to fb.exportReleaseBuild, then the wizard includes these files as viewable source files.

  Important: *For server projects, you can select the services folder when exporting source files. Exporting files that implement services has security implications. These files can expose access to your database, including user names and passwords. See Exporting source files with release version of an application.*

- For AIR projects, any additional output files that you specify in the wizard to include with the AIR or AIRI file.

Running command line builds on Linux and other platforms

The <fb.exportReleaseBuild> task is only supported on Windows and Mac platforms.

However, if you are writing a build script for another platform, use the `-dump-config` option to the mxmlc or compc compiler to write compiler configuration settings to a file. You can then use the -load-config option to read the configuration options.

Modify the configuration settings in the file as necessary. For example, change <debug>true</debug> to <debug>false</debug> if your nightly build is supposed to do a release build.

Run a command line build using Flash Builder compiler settings

1 In Flash Builder, select Project > Properties > Flex Compiler

2 In Additional Compiler Arguments, specify the following argument:

   -dump-config *pathname*, where *pathname* specifies the absolute path to a file on your system.

3 Apply the changes in the Project window.

   The compiler settings are written to the specified file. Remove the -dump-config argument after you have verified that the file has been written.

4 Modify the configuration settings as necessary.

5 In your build script, run the compiler so it includes the saved compiler settings:

   mxmlc -load-config *pathname*

Limitations to command line builds

There are a few limitations to running command line builds using the <fb.exportReleaseBuild> task.

Running command line builds on Mac platforms using Eclipse 3.4

On Eclipse 3.4 on Mac platforms, headless build fails if the Eclipse installation path contains spaces.

Running command line builds on 64-bit platforms

Flash Builder runs on platforms that implement 32-bit Java. To run a command line build on platforms that support 64-bit Java (for example, Mac OS X Snow Leopard), add `-d32` to the command-line options that are passed to Java. For example:

```
java -d32 ...
```

Refactor enhancements

Flash Builder provides enhanced support for refactor/move operations. Flash Builder supports the following targets for refactoring:

- MXML components
- Top-level definitions for ActionScript files and elements, such as class, interface, function, variable, and namespace

When performing a refactor/move operation, Flash Builder opens a dialog that allows you to specify a destination for the move operation and whether to update string literal references. A preview window is available highlighting the original and refactored source of the move operation.

Refactor/move is available from the MXML and ActionScript editors, and also from the Flex Package Explorer. In the editors, place the cursor on the element you want to move. In the Flex Package Explorer, highlight the element you want to move. Then, from the Flash Builder menu, select Source > Refactor > Move.

Support for projects created with Catalyst

Flash Builder provides development support to application designers using Adobe® Flash® Catalyst™. Catalyst exports a project as an FXP file. Catalyst exports components in as an FXPL file. An FXPL file is a library package. The FXP and FXPL files can then be imported into Flash Builder for development. For FXP files, the resulting project is a Flex web project that runs in Adobe Flash Player. An FXPL file contains a library file. You can import an FXPL files as a Flex library project or you can import the contents into an existing Flex project.

You can create an Adobe AIR project from a Catalyst project. Import the FXP file for the Catalyst project into Flash Builder. Convert the application type for the project from Web (runs in Adobe Flash Player) to Desktop (runs in Adobe AIR). See "Changing Flex projects to Adobe AIR projects" on page 37.

Importing a Catalyst FXPL project

A Catalyst FXPL project is a library project created by Adobe Catalyst. When you import an FXPL project, you have the option to import the contents into another Flex project or Flex library project.

This feature is designed to help developers working with Catalyst application designers. However, you can use this feature to import the contents of any library project into another Flex project or Flex library project.

The menus available for this procedure vary slightly for the plug-in configuration of Flash Builder. This procedure assumes that you are importing a library project.

1 From the Flash Builder menu, select File > Import FXP.

   If you have the plug-in version of Flash Builder, select File > Import > Flash Builder > Flash Builder Project.

   You can also use the context menu for the Package Explorer to import a project.

2 Select File and navigate to the location of the file.

3 Specify the import method:

   - Import a new copy of project: Flash Builder appends a numeric identifier to the project name. Previous versions of the project are preserved.

     In the Extract To field, specify a location in which to extract the file. Typically, this location is a directory in your Flash Builder workspace representing a project folder. You can specify a new project folder or overwrite an existing project folder.

   - Import contents into existing project.

     For Source Folder, browse to a `src` folder of an existing project. For Package, browse to an existing package or specify a new package name for the contents.

   - Overwrite existing project: If a project by the same name exists in the workspace, you can overwrite the existing project.

     Select the project to overwrite. The previous version of the project is permanently removed.

4 Click Finish.

   When importing FXPL files, Flash Builder attempts to resolve references to fonts in the FXPL file. See "Resolving font references when importing Catalyst projects" on page 43.

Resolving font references when importing Catalyst projects

When importing an FXP project created with Adobe Catalyst, the imported project can contain references to fonts that are not available on your system.

The Import wizard provides the option to fix font references using CSS. If you select this option, Flash Builder imports the Catalyst style sheet `Main.css`. `Main.css` contains references to the fonts used in the project.

If you get compile errors from the fonts referenced in the style sheet, fix the references in the style sheet with fonts available on your system.

Catalyst FXPL projects do not contain style sheets. Flash Builder attempts to correct any references to fonts when importing an FXPL file. If Flash Builder cannot find a corresponding font on the target system, the original font references are left intact. For FXPL projects, font references that Flash Builder cannot resolve are discovered at runtime. There is either a font substitution or a runtime error for unresolved font references.

***Note:** For FXPL files, Flash Builder modifies the* `fontFamily` *attribute in MXML files when it attempts to resolve font references.*

Support for Flash Professional projects

Use Flash Professional projects to access Flash FLA or XFL files created with Flash Professional CS5. This feature allows Flash Professional developers to take advantage of the editing and debugging environment available with Flash Builder. The features of Flash Professional projects are only available in Flash Builder if you have installed Flash Professional CS5.

Typically, you create a project and files in Flash Professional. Then you create a corresponding project in Flash Builder to edit and debug the files. When editing the files in Flash Builder, you can set breakpoints in the project's ActionScript files. Breakpoints set in files that are in the Flash Professional project are recognized by the Flash Professional debugger when you call Debug Movie.

You can launch Flash Professional from Flash Builder to publish and run the files. You can also launch the Flash Professional debugger from Flash Builder.

Creating a Flash Professional project

1. Select File > New Flash Professional Project.
2. Navigate to the target FLA or XFL file for the project.

   The name of the file becomes the name of the project.
3. Specify a project location:

   You can use either the default project location in the workspace or navigate to a new project location.
4. Click Finish.

   Flash Builder opens the new project in the Package Explorer. The folder containing the target FLA file is accessible. The selected FLA file becomes the target file in the project. ActionScript files that are dependent to the target files are available for editing.

   If Flash Professional is not running, Flash Professional starts.

Working with Flash Professional projects in Flash Builder

You can do the following with source files in a Flash Professional project:

- Edit the ActionScript files that are dependent to the target FLA file.
- Debug the file in the Flash Builder debugger or Flash Professional Debugger:

To debug in Flash Builder, select Run > Debug *file* or click the Debug button from the toolbar.

To debug the file in Flash Professional, select Run > Debug Movie or click the Debug Movie button in Flash Builder. Breakpoints set in Flash Builder are recognized in the Flash Professional debugger.

- Publish the file in Flash Professional CS5:

  Select Project > Publish Movie or click the Publish in Flash Professional button from the toolbar.

- Run the file in either Flash Builder or Flash Professional:

  To run the file in Flash Builder, select Run > Run *file* or click the Run button from the toolbar.

  To run the file in Flash Professional, select Run > Test Movie or click the Test Movie button from the toolbar.

Setting project properties for Flash Professional projects

1. Select Project > Project Properties > Flash Professional.
2. Select Add to add additional files to the project.

   A project can have only one target FLA or XFL file as the default target file. Use the Set as Default button to specify the default target file for the project.

3. Click OK.

Generating accessor functions

Get and set accessor functions (getters and setters) let you keep class properties private to the class. They allow users of the class to access those properties as if they were accessing a class variable (rather than calling a class method).

Flash Builder can generate ActionScript get and set accessor functions for class variables. When you generate getters and setters, Flex Builder provides the following options:

- Make the class variable private.

  Typically, class variables have private access.

- Rename the class variable, suggesting a leading underscore for the variable name.

  By convention, private class variables have a leading underscore.

- Rename the accessor functions.
- Specify whether to generate both getter and setter accessor functions.
- Specify the placement of the accessor function in any of the following locations:
  - Before the first method
  - After the last method
  - Before variable declarations
- Preview the code that will be generated.

Generate get or set accessor functions

1. With an ActionScript file open in the Source Editor, place the cursor on a class variable.
2. Select Source > Generate Getter/Setter from either the Flash Builder menu or the context menu.
3. In the Generate Getter/Setter dialog, specify details for the accessor functions and click OK.

***Note:** If you want to view the code that will be generated, select Preview before clicking OK.*

Customizing File Templates

Flash Builder allows you to customize the default information contained in new MXML, ActionScript, and CSS files. Examples of information you can specify include variables for specifying author and date, variables for opening and closing tags and attributes, variables for various ActionScript declarations, namespace prefixes, and just about any content you want to include in a template file. File templates are especially useful for specifying introductory comments and copyright information.

The content of a new file is specified in a file template available from Preferences > Flash Builder > File Templates. Templates are available for the following types of files:

ActionScript	ActionScript file ActionScript class ActionScript interface
MXML	MXML web application MXML desktop application MXML component MXML module MXML skin ItemRenderer for Spark components ItemRenderer for MX components ItemRenderer for MX DataGrid ItemRenderer for Advanced DataGrid ItemRenderer for MX Tree
FlexUnit	FlexUnit TestCase class FlexUnit TestSuite class FlexUnit4 TestCase class FlexUnit4 TestSuite class
CSS	CSS file

After modifying a template, you can export the template so it can be shared with other members of your team.

Modify a file template

1 Select Preferences > Flash Builder > File Templates

2 Expand the file categories and select a file template to modify.

3 Select Edit and modify the template.

You can type directly in the Template editor or select Variables to insert pre-defined data into the template.

4 Click OK to save the changes.

Changes apply to new files.

Exporting and Importing File Templates

1 Select Preferences > Flash Builder > File Templates

2 Expand the file categories and select a file template.

3 Select Export to export a template to the file system, or Import to import a previously exported template.

Templates are exported as XML files.

Restoring Defaults

Note: *Restoring defaults restores all file templates to the default values. You cannot restore a single template to the default value.*

❖ To restore the default templates, open Preferences > Flash Builder > File Templates and select Restore Defaults

Template Variables

Template Variables for All File Types

Variable	**Description**	**Example**
${date}	Current date	Feb 15, 2009
${year}	Current year	2009
${time}	Current time	3:15 PM
${file_name}	Name of the newly created file	HelloWorld.mxml
${project_name}	Name of the Flex or ActionScript project	Hello_World_Project
${user}	Username of the author	jdoe
$$ ${dollar}	Dollar symbol	$

Template Variables for MXML Files

Variable	**Description**	**Example**

${application}
 	
${component}
 	
${module}
	Specifies the application, component, or module MXML tag names.
	
	For a web application, ${application} expands to "Application."
	
	For a desktop application, ${application} expands to "WindowedApplication."
	
	${component} expands to "Component."
	
	${module} expands to "Module."
	
	These tags are typically used to position the starting and closing tags of a file.
		The following:
		
		`<${application}`
		`${xmlns}${wizard_attributes}${min_size`
		`}>`
		`${wizard_tags}`
		
		`</${application}>`
		
		expands to:
		
		`<s:Application`
		`xmlns:fx="http://ns.adobe.com/mxml/200`
		`9"`
		`xmlns:s="library://ns.adobe.com/flex/s`
		`park"`
		`xmlns:mx="library://ns.adobe.com/flex/`
		`halo" minWidth="1024" minHeight="768">`
		`<s:layout>`
		`<s:BasicLayout/>`
		`</s:layout>`
		
		`</s:Application>`

${xml_tag}	XML version	`<?xml version="1.0" encoding="utf-8"?>`
${xmlns}	Resolves to the namespace definition, based on the project's Flex SDK type and the namespace prefix defined in Preferences.	For a Flex 4 SDK project: xmlns="http://ns.adobe.com/mxml/2009"
${min_size}	Minimum size of an MXML web application.	`minWidth="1024" minHeight="768"`

Variable	**Description**	**Example**
${ns_prefix}	Namespace prefix for the project's Flex SDK. You cannot change the default values for this variable.	For Flex 3: `mx:` For Flex 4: `fx:`
${wizard_attributes}	Specifies the position of the attributes defined by the New *File* wizard.	For a new web application: ${application} ${xmlns}${wizard_attributes}> expands to: <Application xmlns="http://ns.adobe.com/mxml/2009" layout="vertical">
${wizard_tags}	Specifies the layout property for containers defined by the New *File* wizard	For a new application using Flex 4 SDK: <s:layout> <s:BasicLayout/> </s:layout>

Template Variables for ActionScript Files

Variable	Description	Example
${package_declaration}	Generates the package declaration.	For a file in the com/samples package, generates: package com.samples
${import_declaration}	For a new ActionScript class or ActionScript Interface, generates required import declarations .	For a subclass of TextBox, generates: import flex.graphics.TextBox;
${interface_declaration}	For a new ActionScript interface, generates the interface declaration.	For a new Interface that extends IButton interface, generates: public interface IMyButton extends IButton
${class_declaration}	For a new ActionScript class, generates the class declaration.	For a new subclass of CheckBox, generates: public class MyCheckBox extends CheckBox

${class_body}	Generates all the required statements for a new class.	For a new subclass of Button that implements the IBorder interface, generates the following for the class body:
		
		`public function MyButton()`
		`{`
		`   super();`
		`}`
		`public function get`
		`borderMetrics():EdgeMetrics`
		`{`
		`   return null;`
		`}`

${interface_name}
 	
${class_name}
 	
${package_name}
	Specifies the interface, class, or package name.
	
	Typically used when generating comments.
		For example, the following template specification:
		
		`/*`
		` * ${class_name} implements. . .`
		` */`
		
		generates the following code:
		
		`/*`
		` * MyButton implements. . .`
		` */`

Template Variable for CSS Files

Variable	Description	Example

${css_namespaces}	Defines namespaces for Spark and Halo style selectors.	Default values for Flex 3:
		
		`""`
		
		(In Flex 3, namespace declarations are not required in CSS files)
		
		Default values for Flex 4:
		
		`@namespace s`
		`"library://ns.adobe.com/flex/spark";`
		`@namespace mx`
		`"library://ns.adobe.com/flex/halo";`

Template File Example

The following listing shows an example of an MXML Component file template, followed by a new MXML Component file generated from the template.

Example File Template for an MXML Component file

```
${xml_tag}
<!--
* ADOBE SYSTEMS Confidential
*
* Copyright ${year}. All rights reserved.
*
* ${user}
* ${project_name}
* Created ${date}
*
-->
<${component} ${xmlns}${wizard_attributes}>
    ${wizard_tags}

    <${ns_prefix}Script>
    <![CDATA[

    ]]>
    </${ns_prefix}Script>
</${component}>
```

New MXML Component file generated from the example template

```
<?xml version="1.0" encoding="utf-8"?>
<!--
* ADOBE SYSTEMS Confidential
*
* Copyright 2009. All rights reserved.
*
* jdoe
* FileTemplates
* Created Jul 13, 2009
*
-->

<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:s="library://ns.adobe.com/flex/spark"
xmlns:mx="library://ns.adobe.com/flex/halo" width="400" height="300">
    <s:layout>
        <s:BasicLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[

        ]]>
    </fx:Script>
</s:Group>
```

Generating event handlers

Flex applications are event-driven. User interface objects respond to various events, such as a user clicking a button or the initialization of an object is complete. You write event handlers in ActionScript code that define how the UI object responds to the event.

Flash Builder provides event handler assistance that generates the event handler functions for a UI object. Within the generated function, you write the code that defines the object's behavior in response to the event.

You access event handler assistance in three ways:

- Flex Properties View
- Context menu for an item in Design View
- Content assist for an item in Code View

About generated event handlers

When Flash Builder generates an event handler function, it places the event handler in the first Script block of the file. The function is placed at the end of the Script block. The generated event handler has protected access and accepts the appropriate subclass of Event as its only parameter.

Flash Builder generates a unique name for the event handler based on the object's class name, properties, or a custom name you specify. If you do not specify a custom name, the name is generated according to the following process:

- If there is no id or label property for the object, Flex Builder generates a unique name, based on the object's class name.
- If an id property is present, Flash Builder bases the name on the id property. Otherwise, it bases the name on the label property.

You provide the body of the event handler. The following code block shows generated event handlers for a Button.

```
. . .
<Script>
    <![CDATA[
        protected function myButton_clickHandler(event:MouseEvent):void
        {
            // TODO Auto-generated method stub
        }

        protected function myButton_clearHandler(event:Event):void
        {
            // TODO Auto-generated method stub
        }
    ]]>

</Script>
<Button label="Click" id="myButton" click="myButton_clickHandler(event)"
    clear="myButton_clearHandler(event)"/>
. . .
```

Flash Builder designates a default event for each UI item. For example, the default event for a Button is the click event. You can specify the event handler for the default event in the Standard View of the Flex Properties View. To specify event handlers for all events of a UI items, use the Category View.

You can also use content assist in the Source View to generate event handlers.

Generating event handlers using the Flex Properties View

1 In Design View, select an item and then select Standard View in the Properties Inspector.

An editing field for the default event handler is visible in the Common area.

2 To generate an event handler for the default event, do the following:

- In the “On *event*” text field, specify a name for the event (optional) and then click the icon.

  If you do not specify a name, Flash Builder generates a unique name for the event.

  The editor switches to code view, with the body of the generated event handler highlighted. Type in your implementation for the event.

3 To generate an event handler for any event for a selected item do the following:

a Select Category View and expand the Events node to view all the events for the item.

b (Optional) Double-click the name of the event to activate the text box for the event handler name, and type the name for the event handler.

c Click the icon in the Value field to create the event handler.

The editor switches to code view, with the body of the generated event handler highlighted. Type in your implementation for the event.

Note: *When specifying a name for the event handler, you have the option to specify an event parameter. If you do not specify the event parameter, Flash Builder generates the parameter with an appropriate event type.*

Generating event handlers using the context menu for an item

1 In Design View, open the context menu for an item.

2 Perform one of the following actions:

- Select the default event for the item.

  For example, for a Button you would select Generate Click Handler.

- Select Show All Events to open the list of events in the Properties Inspector.

  Specify an event handler from the Property Inspector.

The editor switches to code view, with the body of the generated event handler highlighted. Type in your implementation for the event.

Generating event handlers using content assist

1 In an MXML block in code view, create a component, but do not specify any events.

Make sure to include the closing tag for the component.

2 Enable content assist for the properties of a component by typing a space after the class name.

3 From the list of selected properties, select an event (for example, doubleClick).

4 Select Generate Event Handler.

Flash Builder generates a unique name for the event handler and places the event handler in the Script block.

You must accept the generated name for the event handler. If you specify a custom name, then Flash Builder does not generate the event handler.

***Note:** Flash Builder generates event handlers if the closing tag for a component is present. If the closing tag is not present, Flash Builder generates only the signature for the event handler. You must then create the event handler function in the Script block.*

ASDoc support

Flash Builder displays ASDoc comments in the MXML and ActionScript editors when you use Content Assist or hover over a component or a type in the editor. Flash Builder ASDoc support includes user-generated ASDoc comments as well as ASDoc comments contained in SWCs.

ASDoc comments in Content Assist

As you enter code into the Flash Builder editors, Content Assist prompts you with a list of options for completing your code expression (commonly referred to as *code hints)*. For example, in an MXML document you are prompted with the list of tags that can be added at the current location.

Content Assist now also displays any ASDoc comments available for the MXML component or ActionScript code you are authoring. ASDoc comments are also available by hovering over an MXML component or ActionScript type.

Generating SWCs containing ASDoc comments

Need to document this procedure.

Providing ASDoc comments for custom components

You can document your custom components by adding ASDoc comments to the code that implements them. ASDoc comments will then be available with Content Assist in the MXML and ActionScript editors.

Flash Builder supports the following HTML tags for ASDoc comments used for Content Assist.

b br br/ div h1, h2, h3, h4, h5 p	dl dt dd li ul pre

Flash Builder supports the following tags for ASDoc comments used for Content Assist.

All types	@see @deprecated @langversion @playerversion @productversion @author
Functions	@param @return @throws @event
Variables	@default
Events	@eventType

Applying themes

Themes allow you to implement a more personalized appearance to your applications. Flash Builder provides several themes from which you can choose. You can import additional themes or create your own themes.

Themes provided by Flash Builder include a set of Spark themes and a set of Halo themes. The default theme for Flex 4 components is Spark. Halo is the default theme for Flex 3.

For more information on theme support in Flex see About themes.

Specifying a theme

Specify themes on a project basis. After specifying a theme for a project, all applications in the project share the same theme.

1 Open the Select Project Theme dialog from either Design View or Source View of the MXML Editor:
   - (Design View) Select the Appearance Panel. Then select the Current Theme.
   - (Source view) From the Flash Builder menu, select Project > Properties > Flex Theme
2 Select a theme and then click OK.

Importing themes

You can use Flash Builder to import themes. The files for a theme must be enclosed in a folder. All required files for a Flex theme must be present.

The name of the theme is determined by the name element in the `metadata.xml` file contained within the theme folder. If the name element is not specified, or if `metadata.xml` is not present, then the name of the theme folder becomes the name of the theme.

For more information on the required format for Flex themes, see “Creating themes” on page 56.

Flash Builder themes can be in the following formats:

- Theme ZIP file

  Extract the contents of the ZIP file before importing the theme. The extracted contents should contain all required files.

- CSS or SWC file for a theme

  The CSS or SWC file must be in a folder containing all required files for a Flex theme. When you import a theme using Flash Builder, you select either the CSS or SWC file for the theme.

- MXP file

  You can use Adobe Extension Manager CS4 to package files for a Flex themes in an MXP file. The theme can then imported into Flash Builder using the Extension Manager.

  For more information on packaging themes in an MXP file, see “Creating an extension file (MXP file) for a Flex theme” on page 57.

Importing Flex themes using Flash Builder

1 Open the Select Project Theme dialog from either Design View or Source View of the MXML Editor:
   - (Design View) Select the Appearance Panel. Then select the Current Theme.
   - (Source view) From the Flash Builder menu, select Project > Properties > Flex Theme
2 Select Import Theme, navigate to the folder containing the theme to import, select the CSS or SWC file, and click OK.

Importing Flex themes packaged in an MXP file

1 If you have not already done so, import Adobe Flash® Builder™ 4 into Adobe Extension Manager CS4:

   From Adobe Extension Manager, select File > Import Product.

2 Open Adobe Extension Manager and select Flash Builder 4.

3 Select File > Install Extension, navigate to the MXP file for the theme, and click Open.

After you accept the license, Adobe Extension Manager installs the theme into Flash Builder. The theme is now available in Flash Builder from the Select Project Theme dialog.

***Note:** You can also double-click the MXP file to invoke Adobe Extension Manager, which then automatically installs the theme.*

Downloading themes

You can download themes that can then be imported into Flash Builder.

Downloading Flex themes

1 Open the Select Project Theme dialog from either Design View or Source View of the MXML Editor:

- (Design View) Select the Appearance Panel. Then select the Current Theme.
- (Source view) From the Flash Builder menu, select Project > Properties > Flex Theme

2 Select Find More Themes.

Flash Builder opens your default web browser to a page containing themes to download. You can also navigate to any other site containing themes for Flex that you can download.

3 Select a Flex theme to download.

After you download the theme, you can import the theme, as described in “Importing themes” on page 55.

Creating themes

You can create your own themes and import them into Flash Builder. A Flex theme typically contains the following files:

- SWC, SWF, CSS, PNG, JPEG, and other files that make up your theme.

  The files that make up the theme can vary, but must include a SWC or CSS file.

- `preview.jpg` file

  The preview image file for the theme. If your theme folder does not contain `preview.jpg`, then Flash Builder uses a default preview image for the theme.

- `metadata.xml` file

  Contains information about the theme, including which versions of the SDK the theme is compatible with. If your theme folder does not contain this file, then Flash Builder creates one when importing the theme.

Typically you package a theme in ZIP file, but the ZIP file must be extracted before you can import the theme into Flash Builder. You can also package the theme files in an Adobe Extension Manager file (MXP file), and use Adobe Extension Manager to import the theme into Flash Builder.

For more information, see About themes.

Metadata.xml file

The following table lists the elements that can be included in `metadata.xml`.

Element Name	**Description**
Name	The name of the theme that appears in Flash Builder. When importing a theme using Flash Builder, you can override the name specified in the `metadata.xml` file.
Category	Author of the theme. The category under which the theme is displayed in Flash Builder.
sdks	Specifies the Flex SDK versions for which the theme is compatible. This is a parent element for minVersionInclusive and maxVersionExclusive. If the sdks element is absent, then the theme is valid for all SDKs.
minVersionInclusive	Earliest Flex SDK version for which this theme is compatible. If absent, then this theme is compatible with all earlier versions of the Flex SDK.
maxVersionExclusive	Latest SDK version for which this theme is compatible. If absent, then this theme is compatible with all later versions of the Flex SDK.
mainFile	Top-level file for implementing the theme. This file can reference other files in the theme. For example, a CSS file could reference a SWC or SWF file. The -theme compiler argument references the specified file.

The following example shows a typical `metadata.xml` file for a theme created by Company ABC.

```
<theme>
    <name>WindowsLookAlike</name>
    <category>ABC</category>
        <sdks>
            <minVersionInclusive>2.0.1</minVersionInclusive>
            <maxVersionExclusive>4.0.0</maxVersionExclusive>
        </sdks>
    <mainFile>WindowsLookAlike.css</mainFile>
</theme>
```

According to the `metadata.xml` file, the theme is compatible with the Flex 2.0.1 SDK. It is also compatible with SDKs up to, but not including, Flex 4.0.0. When this theme is selected, `WindowsLookAlike.css` is the file that will be added to the `-themes` compiler argument.

Creating an extension file (MXP file) for a Flex theme

You can use Adobe Extension Manager CS4 to create an extension file (MXP file) for a Flex theme. The MXP file can be imported into Flash Builder using Adobe Extension Manager CS4.

Place all your theme files in a staging folder, and then create an extension installation file (MXI file), which is used by Extension Manager to create the MXP file. For information on the format of an MXI file, refer to the Extension File Format document.

When creating the MXI file, you specify destination paths for each of the theme's files. The destination paths are in this format:

```
$flexbuilder/<Theme Name>
```

- `$flexbuilder` is defined in the Flash Builder configuration file, `XManConfig.xml`. Extension Manager expands `$flexbuilder` according to this definition. `XManConfig.xml` is at the following location on your file system:

  `/<Install Dir>/Flash Builder 4/configuration/XManConfig.xml`

- <*Theme Name*> is the name of the folder that will contain the Flex theme.

Creating an MXP Extension file for a Flex theme

1 Place all the files for the theme, including the MXI file, in a staging folder.
2 From the Extension Manager, select File > Package Extension.
3 Navigate to the extension installation file and select it.
4 Navigate to a location for the package file, and name it using the extension .mxp.

   You can then test the extension file by installing it using the Extension Manager.

Adding additional themes

You can specify more than one theme file to be applied to an application. If there are no overlapping styles, both themes are applied completely. There are other considerations when adding additional themes, such as the ordering of the theme files.

To add additional themes, use the command line compiler, `mxmlc` with the `theme` compiler option to specify the path to the theme files.

Using themes provides details on specifying compiler arguments and the ordering of theme files.

Support for Flex 4 View States

You can use Adobe® Flash® Builder™ to create applications that change their appearance depending on tasks performed by the user. For example, the base state of the application could be the home page and include a logo, sidebar, and welcome content. When the user clicks a button in the sidebar, the application dynamically changes its appearance (its *state*), replacing the main content area with a purchase order form but leaving the logo and sidebar in place.

In Flex, you can add this kind of interaction with view states and transitions. A *view state* is one of several views that you define for an application or a custom component. 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.

About view states and transitions

A *view state* is one of several layouts that you define for a single MXML application or component. You create an application or component that switches from one view state to another, depending on the user's actions. You can use view states to build a user interface that the user can customize or that progressively reveals more information as the user completes specific tasks.

Each application or component defined in an MXML file always has at least one state, the *base state*, which is represented by the default layout of the file. You can use a base state as a repository for content such as navigation bars or logos shared by all the views in an application or component to maintain a consistent look and feel.

You create a view state by modifying the layout of an existing state or by creating a completely new layout. Modifications to an existing state can include editing, moving, adding, or removing components. The new layout is what users see when they switch states.

For a full conceptual overview of view states, including examples, see View states.

Generally, you do not add pages to a Flex application as you do in an HTML-based application. You create a single MXML application file and then add different layouts that can be switched when the application runs. While you can use view states for these layouts, you can also use the ViewStack navigator container with other navigator containers.

When you change the view states in your application, the appearance of the user interface also changes. By default, the components appear to jump from one view state to the next. You can eliminate this abruptness by using transitions.

A *transition* is one or more visual effects that play sequentially or simultaneously when a change in view state occurs. For example, suppose you want to resize a component to make room for a new component when the application changes from one state to another. You can define a transition that gradually minimizes the first component while a new component slowly appears on the screen.

Support for Flex 3 view states

Flash Builder provides support for view states as implemented in Flex 3. If you create a project that uses the Flex 3 SDK, the MXML editor in both Design and Source mode reverts to the Flex Builder 3 implementation. For information on editing states for the Flex 3 SDK, refer to the Flex Builder 3 documentation.

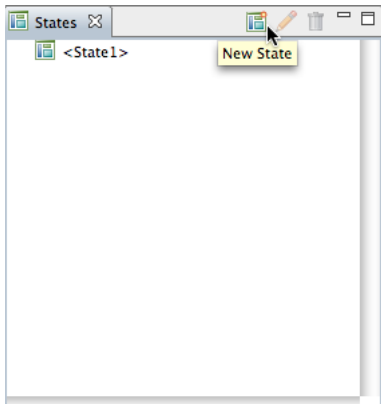
Creating a view state

By default, an application has a single view state, which you typically use as the base state. Use the Flash Builder States View to add additional states and to edit the layout and components for each state.

1 Using the layout tools in Flash Builder, design the layout of the base state of your application or component.

   For more information, see Building a user interface with Flash Builder.

2 In the States view (Window > Other Views > Flash Builder > States), click the New State button in the toolbar.



   The New State dialog box appears.

3 Enter a name for the new state.

4 Specify whether to create a state that is a duplicate of an existing state or to create a new, blank state. Click OK.

5 Use the layout tools in Flash Builder to modify the appearance of the state.

   You can edit, move, add, or delete components. As you make changes, the changes defining the new state become part of the MXML code.

6 Define an event handler that lets the user switch to the new state.

   For more information, see “Switching view states at run time” on page 60.

Setting a non-base state as the starting view state

By default, an application displays the base state when it starts. However, you can set another view state to be the state that displays when the application starts.

1 In States View (Window > States), double-click the view state that you want to use as the starting state.

2 In the Edit State Properties dialog box that appears, select the Set As Start State option and click OK.

Setting the view state of a component

If your application has multiple states, you can set the view state of a single component.

1 In Design View of the MXML editor, select a component in your layout.

2 In Properties View, use the In States field to select the states in which the component is visible.

Switching view states at run time

When your application is running, users need to switch from one view state to another. You can define event handlers for user controls so that users can switch states at run time.

The simplest method is to assign the `currentState` property to the click event of a control such as a button or a link. The `currentState` property takes the name of the view state you want to display when the click event occurs. In the code, you specify the `currentState` property as follows:

```
click="currentState='viewstatename'"
```

If the view state is defined for a specific component, you must also specify the component name, as follows:

```
click="currentState='componentID.viewstatename'"
```

For more information, see Using View States.

1 Ensure that the initial state has a clickable control, such as a Button control.

   In Design mode of the MXML editor, select the control and enter the following value in the On Click field in the Properties view:

   ```
   currentState='viewstatename'
   ```

   viewstatename is the name for the state.

2 If you want to switch to the base state, enter:

   ```
   currentState=''
   ```

   `''` is an empty string, represented by two single quotes.

3 To test that the states switch correctly in the application when the button is clicked, click the Run button in the Flash Builder toolbar.

You can define a transition so that the change between view states is smoother visually. For more information, see "Creating a transition" on page 63.

Creating view state groups

Flex provides support for view state groups. The `stateGroups` attribute of the `<States>` tag lets you group one or more states together. For example, if multiple components appear in the same set of view states, you can create a view state group that contains all these view states. Then, when you set the `currentState` property to any view state in the group, the components appears. For more information, with examples, see Defining view state groups.

Design mode of the MXML editor does not support editing state groups. Use Source mode to create and edit state groups. Source mode provides code hinting and a Show State pop-up menu to assist you in creating and editing state groups.

If you create view state group, be careful using Design View. If you delete a state using Design View, you can inadvertently leave a reference to a deleted component in a state group.

Deleting a view state

You can delete view states from an application using Design View of the MXML editor. However, if you have created a state group then use Source View to delete a state. This avoids inadvertently leaving a reference to a component in a deleted state.

1 In the Design View of the MXML editor, select the view state that you want to delete from the States View (Window > States).

2 Click the Delete State button on the States View toolbar.

Working with multiple states in an application

If you have an application that contains more than one state, Design mode of the MXML editor allows you to switch the view for each state, displaying only the components that defined for a specific state. For each component, you can specify the state in which it is visible.

Edit the component of a specific state

1 In Design View of the source editor, use the States View to add one or more additional states to an application.

2 Use the State drop-down menu to switch the view to the selected state.

3 Add, move, delete, or modify the components in the state.

   Changes you make to a specific state do not appear in other states unless you specify that the component appears in more than one state.

Specify that a component appears in multiple states

1 In Design View of the source editor, use the States View to add one or more additional states to an application.

2 For any component in a state, select the component.

3 In the Properties View, select which states the component appears.

   You can specify that the component appear in all states, or select one or more states in which the component appears.

   If you specify a specific state for a component, the component does not display in the editor when editing another state.

Be careful when editing applications that contain multiple states. Components might seem to “disappear” when you switch the editor to a state that doesn’t contain a component visible in another state.

Creating and editing view states in source code

Source mode of the MXML editor contains several features to help you edit source code for view states.

When an application declares view states, the MXML editor provides a Show State pop-up menu. When you select a specific view state in the Show State menu, components that do not appear in that state are de-emphasized in the editor.

The `includeIn` and `excludeFrom` properties for MXML components specify the view state or state group in which a component appears. Code hinting in the MXML editor assists you in selecting a view state or state group for these properties.

You can also use dot notation with component attributes to specify a view state in which the attribute applies. For example, if you want a Button component to appear in two view states, but also have the label change according to the view state, use the dot operator with the `label` property. Code hinting in the MXML editor assists you in selecting the view state. For example:

```
<s:Button label.State1="Button in State 1" label.State2="Same Button in State 2">
```

Example working with view states in source code

1 Create an application that contains more than one view state.

In Source mode of the MXML editor, add the following code after the `<s:Application>` tag.

```
<s:states>
    <s:State name="State1" />
    <s:State name="State2" />
    <s:State name="State3" />
</s:states>
```

Notice that the MXML editor adds a Show State pop-up menu after you define states in the application.

2 In Source mode, add the following Button components:

```
<s:Button includeIn="State1" label="Show State 2"
    click="currentState='State2'" />
<s:Button includeIn="State2" label="Show State 3"
    click="currentState='State3'" />
<s:Button includeIn="State3" label="Show State 1"
    click="currentState='State1'" />

<s:Button
    label.State1="All States: State 1 Label"
    label.State2="All States: State 2 Label"
    label.State3="All States: State 3 Label"
    x="0" y="30"/>
```

By default, the editor displays code for all states.

Note: *The click event handlers for the first three buttons cycle through the view states.*

3 Still in Source mode, select different view states from the Show State pop-up menu.

For components that are not visible in the selected state, the editor displays the code as light grey.

All the code is editable, but de-emphasizing components that do not appear in the selected view state assists in maintaining code for each view state.

4 Switch to Design mode for the MXML editor.

Using either the States View or the States pop-up menu, select different view states. The editor displays the components according to properties defined for the selected view state.

5 Run the application. Click the top button to cycle through the view states.

For more information on creating and editing states in source code, see Create and apply view states.

Creating a transition

When you change the view states in your application, the components appear to jump from one view state to the next. You can make the change visually smoother for users by using transitions. A transition is one or more effects grouped together to play when a view state changes. For example, you can define a transition that uses a Resize effect to gradually minimize a component in the original view state, and a Fade effect to gradually display a component in the new view state.

1 Make sure you create at least one view state in addition to the base state.

2 In Source View of the MXML editor, define a Transition object by writing a `<s:transitions>` tag and then a `<s:Transition>` child tag, as shown in the following example:

```
<s:transitions>
    <mx:Transition id="myTransition">
    </mx:Transition>
</s:transitions>
```

To define multiple transitions, insert additional `<s:Transition>` child tags in the `<s:transitions>` tag.

3 In the `<s:Transition>` tag, define the change in view state that triggers the transition by setting the tag's `fromState` and `toState` properties, as in the following example (in bold):

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

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

4 In the `<mx:Transition>` tag, specify whether you want the effects to play in parallel or in sequence by writing a `<mx:Parallel>` or `<mx:Sequence>` child tag, as in the following example (in bold):

```
<mx:Transition id="myTransition" fromState="*" toState="checkout">
    <mx:Parallel>
    </mx:Parallel>
</mx:Transition>
```

If you want the effects to play simultaneously, use the `<mx:Parallel>` tag. If you want them to play one after the other, use the `<mx:Sequence>` tag.

5 In the `<mx:Parallel>` or `<mx:Sequence>` tag, specify the targeted component or components for the transition by setting the property called `target` (for one target component) or `targets` (for more than one target component) to the ID of the target component or components, as shown in the following example:

```
<mx:Parallel targets="{[myVGroup1,myVGroup2,myVGroup3]}">
</mx:Parallel>
```

In this example, three VGroup containers are targeted. The `targets` property takes an array of IDs.

6 In the `<mx:Parallel>` or `<mx:Sequence>` tag, specify the effects to play when the view state changes by writing effect child tags, as shown in the following example (in bold):

```
<mx:Parallel targets="{[myVBox1,myVBox2,myVBox3]}">
    <mx:Move duration="400"/>
    <mx:Resize duration="400"/>
</mx:Parallel>
```

For a list of possible effects and how to set their properties, see Introduction to effects.

7 To test the transition, click the Run button in the Flash Builder toolbar, then switch states after the application starts.

Generating custom item renderers

Spark list-based controls, such as List and ComboBox, support custom item renderers. You can also use Spark item renderers with some MX controls, such as the MX DataGrid and MX Tree controls.

Use custom item renderers to control the display of a data item in a DataGroup, SkinnableDataContainer, or in a subclass of those containers. The appearance defined by an item renderer can include the font, background color, border, and any other visual aspects of the data item. An item renderer also defines the appearance of a data item when the user interacts with it. For example, the item renderer can display the data item one way when the user moves the mouse over the data item. It displays the differently when the user selects the data item by clicking on it.

Using Flash Builder, you can generate and edit item renderers. When Flash Builder generates item renderers, it uses one of the following templates:

- Spark components

  Use this template for Spark list-based controls, such as List and ComboBox.
- MX Advanced DataGrid
- MX DataGrid
- MX Tree

You can open the New MXML Item Renderer wizard from both Design mode and Source mode of the MXML editor. In the New MXML Item Renderer wizard, you specify a name and template for the item renderer. Flash Builder generates an MXML file that implements the item renderer.

Components in the application reference the generated item renderer using the `itemRenderer` property.

For details on creating and using item renderers, see Custom Spark Item Renderers.

Generate and edit an item renderer for an MX Tree component (Design mode)

This example generates an item renderer for an MX Tree component, showing you how to use a combination of Flash Builder views to edit the item renderer. It assumes that you are working in a Flex project using the default Spark theme.

1 Create an application file. In Design mode of the editor, add an MX Tree component to the application.

  Populate your Tree with data that can be displayed when you run the application.

2 From the context menu for the Tree, select Create Item Renderer.

  The New MXML Item Renderer dialog opens.

  You can also do the following to open the New MXML Item Renderer dialog.

  - In the Common section of the Properties view, select the icon near the Item Renderer Field field.
  - From the Flash builder menu, select New > MXML Item Renderer.

3 Specify the following in the New MXML Item Renderer Dialog:

  - Source Folder and Package for the generated item renderer declaration.
  - Name

The name for the item renderer class you are creating.

- Template

  Select the template to use when generating the item renderer.

4 Click Finish.

Flash Builder generates an ItemRenderer class file and opens it in Design mode of the MXML editor.

The ItemRenderer component is selected.

The normal state of the Tree is selected in States view.

5 For each state of the Tree, modify the appearance in the generated ItemRenderer class.

a Open Outline view:

Notice that the top-level node, MXTreeItemRenderer, is selected.

In the Style section of Properties view, modify the appearance of tree items.

b In Outline view, select other components of the MXTreeItemRenderer to modify the appearance of those components.

Notice that the Style section tools are not always available.

If the Style section tools are not available, then use Source mode of the editor to define the appearance. When you switch to Source mode, the source for the selected component in Outline view is highlighted.

6 Run the application to see how the ItemRenderer changes the appearance of the Tree.

Creating and editing item renderers (Source mode)

You can open the New MXML Item Renderer dialog directly in Source mode of the editor. For example, do the following to create an item renderer for a Spark List component.

1 In Source mode of the editor, place your cursor inside a `<s:List>` tag and type the following:

```
<s:List itemRender="
```

After you type the first quote for the item renderer class name, a context menu appears.

2 Double-click Create Item Renderer to open the New MXML Item Renderer dialog.

This dialog is the same dialog that opens in Design Mode.

See the instructions in "Generate and edit an item renderer for an MX Tree component (Design mode)" on page 64 for creating the item renderer.

3 Click Finish.

Flash Builder generates a new item renderer based on your selections in the New MXML Item Renderer dialog. The editor switches to the source for the newly generated class.

4 Edit the item renderer class.

Save your class file and application file.

ItemRenderer declaration

In Flash Builder, an ItemRenderer declaration is the file that implements the custom ItemRenderer for a component.

You can view the custom ItemRenderer declaration for selected components:

1. In Design mode of the MXML editor, select a component that you have implemented a custom item renderer for.
2. From the context menu for the component, select Open Item Renderer Declaration.

   The class implementing the item renderer opens in Source mode of the editor. You can also do the following to open the item renderer declaration:

   - Select the component in Design mode. In the Common section of the Properties view, select the icon near the Item Renderer field.
   - In Source mode, with the component selected, from the Flash Builder menu, select Navigate > Open Skin Declaration.

Creating and editing Flash components

Adobe Flash® Professional CS5 creates applications compatible with Adobe Flash Player 10. Adobe applications also support Flash Player 10, which means that you can import assets from Flash Professional CS5 to use in your applications. You can create controls, containers, skins, and other assets in Flash Professional CS5, and then import those assets into your application as SWC files. For information on creating components using Flash Professional CS5, see Flex Skin Design Extensions and Flex Component Kit for Flash Professional.

In the Design View of the Flash Builder editor, you can insert a new Flash component by adding a placeholder for a Flash component or container. You can invoke Flash Professional CS5 from Flash Builder to create the component or container. You can also invoke Flash Professional CS5 to edit previously created Flash components.

If your application contains an SWFLoader component to launch Flash movie files, you can launch Flash Professional CS5 to create or edit the associated FLA and SWF files.

Inserting a Flash component or Flash container

1. In the Flash Builder editor, select Design View and make sure the Components View is visible.
2. From the Custom folder in the Components View, drag either a New Flash Component or a New Flash Container to the design area.

   You can resize or position the component or container.
3. From either the context menu or the Standard View of the File Properties window, select Create in Adobe Flash.

   ***Note:** You can also double-click the component in Design View to create the item in Adobe Flash.*
4. In the dialog that opens, specify names for the class and the SWC file, then click Create to open Adobe Flash Professional CS5.
5. In Flash Professional CS5, edit the component or container. Select Done when you are complete to return to Flash Builder.

Editing a Flash component or Flash container

This procedure assumes you have previously inserted a Flash component or container into Flash Builder.

1. In Design View of the Flash Builder editor, select the Flash component or container you want to edit.
2. From either the context menu or the Standard View of the Flex Properties window, select Edit in Adobe Flash.

   ***Note:** You can also double-click the component in Design View to edit the item in Adobe Flash.*

3 In Flash Professional CS5, edit the component or container. Select Done when you are complete to return to Flash Builder.

Creating or editing a Flash movie associated with a SWFLoader component

This procedure assumes your application contains an SWFLoader component.

1 In Design View of the Flex editor, select the SWFLoader component.

2 From either the context menu for the component or the Standard View of the Flex Properties window, launch Flash Professional CS5 by doing one of the following:

- Select Create in Adobe Flash to create a new Flash movie associated with the SWFLoader component.
- Select Edit in Adobe Flash to edit the Flash movie associated with the SWFLoader component.

3 After you have finished editing the movie, select Done to return to Flash Builder.

Importing Flash CS3 Assets

You can add Flash components that were created in Adobe Flash CS3 Professional to your user interface.

Note: *Before you can create Flex components in Flash CS3, you must install the Flex Component Kit for Flash CS3. For more information, see the article Importing Flash CS3 Assets into Flex.*

1 Ensure that the Flash component is saved in the library path of the current project.

The library path specifies the location of one or more SWC files that the application links to at compile time. The path is defined in the Flex compiler settings for the project. In new projects the `libs` folder is on the library path by default.

To set or learn the library path, select the project in the Package Explorer and then select Project > Properties. In the Properties dialog box, select the Flex Build Path category, and then click the Library Path tab.

The library path can also be defined in the flex-config.xml configuration file in Adobe LiveCycle® Data Services ES.

2 Open an MXML file and add a Flash component in one of the following ways:

- In the MXML editor's Design mode, expand the Custom category of the Components view and drag the Flash component into the MXML file. For documents that are already open, click the Refresh button (the green circling arrows icon) to display the component after you insert it.
- In Source mode, enter the component tag and then use Content Assist to quickly complete the tag.

Debugger enhancements

New Debugger Features

Setting conditional breakpoints

You can specify conditions for breakpoints to stop the debugger from executing when specific conditions are met. When you set a conditional breakpoint, you specify an ActionScript expression that is evaluated during the debugging session. You configure the conditional breakpoint to halt execution for any of the following conditions:

- The expression evaluates to true.
- The value of the expression changes.

- A specified Hit Count has been reached.

Setting a conditional breakpoint

1 From the context menu for a breakpoint, select Breakpoint Properties.

2 In the Breakpoint Properties window, specify any of the following:

  - Enabled

    Toggle to enable or disable the breakpoint.

  - Hit Count

    Select Hit Count to enable a counter for the breakpoint. Specify a number for the Hit Count.

    If you specify both Hit Count and Enable Condition, the Hit Count is the number of times that the specified condition is met (either evaluates to true or the value of the condition changes).

    If you specify Hit Count only, then Hit Count is the number of times the breakpoint has been reached.

  - Enable Condition

    Select Enable Condition and enter an ActionScript expression to evaluate. See "Examples of expressions" on page 69 for information on types of expressions supported for evaluation.

    ***Note:** Flash Builder checks the syntax of the expression and notifies you of syntax errors. If you have an assignment operator in the expression, Flash Builder displays a warning.*

  - Suspend when:

    Specify when to halt execution, either when the expression for the condition evaluates to true or the value of the expression changes.

Using the Expressions view

Evaluation of watch expressions has been enhanced in this version of Flash Builder. Use the Expressions view to watch variables you selected in the Variables view and to add and evaluate watch expressions while debugging your applications.

While debugging, you can inspect and modify the value of the variables that you selected to watch. You can also add watch expressions, which are code expressions that are evaluated whenever debugging is suspended. Watch expressions are useful for watching variables that may go out of scope when you step into a different function and are therefore not visible in the view.

The Expressions view provides the following commands, which are available from the Variables view toolbar (as shown left to right):

Command	Description
Show Type Names	Shows the object types for items in the Expressions view.
Show Logical Structure	This command is not supported in Flash Builder.
Collapse All	Collapses all expressions in view.
Remove Selected Expressions	Removes the selected variable or watch expression.
Remove All Expressions	Removes all variables and watch expressions from the Expressions view.

You can also hover the mouse pointer over an expression or variable in the source editor to see the value of that expression or variable as a tooltip. You can add the expression to the Expressions view by right-clicking and selecting Watch from the menu.

Examples of expressions

The Flash Builder Debugger supports a wide range of simple and complex expressions. The following table lists examples of expressions that can be evaluated during a debugging session. This is not the complete list of expressions supported, but just a sampling of what you can do.

Examples of supported expressions

Expression	Description
myString.length	Returns the length of a string.
myString.indexOf('@')	Tracks the index of the '@' character.
"constant string".charAt(0)	Tracks the character at a specific position in a string. String constants are supported.
employees.employee.@name	employees is an XML variable. This type of expression is useful for debugging E4X applications.
x == null	Reserved words representing values in expressions.
user1 === user2	Most ActionScript operators are supported.
MyClass.myStaticFunc()	Functions resolved to a class.
this.myMemberFunc()	Functions resolved using the keyword `this`.
String.fromCharCode(33)	String is actually a function, not a class, and String.fromCharCode is actually a dynamic member of that function.
myStaticFunc()	Can be valuated only if myStaticFunc is visible from the current scope chain
myMemberFunc()	Can be valuated only if myMemberFunc is visible from the current scope chain.
Math.max(1,2,3)	Math functions are supported.
mystring.search(/myregex/i)	Regular expressions are supported.
["my", "literal", "array"]	Creation of arrays.
new MyClass()	Instantiation of classes.
"string" + 3	Correctly handles string plus Integer.
x >>> 2	Logical shift operations supported.
3.5 + 2	Performs arithmetic operations correctly.

Limitations of expression evaluation

There are some limitations to expression evaluation.

- Namespaces are not supported.
- Inline objects are not supported.
- The keyword `super` is not supported.
- Fully qualified class names are not supported.

For example, you cannot evaluate mx.controls.Button.

You can refer to the unqualified class name. For example, you can specify Button to refer to mx.controls.Button.

If a class name is ambiguous (two classes with the same name in different packages,) then you cannot control which class will be evaluated. However, you can specify:

```
getDefinitionByName("mx.controls.Button")
```

- Most E4X expressions can be evaluated, but E4X filter expressions are not supported.

  For example, you cannot evaluate `myxml.(@id=='3'))`.
- You cannot call functions that are defined as a variable.

Using watchpoints

When debugging an application, you can set watchpoints on specific instances of variables to halt execution when the watched variable changes value. Because watchpoints are set on a specific instance of a variable, you cannot set the watchpoint in the code editor. Instead, you set a watchpoint from the Variables view during a debugging session.

When setting watchpoints, keep in mind the following:

- When a debugging session ends, all watchpoints are removed.
- You cannot set watchpoints on getters, but you can set them on the field of a getter.

  For example, you cannot set a watchpoint on `width`, but you can set a watchpoint on `_width`.
- You cannot set watchpoints on local variables, but you can set watchpoints on members of local variables, as illustrated in the following code fragment.

```
public class MyClass
{
   // These are fields of a class, so you can set a watchpoint on
   // 'memberInt', and on 'memberButton', and on 'memberButton._width':
   private var memberInt:int = 0;
   private var memberButton:Button = new Button();

   public function myFunction():void {
      // You CANNOT set a watchpoint on 'i', because it is local:
      var i:int = 0;

      // You CANNOT set a watchpoint on 'someButton', because it is local;
      // but you CAN set a watchpoint on 'someButton._width':
      var someButton:Button = new Button();

...
}
```

- Execution halts for a watchpoint when the original value of an object instance changes.

  This differs from using an expression in a conditional breakpoint to halt execution whenever a variable changes value.

Setting watchpoints

❖ In a debugging session, there are two ways to set a watchpoint:

  - In the Variables view, open the context menu for a variable, and select Toggle Watchpoint
  - From the Flash Builder Run menu, select Add Watchpoint.

From the Add Watchpoint dialog, select the variable you want to watch.

The Variables view displays a “pencil icon” to indicate that a watchpoint has been set on that variable.

***Note:** If you attempt to set a watchpoint on a getter, Flash Builder opens a dialog suggesting a valid variable for the watchpoint. If you delete the suggested variable, the dialog lists all valid variables for the object.*

Using Run to Line

Flash Builder provides the Run to Line command to break out of a loop during a debugging session.

While debugging, you might find that your breakpoint is within a loop that repeats many times. To break out of this loop, use the Run to Line command, available from the Run menu.

Network Monitor

The Network Monitor is a useful tool for monitoring and debugging applications that access data services. The Network Monitor allows you to examine the data that flows between an application and a data service. It also examines XML, AMF, and JSON data, which are sent using SOAP, AMF, HTTP, and HTTPS protocols.

The Network Monitor is active in the Flash Development and Flash Debug perspectives.

Enabling network monitoring

You enable the Network Monitor for individual Flex projects. The monitor state (enabled or disabled) applies to all applications within that project. You cannot enable or disable the Network Monitor on an individual application basis.

By default the Network Monitor is not enabled. You enable the Network Monitor by selecting the Enable Monitor icon in the Network Monitor toolbar.

Enabling the Network Monitor

This procedure assumes you are in the Flex Development or Flex Debug perspective.

1 If the Network Monitor view is not open, from the Flash Builder menu select Windows > Other Views > Flash Builder > Network Monitor.

2 If the Network Monitor is not enabled, in the Network Monitor toolbar click the Enable Network Monitor button.

   This button is a toggle for enabling or disabling the Network Monitor.

Monitoring remote services

To monitor your application, run either the development or debug version of the application with the Network Monitor enabled.

In general, the Network Monitor captures and stores all event data until you either quit the application or explicitly clear the data. The events are displayed in chronological order.

Starting a monitoring session

1 Run either a development or debug version of the application that accesses remote services.

2 For each access to a remote service, the Network Monitor lists the following:

   - Time of the request

- Requesting service
- The operation and URL if applicable
- The time of the response
- Elapsed time

3 Select a column header to sort the returned data according to the values in that column.

Click the column again to invert the order of the data.

4 Select the request and parameter tabs at the bottom of the monitor to view the details about the request operation.

The actual data sent in the request, as well as other information about the request, can be viewed from these tabs.

5 Select the response and result tabs at the bottom of the monitor to view the details about the response.

The actual data sent in the response, as well as other information about the response, can be viewed from these tabs.

6 Double-click an entry to go to the source code for that operation.

The Flash Builder source editor opens with the relevant line of source code highlighted.

***Note:** For most events, the Network Monitor can correlate an event with the Flex source code. For some events that are triggered outside the scope of the Network Monitor, the monitor cannot find the Flex source code.*

7 Click the Save button on the Network Monitor toolbar to write all captured information to an XML file.

***Note:** Use the generated XML file to study the data offline. You cannot import the data from this file back into the Network Monitor.*

8 Click the Clear icon in the Network Monitor toolbar to remove all captured information from the monitor.

Suspending a monitoring session

You can suspend and resume network monitoring. Suspending and resuming a session applies to all applications in the Flex project. For example, you cannot suspend one application in the project and continue monitoring another.

1 Click the Suspend button in the Network Monitor toolbar to suspend the monitoring of a session.

2 Click the Resume button in the toolbar to continue monitoring the session.

Stopping a monitoring session

To stop monitoring a session, you disable the Network Monitor.

1 (Optional) Close the Network Monitor.

***Note:** Simply closing the Network Monitor does not stop the monitoring session. Monitoring is still active, even if the Network Monitor is closed.*

2 Click the Enable Network Monitor button.

This button is a toggle for enabling or disabling the Network Monitor.

***Note:** Disabling the Network Monitor applies to all applications in the Flex project.*

Support for HTTPS protocol

The Network Monitor supports monitoring HTTPS calls to a server certified by a certificate authority (CA) or that has a self-signed certificate.

To monitor calls over the HTTPS protocol, modify the default preference for the Network Monitor to ignore SSL security checks. Open the Preferences dialog and navigate to Flash Builder > Network Monitor.

Viewing Network Monitor data

The leftmost panel of the Network Monitor provides information about the source of the data. It displays the following information:

- Source URL for the data service
- The type of service displayed

  For example RemoteService, HTTPService, or WebService.
- The request time, response time, and elapsed time for the data request
- The name of the operation called from the data service.

The Network Monitor has two tabs for viewing data, allowing you to view the request data and response data.

For each request and response, you can view the data in Tree View, Raw View, or Hex view. Select the corresponding icon for each view to change how the Network Monitor displays the data.

- Tree View

  Shows the XML, JSON, and AMF data in a tree structure format. This is the default view for data.
- Raw View

  Shows the actual data that is transferred.
- Hex View

  Shows the data in hexadecimal format. Hex view is useful when debugging binary data sent over a network.

By default, the Network Monitor clears all recorded data with every launch of an application. However, you can change the default behavior and retain all monitor data. Retained monitor data includes the data from all applications in all projects. Open the Preference dialog and navigate to Flash Builder > Network Monitor. Deselect Clear Entries on Start.

Saving Network Monitor data

You can save Network Monitor data to an XML. Click the Save button in the Network Monitor view to save Network Monitor data.

Monitoring multiple applications

You can monitor multiple applications simultaneously. There are two scenarios for monitoring multiple applications:

- Monitoring multiple applications in the same project

  You can only have one Network Monitor per Flex project. When monitoring multiple applications in the same project, events from all applications appear in the monitor according to the time the event occurred.

  You cannot filter events in the monitor according to specific applications.
- Monitoring multiple applications in different projects

  You can open a Network Monitor for each active Flex project. Each Network Monitor is independent of the other monitor, displaying only the events for its specific project.

  Suspending or disabling a Network Monitor in one project does not apply to monitors in other projects.

Limitations of the Network Monitor

Be aware of the following limitations when monitoring network data:

- The Network Monitor does not support applications that were created using pure ActionScript and Library projects.
- The Network Monitor does not support the Real Time Messaging Protocol (RTMP). For example, you cannot monitor streaming video.

Profiler enhancements

Flash Builder provides usability enhancements to the Flex Builder Profiler that allows you to better diagnose why objects are kept alive in the memory heap of an application (memory leaks). For any object listed in the Memory Snapshot or Loitering Objects panel, you can now view all the shortest paths from the object to the garbage collector root (GC root). The number of paths to display is configurable.

In earlier versions of the Profiler, getting the shortest path information to GC root from the Object Reference panel was not obvious and involved tedious inspection of back references to objects.

Configuring preferences for object references in the profiler

Before running the profiler to diagnose memory leaks, you should first configure preferences for Object References.

1 From the Flash Builder preferences window, select Flex > Profiler > Object References.

2 In the Object References panel, select Show All Back Reference Paths.

   This preference ensures that you view all the paths for an object in memory.

3 Specify the maximum number of back reference paths to find.

   This is the number of paths displayed if you do not specify to view all paths.

Diagnosing objects in a memory snapshot

This procedure shows how to create a memory snapshot and view all of the shortest paths from an object in the snapshot to the GC root. It assumes that you have a Flex application that can be run from the Flex Profiler perspective.

1 Switch to the Profiler Perspective and select the Profile button to run the Profiler on the application.

2 In the Configure Profiler dialog, select Generate Allocation Object Stack Traces and click Resume.

   Enable Memory Profiling, Watch Live Data, and Enable Performance Profiling are selected by default.

   After a few moments, the Live Objects panel displays information on objects in memory.

3 In the Profile panel, select the running application and then select the Take Memory Snapshot button.

4 Double-click the memory profile snapshot node beneath the running application to open the Memory Snapshot panel.

5 In the Memory Snapshot panel, double-click a class to open the Object References panel.

6 Expand an instance of a class to view the paths captured in the memory snapshot.

7 Expand one or more paths and navigate through the stack trace to view the allocation trace for each element in the path.

   Use the information in the allocation trace to determine where an object is created, and remove references as needed so the garbage collector can remove the object.

Note: *This procedure shows how to get object reference information from the Memory Snapshot panel. You can get the same information from the Loitering Objects panel. You can view the Loitering Objects panel by taking two memory snapshots, and then clicking the Loitering Objects button.*

FlexUnit test environment

The FlexUnit test environment allows you to generate and edit repeatable tests that can be run from scripts or directly within Flash Builder. Flash Builder supports both FlexUnit 4 and Flex Unit 1 open source frameworks.

From Flash Builder, you can do the following:

- Create unit test cases and unit test suites

  Flash Builder wizards guide you through the creation of test cases and test suites, generating stub code for the tests.
- Run the test cases and test suites

  You can run test cases and test suites various ways from within Flash Builder or outside the Flash Builder environment. The results of the tests are displayed in a test application. Flash Builder opens a Flex Unit Results View for analysis of the test run.
- Navigate to source code from the Flex Unit Results View

  In the Test Results panel, double-click a test to open the test implementation.

  The Test Failure Details panel lists the source and line number of the failure. If the listed source is in the current workspace, double-click the source to go directly to the failure.

Creating FlexUnit tests

You can create FlexUnit test case classes and test case suites for the following types of projects:

- Flex project
- ActionScript project
- Flex Library project
- AIR project

When creating a test case class, you can specify the following options:

- A `src` folder in a Flex project for the class
- A package for the class
- The classes to test
- Methods to test for each specified class

A FlexUnit test case suite is a series of tests based on previously created test case classes, specified methods in those classes, and other test case suites.

Creating a FlexUnit test case class

When you create a FlexUnit test case class, Flash Builder generates an ActionScript file for the test case class, which it places in a package for test cases.

The following procedure assumes that you have created a project in Flash Builder in which you want to create and run Flex Unit tests.

1 Select the Flex project and then from the context menu, select New > Test Case Class.

   If you select an ActionScript class file in project, then that class is automatically selected for the FlexUnit test case in the New Test Case Class wizard.

2 In the New Test Case Class wizard, specify whether to create a class in the FlexUnit 4 style or FlexUnit 1 style.

3 Specify a name for the test case class.

4 (Optional) Specify a source folder and package for the test case class, or accept the defaults.

   The default source folder is the `src` folder of the current project. The default package is `flexUnitTests`, which is at the top level of the default package structure for the project.

5 (Optional) Enable the Select Class to Test toggle, and browse to a specific class. Click Next.

6 (Optional) Select the methods in the selected class that you want to test.

7 Click Finish.

   Code the test case you created. Use the generated code stubs as a starting point.

Creating a FlexUnit test case suite

This procedure assumes that you have previously created test case classes.

1 Select the Flex project and then create a test case suite from the context menus by selecting New > Test Suite Class.

2 In the New Test Suite Class wizard, specify whether to create a class in the FlexUnit 4 style or FlexUnit 1 style.

3 Provide a name for the test suite.

4 Navigate in the test suites and test cases to select the classes and methods to include in the test suite. Click Finish.

Customizing default FlexUnit test case classes and test case suite classes

You can customize the default FlexUnit test case classes and test case suite classes that are created by Flash Builder. Flash Builder uses file templates to create the default versions of these files.

The file templates for FlexUnit are available from the Preferences window at Flash builder > File Templates > FlexUnit. There are separate templates for FlexUnit1 and FlexUnit4 test case classes and test suite classes.

See "Customizing File Templates" on page 46 for information on how to modify the default file templates.

***Note:** `FlexUnitCompilerApplication.mxml` and `FlexUnitApplication.mxml` derive from the template for MXML Web Application or MXML Desktop Application. The template that is used depends on whether the Flex project is configured for a web application (runs in Adobe® Flash® Player) or a desktop application (runs in Adobe AIR®).*

More Help topics

Open source language reference for FlexUnit

Open source documentation for FlexUnit

Running FlexUnit tests

FlexUnit tests can be run from within Flash Builder or from outside Flash Builder using SWFs generated for the FlexUnit test. In either case, the results of the tests are displayed in the FlexUnit Results View.

You can also configure and save a FlexUnit test before running it.

By default, FlexUnit tests run in the Flash Debug perspective. You can launch tests from the Flash Development or Flash Profile perspectives, but Flash Builder switches to the Flash Debug perspective when running the test.

You can modify the default perspective for FlexUnit tests. Open the Preferences window, and navigate to Flash Builder > FlexUnit.

FlexUnit compiler application and FlexUnit application

When you create a FlexUnit test case, Flash Builder creates the following FlexUnit compiler application and a FlexUnit application:

- FlexUnitCompilerApplication.mxml
- FlexUnitApplication.mxml

Flash Builder uses these applications when compiling and running FlexUnit tests. Flash Builder places the applications, in the `src` directory of the project.

This application contains references to all FlexUnit test cases and test suites generated by Flash Builder. Flash Builder places all FlexUnit tests in the `<fx:Declarations>` tag of this application. You typically do not edit or modify this file directly.

Refresh the FlexUnit compiler application for the following circumstances:

- You manually add a test case.

  If you create a test case class without using the New Test Case wizard, refresh `FlexUnitCompilerApplication.mxml`. Place the new test case in the package with the other test cases.
- You rename a test case
- You delete a test case

Refresh `FlexUnitCompilerApplication.mxml`:

1. If the FlexUnit Results view is not open, select Windows > Other Views > FlexUnit Results. Click OK.
2. In the FlexUnit Results view, click the Refresh button.

Run a FlexUnit test within Flash Builder

You can run FlexUnit tests within Flash Builder on a project level or for individual test cases.

You typically run FlexUnit tests from the context menu for a project or from the context menus for an individual test case.

However, you can also run FlexUnit tests from the Flash Builder Run menu, the Flash Builder Run button, or from the Execute FlexUnit Test button in the FlexUnit Results view.

If you run tests from the Flash Builder Run menu, a Test Configuration dialog opens, allowing you to select which test classes and methods to run. Test cases for library projects cannot be run using the Flash Builder Run menu.

Flash Builder provides the following keyboard shortcuts to quickly launch FlexUnit tests:

- `Alt+Shift+A, F`

  Runs all FlexUnit tests in the project.
- `Alt+Shift+E, F`

  Runs the selected FlexUnit test.

Run FlexUnit tests from the current selection in the editor. (See “Configuring FlexUnit tests” on page 78.

1 Select a project and run the test:

   From the context menu for a project, select Execute FlexUnit Tests.

   From the Flash Builder Run menu or Run button, select Run > FlexUnit Tests.

2 (Flash Builder Run menu) In the Run FlexUnit Tests configuration dialog, select the test cases and methods to run in the test. Click OK to run the tests.

3 View the test results.

   Flash Builder generates a SWF file in the bin-debug folder of the project.

   An application opens displaying information about the test and indicates when the test is complete.

   The FlexUnit Results View panel opens displaying the results of the test. See “Viewing results of a FlexUnit test run” on page 79.

Run individual FlexUnit tests:

1 In the Project Explorer, navigate to the `flexUnitTest` package:

   From the context menu for a FlexUnit test file, select Execute FlexUnit Tests.

2 View the test results.

   Flash Builder generates a SWF file in the bin-debug folder of the project.

   An application opens displaying information about the test and indicates when the test is complete.

   The FlexUnit Results View panel opens displaying the results of the test. See “Viewing results of a FlexUnit test run” on page 79.

Run a FlexUnit test outside the Flash Builder environment

This procedure assumes that you have previously run a FlexUnit test within Flash Builder and that Flash Builder is running.

1 Copy the generated SWF file for a test from the bin-debug folder in your project to a folder outside your development environment.

   You can copy the automatically generated SWF file or a SWF file from a FlexUnit test that you have previously saved and configured.

2 Run the copy of the SWF file.

   Flash Player opens displaying information about the test and indicates when the test is complete.

   The FlexUnit Results View panel opens in Flash Builder displaying the results of the test.

Configuring FlexUnit tests

1 Open the Run FlexUnit Tests configuration dialog:

   You can open the Run FlexUnit Tests configuration dialog from the Run menu or from the FlexUnit Results view.

   - Select a project. From the Flash Builder menu, select Run > Run > Execute FlexUnit Test.
   - From the FlexUnit Results view, select the Execute FlexUnit Tests button.

     If the FlexUnit Results view is not open, select Window > Other Views > Flash Builder > FlexUnit Results.

2 In the Test Configuration dialog, select the test cases and methods to save as a test configuration.

***Note:** The Test Configuration dialog is not available when you run a test from the context menu in the Package Explorer.*

3 (Optional) Select Load to import previous configurations saved as XML files.

4 Click Save.

Flash Builder writes an XML file and an MXML file in the `.FlexUnitSettings` folder of your project.

You can use the XML file in build scripts to execute the test.

You can generate a SWF file from the MXML file. This SWF file can be used for testing outside the Flash Builder environment. Typically, you copy the generated MXML file to the `src` folder in your project to generate the SWF file.

Viewing results of a FlexUnit test run

The FlexUnit Results view displays results of a FlexUnit test, detailing any failures. You can navigate through the results, filter the display, write the results to a file, and load the results from a file.

You can also rerun tests, cancel a running test, and clear the results from the view.

If the FlexUnit Results view is not open, select Window > Other Views > Flash Builder > FlexUnit Results.

Test Results Panel

This panel lists all tests within the test run, indicating whether the test passed or failed.

Double-click a test in the list to go to the test in the ActionScript editor.

Test Failure Details Panel

Select a test in the Test Results panel to display failure details.

Each failure detail lists the source file and method, including the line number of the failure.

If the source file is local to the working space, double-click the listing to go to the failure in the ActionScript editor.

FlexUnit Results view menu

From the FlexUnit Results view menu, you can do the following:

- Filter the results displayed

  Hide the Test Failure Details panel.

  Display only test runs that failed.

- Scroll through the test runs displayed in the Test Results panel.
- Cancel a running test.
- Save the results of a test run or the configuration of a test run.
- Load results previously saved to a file.
- Clear the results from the panel.
- Rerun the current test. You can choose from:
  - Run all tests.
  - Run failures only.
  - Run the selected test.

- Refresh the FlexUnit configuration.

  If you modify a test or add or remove tests, click refresh to load the new FlexUnit configuration.

- Configure and run FlexUnit tests.

  Use the Execute FlexUnit Tests button to configure and run FlexUnit tests.

Chapter 3: Working with data and services in Flex

Flash Builder provides tooling support for building applications that access the following types of remote services:

- PHP
- ColdFusion
- BlazeDS
- LiveCycle Data Services
- HTTP services
- Web services
- J2EE
- ASP.NET

Workflow for accessing data services

- Connect to remote data
- Build the client application
- Manage the access of data from the server
- Monitor and debug the application

The workflow for accessing data services is not a sequential process. For example, many developers start with building a client application before creating or configuring access to the server. Other developers start with creating and configuring services, including managing the access of the data.

Connecting to remote data

Connecting to remote data requires a Flex server project. From the Flex server project you either create a new service on a remote server or connect to an existing remote service. You then configure custom data types for data retrieved from the data service.

Creating a server project

Create a Flex project that corresponds to the data service you are accessing. ColdFusion and PHP server projects support AMF-based services for access to data on the server. Additionally, all projects support connecting to existing HTTP and web services.

Note: *A cross-domain policy file is necessary when accessing services that are on a different domain from the SWF for the Flex application. Typically, AMF-based services do not need a cross-domain policy file because these services are on the same domain as the application. For more information, see Using cross-domain policy files.*

You can create the following types of Flex server projects:

Server Type	Services supported
PHP	• AMF-based PHP services • Web services • HTTP services
ColdFusion	• ColdFusion Flash Remoting • BlazeDS • LiveCycle Data Services • Web services • HTTP services
J2EE	• Web services • HTTP services
ASP.net	
No server specified	• Web services • HTTP services

For more information about creating Flex projects, see Working With Projects in the Flex help.

Connecting to data services

- "Working with ColdFusion services" on page 82
- "Working with PHP services" on page 83
- "Working with HTTP services" on page 84
- "Working with web services" on page 85

Working with ColdFusion services

Before connecting to a ColdFusion data service, create a server project for ColdFusion and make that the active project. When creating the project, select a remote access object service.

For ColdFusion, you have two options:

- LiveCycle Data Services
- ColdFusion Flash Remoting

You can only specify one ColdFusion remote access object service in a project. The remote service must be available under the web root you specified when creating the project. All applications in the project have access to the remote service.

Connecting to ColdFusion data services

1 Make sure you have a Flex server project for ColdFusion and that it is the active project.

   Note: *To make a project active, open an application file in the project and make it the active file in the editor.*

2 From the Flash Builder Data menu, select Connect to Data/Service.

3 In the Select Service Type dialog, select ColdFusion. Click Next.

4 In the Configure ColdFusion Service dialog, specify a name for the service and indicate whether to import an existing CFC or create a new one, as follows:

- Import existing CFC: Navigate to the location of the service.
- New CFC: Specify the default location for the service, or browse to a valid location.

5 Click Finish.

Flash Builder does the following:

- If you are creating a new CFC, Flash Builder generates a CFC file that contains stubs for common service operations.
- Flash Builder opens the CFC file in the source editor.
- The operations for the service are now available in the Data/Services view.

6 Modify the implementation of your service.

If you are creating a new service, the generated stubs contain examples of code to access a database.

Modify or replace the code in your service as necessary. You can modify parameters to operations and add additional functions.

***Note:** You can edit the file in Flash Builder or the editor of your choice.*

7 (Optional) If you modified an operation signature, added an operation, or removed an operation, refresh the service by selecting Refresh from the context menu for the service.

After implementing the service, see "Configuring return types for a data service" on page 85.

Working with PHP services

PHP data services are available using Action Message Format (AMF). For information on installing and configuring access to AMF services, see "Accessing services using Action Message Format (AMF)" on page 86.

Before connecting to a PHP data service you must first create a server project for PHP and make that the active project.

The data service must be available under the web root you specified when creating the project. All applications in the project have access to the remote service.

Connecting to PHP data services

1 Make sure you have a Flex Server project for PHP and that it is the active project.

***Note:** To make a project active, open an application file in the project and make it the active file in the editor.*

2 From the Flash Builder Data menu, select Connect to Data/Service.

3 In the Select Service Type dialog, select PHP. Click Next.

4 In the Configure PHP Service dialog, specify a name for the service and indicate whether to import an existing service or create a new one:

- Import existing service: Navigate to the location of the service.
- New service: Specify the default location for the service, or browse to a valid location.

5 Specify the services folder and gateway URL for your AMF location.

See "Accessing services using Action Message Format (AMF)" on page 86 for more information.

6 If this is the first PHP service accessed on your web server, you are asked to install a PHP Introspection Service on your server. Click OK.

7 Click Finish.

Flash Builder does the following:

- If you are creating a new service, Flash Builder generates a service file that contains stubs for common service operations.
- Flash Builder opens the service file in the source editor.
- The operations for the service are now available in the Data/Services view.

8 Modify the implementation of your service.

If you are creating a new service, the generated stubs contain examples of code to access a database.

Modify or replace the code in your service as necessary. You can modify parameters to operations and add additional functions.

***Note:** You can edit the file in Flash Builder or the editor of your choice.*

9 (Optional) If you modified an operation signature, added an operation, or removed an operation, refresh the service by selecting Refresh from the context menu for the service.

After implementing the service, see "Configuring return types for a data service" on page 85.

Working with HTTP services

You can create an application that connects to an HTTP service through a Flex server project or through a Flex project that does not specify a server technology. However, a cross-domain policy file is necessary when accessing services that are on a different domain from the SWF for the Flex application. For more information, see Using cross-domain policy files.

Connecting to HTTP data services

1 Select a Flex project for the application that accesses HTTP services.

2 From the Flash Builder Data menu, select Connect to Data/Service.

3 In the Select Service Type dialog, select HTTP. Click Next.

4 Specify a name for the service:

5 Click Add and Delete to add or remove operations for the service:

By default, a Get operation is already defined.

6 Configure the operations:

- Specify the operation name.
- Click the method to select Get or Post.
- For Post operations, select the content type.
- For Post operations, specify any parameters for the operation.
- Specify a URL for the HTTP service.

  If the URL for a Get operation contains parameters, then the Parameters table lists the name and data type of the parameters.
- Specify a package name for the service.

7 Click Finish.

Flash Builder adds the HTTP service to the Data/Services view.

Expand the service to view the available operations.

After implementing the service, see “Configuring return types for a data service” on page 85.

Working with web services

Placeholder for documentation of web services.

Configuring return types for a data service

After connecting to a data service, configure the data types returned by calls to the remote service.

Typically, you define custom types for records returned from the database. For example, if you are retrieving records from an employee database, then you would define the return type as Employee.

Configuring the type for data returned from a remote service

To configure the return type, call to the data service from an operation in the Data/Services view, sample the result, and then define or specify data types as necessary.

1 In the Data/Services view, from the context menu for an operation, select Configure Return Type.

2 In the Get Sample Data dialog, specify the type and value for any arguments to the operation:

  a If the type of the argument is not correct, click the argument type and select a new type from the dropdown list.

  b Specify values for the arguments.

  Type directly in the Value field or click the Ellipsis button to open the Input Argument dialog.

  c Click Invoke Operation.

3 Examine the returned data to see if it is correct.

If the data returned is not what you expect, or you encounter errors, click Cancel. Modify the implementation of the service. If you modified the signature or return type, refresh the service. Then try again.

Once the returned data is correct, click Next.

***Note:** If you need to troubleshoot the implementation of the service, it is often easier to write test scripts that access the service from a web browser and analyze those results. For information on test scripts, see “Debugging applications for remote services” on page 91.*

4 Specify or define a return type for the operation and click Next.

You can select a data type from the dropdown list, or define a custom data type for the returned data.

For example, if you are returning a list of employees with associated data, then you might define the return type as Employee.

5 View and edit the properties of the returned data and click Finish.

After creating or importing the service,

Accessing services using Action Message Format (AMF)

To access data services implemented with PHP, you must install and configure an implementation of AMF services, such as AMFPHP or WebORB for PHP. When creating access to a data service, you specify the location of the services folder and the AMF gateway URL.

This release of Flash Builder is compatible with the following open source software for accessing remote PHP services:

- AMFPHP
- WebORB for PHP

***Note:** ColdFusion 8 has built-in support for AMF. When creating a ColdFusion server project, specify ColdFusion Flash Remoting for the remote object access service.*

Installing and configuring AMFPHP

The free, open source software, AMFPHP, uses AMF to access remote services. AMFPHP is available for download at various locations, including sourceforge.net.

1 Download the AMFPHP installation file, which is available in ZIP format.

2 Extract the contents of the downloaded file to a temporary location.

   The extracted contents has the following structure:

```
+ amfphp-version
    + amfphp
        - gateway.php
        + browser
        + core
        + services
```

3 Move the amfphp folder to the web root of your HTTP server.

```
+ <webroot>
    + amfphp
        - gateway.php
        + browser
        + core
        + services
```

4 Use the following information to configure access to AMFPHP remote services:

   - services folder

     The PHP code implementing remote services reside in this folder. When configuring access to data services, you specify the file path to the services folder.

   - gateway URL

     The gateway URL is the URL to the `gateway.php` file. For example, if your web root is http://localhost, the gateway URL will be:

     ```
     http://localhost/amfphp/gateway.php
     ```

Installing and configuring WebORB for PHP

WebORB for PHP is open source software that uses AMF to access remote services. WebORB for PHP is available for download from Midnight Coders.

1 Download the WebORB for PHP installation file, which is available in ZIP format.

2 In the web root of your HTTP server, create a directory called weborb.

3 Extract the contents of the downloaded file to the weborb directory.

The WebORB for PHP installation has the following structure:

```
+ <webroot>
    + weborb
        + services
        - weborb.php
        . . .
```

4 Note the following information for configuring access to AMFPHP remote services:

- `services` folder

  The PHP code implementing remote services reside in this folder. When configuring access to data services, you specify the file path to the services folder.

- gateway URL

  The gateway URL is the URL to the `weborb.php` file. For example, if your web root is http://localhost, the gateway URL will be:

  ```
  http://localhost/weborb/weborb.php
  ```

Building the client application

When building a client application you can either build it directly from the Design View of the Code Editor, or you can generate a Form from the operations in the Data/Services view.

After laying out the components for the application, you generate event handlers to handle user interaction with the application.

Using Design View to build an application

Build the user interface of your application using Design View of the code editor. Drag and drop components into the Design Area, arrange the components, and configure their properties. After designing the application layout, you bind data returned from the data service to the components.

Typically you use the following Data Controls in your application, but you can choose from any components available. Flash Builder assists you in binding data from the data service to the following Data Control components:

Data Controls
DataGrid
AdvancedDataGrid
List
HorizontalList
TileList
Tree

You also add Controls such as Button, Combo Box, and Text to handle user input.

Building a client application and binding data from the data service

This procedure assumes you have connected to a data service and configured return types for the data.

1 In Design View, drag one or more Data Controls from Components View to the Design Area.

   Typically, you arrange the Controls within a Layout component such as a Container or Panel.

2 From the Data/Service view, drag an operation onto a component in the Design Area.

   Flash Builder configures the component according to the type of data returned from the operation.

   Modify the component as required for your application. For example, add or remove fields and change the headings to more meaningful names.

3 Add control items for user input, such as Combo Box, Buttons or Text input fields.

4 Configure event handlers for the application.

   Flex Builder automatically generates event handlers when you bind an operation to a component. However, you must provide event handlers for user input components such as Buttons or Text. You can also modify any generated event handler.

   For more information, see “Generating event handlers” on page 88.

Generating a Form for an application

Flash Builder can generate a Form based on the type of data accessed by a database operation. After generating the Form, you can add or modify components in the Form.

Generating a Form from a database operation

This procedure assumes you have connected to a data service and configured return types for the data.

1 In Data/Services view, select an operation and from the context menu, select Generate Form.

2 For the Generate Form For field, select the type of form to generate:

Data Type	Based on the attributes in a data type from the service. Typically used to add or remove records from a database.
New Service Call	Form is based on the return value of a selected database operation. Typically used to update a single record in a database.
Existing Call Result	Form uses a result from a database operation to generate a “Master/Detail” view.

3 Depending on your selection, specify additional information to generate for the Form.

   For example, specify the service, operation, and additional fields to generate.

4 Click OK.

   In the generated Form, fields are bound to data returned from the data service. Modify the form as necessary for your application.

Generating event handlers

When you bind a data service operation to a component, Flash Builder generates an event handler that retrieves data from the service to populate the component.

For example, if you bind an operation such as getAllItems() to a DataGrid, Flash Builder generates a creationComplete event handler.

```
<DataGrid creatonComplete="getAllItemsResult.token = productService.getAllItems()" ... >
```

When you run the application, once the DataGrid has been created it populates itself with data retrieved from the service.

You can accept the generated event handlers or replace them with others according to your needs. For example, you might want to replace the creationComplete event handler on the DataGrid with a creationComplete handler on the Application.

You can also generate or create event handlers for Controls that accept user input, such as Buttons or Text. Do either of the following to generate event handlers:

- From the Data/Services view, drag an operation onto the Control.

  Flash Builder generates an event handler for the default event for the component. For example, for a Button, this would be the Click event.

- In Design View, select the Control and then in the Property Inspector, click the generate event icon.

  Flash Builder opens the Source View of the editor and generates a stub for the event handler.

  Fill in the remaining code for the event handler. Flash Builder provides Content Assist to help you code the event handler. For more information see “Coding access to data services” on page 91.

Managing the access of data from the server

Paging Paging refers to incrementally retrieving large data sets from a remote service.

For example, suppose you want to access a database that has 10,000 records and then display the data in a DataGrid that has 20 rows. You can implement a paging operation to fetch the rows in 20 set increments. When the user requests additional data (scrolling in the DataGrid), the next page of records is fetched and displayed.

Data management In Flash Builder, data management refers to the synchronization of data between the client application and a data service. For example, suppose you have an application that accesses a database of employees. If your application modifies information for an employee, you need to synchronize these changes with the data service.

Data synchronization involves coordinating several operations (create, get, update, delete) to respond to events from the client application, such as updating an Employee record.

Enabling paging

To enable paging your data service must implement an operation with the following signature:

```
getItems_paged(startIndex:Number, numItems:Number): myDataType
```

operation name	You can use any legal name for the operation.
startIndex	The initial row of data to retrieve. The data type for startIndex should be defined as Number in the client operation.
numItems	The number of rows of data to retrieve in each page. The data type for numItems should be defined as Number in the client operation.
myDataType	The data type returned by the data service.

You can optionally implement a count() operation that returns the number of items returned from the service. Use the value returned from the count() operation to retrieve all the data from the data service. The first set of data is displayed while the remaining data is being fetched.

If you do not implement a count() operation, additional data is not fetched unless the user requests the data.

Enable paging for an operation

This procedure assumes you have coded both getItems_paged and count() operations in your remote service. It also assumes you have configured the return data type for the operation, as explained in "Configuring return types for a data service" on page 85.

1 In the Data/Services view, from the context menu for the getItems_paged operation, select Enable Paging.

2 If you have not previously identified a unique key for your data type, in the dialog that opens, specify the attributes that uniquely identify an instance of this data type and click Next.

   Typically, this is the primary key attribute.

3 Specify the count() operation. Click Finish.

Paging is now enabled for that operation.

Enabling data management

To enable data management implement one or more of the following operations, which are used to synchronize data on the remote server:

- Add (createItem)
- Get All Properties (getItem)
- Update (updateItem)
- Delete (deleteItem)

These operations must have the following signatures:

```
createItem(item:myDatatype):int
deleteItem(itemID:Number):void
updateItem((item: myDatatype):void
getItem(itemID:Number): myDatatype
```

operation name	You can use any legal name for the operation.

item	An item of the data type returned by the data service.
itemID	A unique identifier for the item, usually the primary key in the database.
myDataType	The data type of the item available from the data service.

Enable data management for an operation

This procedure assumes you have implemented the required operations in your remote service. It also assumes you have configured the return data type for the operations that use a custom data type, as explained in "Configuring return types for a data service" on page 85.

1 In the Data/Services view, expand the Data Types node.

2 From the context menu for a data type, select Enable Data Management.

3 If you have not previously identified a unique key for your data type, in the dialog that opens, specify the attributes that uniquely identify an instance of this data type and click Next.

   Typically, this is the primary key attribute.

4 Specify the Add, Get All Properties, Update, and Delete operations. Click Finish.

Data management is now enabled for that operation.

Coding access to data services

[*Placeholder for section to explain how to code data services.*]

Topics for this section:

- Service code generated by Flash Builder for the client application
- <Declarations> tag
- Accessing services using the Async token
- Using event handlers with service calls
- Difference between server-side typing and client-side typing
- Binding service operations to UI components

Debugging applications for remote services

There are several ways to debug applications that access remote services. This release features the Network Monitor, which you can use to view data sent between the server and client. You can also write scripts to test server code, and write output stream information to log files.

Network Monitor

The Network Monitor is available in Flash Builder from the Flex Debugging Perspective. The monitor must be enabled before it can be used to monitor data. Refer to Network Monitorfor details about enabling and using the Network Monitor.

Scripts to test server code

Use test scripts to view and debug server code before attempting to access data from the server using AMF. Test scripts provide the following benefits:

- You can view test results from a web browser.

  As you make changes to the code, simply refresh the browser page to view the results.
- You can echo or print results to the output stream, which you cannot do directly from AMF.
- Error displays are nicely formatted and generally more complete than errors captured using AMF.

ColdFusion Scripts

Use the following script, `tester.cfm`, to dump a call to a function.

```
<!--- tester.cfm --->
<cfobject component="EmployeeService" name="o"/>
<cfdump var="#o.getAllItems()#">
```

In `tester2.cfm`, you specify the method and arguments to call in the URL.

```
<!--- tester2.cfm --->
<cfdump var="#url#">

<cfinvoke component="#url.cfc#" method="#url.method#" argumentCollection="#url#"
returnVariable="r">

<p>Result:

<cfif isDefined("r")>
    <cfdump var="#r#">
<cfelse>
    (no result)
</cfif>
```

For example, call the getItemId method in EmployeeService with the following URL:

```
http://localhost/tester2.cfm?EmployeeService&method=getItemId&id=12
```

tester3.cfm writes a log that records calls to operations and dumps the input arguments using cfdump.

```
<!--- tester3.cfm --->
<cfsavecontent variable="d"><cfdump var="#arguments#"></cfsavecontent>

<cffile action="append"
file="#getDirectoryFromPath(getCurrentTemplatePath())#MyServiceLog.htm"
output="<p>#now()# operationName #d#">
```

PHP Scripts

Use the following script, `tester.php`, to dump a call to a function.

```
<pre>
<?php
include('MyService.php');
    $o = new MyService();
    var_dump($o->getAllItems());
?>
</pre>
```

Add the following code to your PHP service to log messages during code execution.

```
$message = 'updateItem: '.$item["id"];
$log_file = '/Users/me/Desktop/myservice.log';
error_log(date('d/m/Y H:i:s').' '.$message.PHP_EOL, 3, $log_file);
```

Add the following code to your PHP service to enable dumping to a log file:

```
ob_start();
var_dump($item);
$result = ob_get_contents();
ob_end_clean();

$message = 'updateItem: '.$result;
$log_file = '/Users/me/Desktop/myservice.log';
error_log(date('d/m/Y H:i:s').' '.$message.PHP_EOL, 3, $log_file);
```

Data services in Flash Builder

Data services typically contain CRUD operations to create, retrieve, update, and delete remote data. You can add additional operations to customize your implementation.

Flash Builder data service tools use Action Message Format (AMF) to represent data. AMF associates data with attributes as name/value pairs. For data stored in a database, the attribute is the column name and the value is the specific value for that column. Data retrieved from web services or HTTP services similarly associate the data with attributes.

Defining data types for services

When you create a new PHP or ColdFusion data service in Flash Builder, Flash Builder generates code that implements the service. The generated server code contains stubs for the basic CRUD operations that you then modify to implement your service. You typically write queries that return data as query objects or arrays of objects.

The data types for the service are managed on the client (client-side typing). You use Flash Builder to analyze service operations by using the Configure Return Type option from the Data/Services view. Flex Builder samples the data defined by the service and defines a data type for returned data. You specify a name for the data type, which then is available for all operations in the service.

You can, however, write server code that defines data types on the server (server-side typing). Server-side typing provides a guarantee that the properties of the data are correctly defined and accessed. However, server-side code is generally more complex to write than client-side code. In this release, Flash Builder samples and configures data types even if the data type is defined in the server code.

The tutorial in this chapter provides examples of client-side typing.

Tutorial: Creating an application accessing remote data services

This tutorial walks you through the basic steps of creating an application that connects to remote data services. The data service for this tutorial is implemented in PHP.

The application lists the employees in an Employees database. Because the database contains thousands of records, the application uses paging to retrieve the records.

The application allows you to add, update, and delete employee records. All modifications to the database are applied when you click a Commit button. You can revert changes made before you commit them to the database. The application uses data management features to synchronize data on the server for these operations.

Installing the tutorial database

This tutorial uses the employees table from the Employees sample database, which is available as a free download from MySQL. Follow the installation instructions from MySQL to install the database.

After installing the database, modify the structure of the emp_no field, which is the primary key. For this pre-release of Gumbo, this field needs to be set to auto_increment.

***Note:** You can substitute a database of your choice for this tutorial. In the tutorial procedures, make sure you substitute the correct values for the database, table, and records.*

Next: "Connecting to PHP data services" on page 94

Connecting to PHP data services

This tutorial assumes you have an installation of PHP, MySQL, and a web server, and that you have also installed the database extensions for MySQL.

The tutorial also assumes you have set the primary key, emp_no, to the employees database to auto_increment.

Flash Builder accesses remote PHP services using Action Message Format (AMF). Before beginning this tutorial you must install and configure access to AMF services. See "Accessing services using Action Message Format (AMF)" on page 86.

Create a Flex server project for PHP

1. In Flash Builder, select New > Flex Project.
2. Specify "EmployeeServiceProject" for the project name and set the application server type to PHP. Click Next.
3. In the Server Location area, specify the following:
   - For Web Root, navigate to the root of your HTTP server.
   - For Root URL, enter the URL for your web server. Typically, this is http://localhost.
   - Click Validate Configuration.
4. Click Finish.

Create a new PHP service

1. From the Flash Builder menu, select Data > Connect to Data Service.
2. Select PHP and click Next.
3. In the configure PHP Service dialog, specify the following:
   - Service Name: EmployeeService
   - Create New PHP File, Use Default Location: keep the default selections.
   - Services Folder, Gateway/Endpoint URL: Navigate to the services folder of your AMF services installation and specify the gateway URL. See "Accessing services using Action Message Format (AMF)" on page 86 for more information.

4 Click Finish.

Flash Builder generates a new PHP service and opens the EmployeeService.php file in a text editor.

Access to the service operations is available from the Data/Services view.

The generated service code is available from the Flex Package Explorer, which is a node under the services folder in your Flex project.

Next: "Coding PHP Services" on page 95

Coding PHP Services

This section shows you how to modify the generated service stubs to create a PHP service.

Note: *Flash Builder provides a text editor for editing the generated service code. However, you can edit the PHP code in an editor of your choice.*

Code access to your database

❖ Uncomment the connect() function and modify it with values that provide access to the employees database:

```
$connection = mysql_connect("localhost", "admin", "password") or die(mysql_error());
mysql_select_db("employees", $connection) or die(mysql_error());
```

Code the getItems_paged() and count() functions

Because the Employees sample database is a large database, this tutorial implements paging to manage the download of large data sets.

1 Uncomment getItems_paged() and edit the generated code:

```
public function getItems_paged($startIndex, $numItems) {
     $this->connect();
     $startIndex = mysql_real_escape_string($startIndex);
     $numItems = mysql_real_escape_string($numItems);
     $sql = "SELECT * FROM employees LIMIT $startIndex, $numItems";
     $result = mysql_query($sql) or die('Query failed: ' . mysql_error());
     return $result;
}
```

2 Uncomment the count() function and make the following edits:

```
public function count() {
     $this->connect();
     $sql = "SELECT * FROM employees";
     $result = mysql_query($sql) or die('Query failed: ' . mysql_error());
     $rec_count = mysql_num_rows($result);
     mysql_free_result($result);
     return $rec_count;
}
```

Code the data management operations

To implement data management, code the createItem, updateItem, getItem, and deleteItem functions. The client application uses a combination of these operations to update server data.

1 Uncomment and edit createItem():

```
public function createItem($item) {
    $this->connect();
    $sql = "INSERT INTO employees (birth_date, first_name, last_name, gender, hire_date)
    VALUES ('$item[birth_date]','$item[first_name]',
            '$item[last_name]','$item[gender]','$item[hire_date]' )";
    $result = mysql_query($sql) or die('Query failed: ' . mysql_error());
    return mysql_insert_id();
}
```

2 Uncomment and edit updateItem():

```
public function updateItem($item) {
    $this->connect();

    $sql = "UPDATE employees SET birth_date = '$item[birth_date]',
            first_name = '$item[first_name]', last_name = '$item[last_name]',
            gender = '$item[gender]', hire_date = '$item[hire_date]'
            WHERE  emp_no = $item[emp_no]";

      $result = mysql_query($sql) or die('Query failed: ' . mysql_error());
}
```

3 Uncomment and edit getItem():

```
public function getItem($itemID) {
    $this->connect();
    $itemID = mysql_real_escape_string($itemID);
    $sql = "SELECT * FROM employees where emp_no=$itemID";

    $result = mysql_query($sql) or die('Query failed: ' . mysql_error());
    return $result;
}
```

4 Uncomment and edit deleteItem():

```
public function deleteItem($itemID) {
      $this->connect();
      $itemID = mysql_real_escape_string($itemID);
      $sql = "DELETE FROM employees WHERE emp_no = $itemID";
      $result = mysql_query($sql) or die('Query failed: ' . mysql_error());
}
```

Next: “Configuring return types” on page 96

Configuring return types

Each operation in the service retrieves data from the remote server. For each operation, configure the data type for the returned data.

Configure the return data type for getItems_paged()

1 In the Data/Services view, select the getItems_paged operation.

2 From the context menu for the getItems_paged operation, select Configure Return Type.

3 In the Get Sample Data dialog, specify the following for argument type and argument value:

Argument	Argument Type	Value
startIndex	Number	1
NumItems	Number	10

4 Click Invoke Operation.

The first 10 records are retrieved from the data service and are ready for display. Additional records are fetched when needed.

5 In the Return Type Name field, type Employee and click Finish.

Employee is a custom data type defining returned records of employee.

Configure the return data type for createItem()

1 In the Data/Services view, from the context menu for createItem, select Configure Return Type.

2 Specify the argument type and argument value:

Argument	Argument Type	Value

item	Employee	Select the Ellipses button and specify the following:
		`{`
		`gender: 'M',`
		`birth_date: 2000-01-01,`
		`last_name: "last",`
		`hire_date: 2000-01-01,`
		`first_name: "first"`
		`}`

3 Invoke the operation and note the returned value, which is the emp_id for the created item.

Use this value for emp_no when configuring getItem(), updateItem(), and deleteItem().

4 Click Finish.

Configure the return data type for getItem()

1 In the Data/Services view, from the context menu for createItem, select Configure Return Type.

2 Specify the argument type and argument value:

Argument	Argument Type	Value
itemID	Number	*emp_no*

3 Invoke the operation.

4 In the Return Type Name field, select Employee from the dropdown list and click Finish.

Configure the return data type for updateItem()

1 In the Data/Services view, from the context menu for updateItem, select Configure Return Type.

2 Specify the argument type and argument value:

Argument	Argument Type	Value

item	Employee	Select the Ellipses button and specify the following:
		`{`
		`emp_no: `*`emp_no`*`,`
		`gender: 'M',`
		`birth_date: 2000-01-01,`
		`last_name: "New Last",`
		`hire_date: 2000-01-01,`
		`first_name: "New First"`
		`}`

3 Invoke the operation, click OK, and click Finish.

Configure the return data type for deleteItem()

1 In the Data/Services view, from the context menu for deleteItem, select Configure Return Type.

2 Specify the argument type and argument value:

Argument	Argument Type	Value
itemID	Number	*emp_no*

3 Invoke the operation, click OK, and click Finish.

Next: “Manage data retrieved from the server” on page 98

Manage data retrieved from the server

The Employee database contains thousands of records. Implement paging to retrieve the data in sets. Also, the client application updates information on the server, so implement data management.

Enable paging

1 In the Data/Services view, select the getItems_paged operation.

2 From the context menu for the getItems_paged operation, select Enable Paging.

3 In the Select Unique Identifier dialog, select emp_no and click Next.

4 In the Confirm Paging dialog specify the count() operation from the dropdown list.

5 Click Finish.

Enable data management

1 In the Data/Services view, expand the Data Types node for EmployeeService and select the Employee data type.

2 From the context menu for the Employee data type, select Enable Data Management.

3 In the Select Unique Identifier dialog, emp_no has already been selected. Click Next.

4 In the Map Database Operations dialog, specify the following operations and click Finish.

- **Add (Create) Operation**: createItem(item: Employee)
- **Get All Properties Operation**: getItem(itemID: Number)
- **Update Operation**: updateItem(item: Employee)
- **Delete Operation**: deleteItem (itemID: Number)

Data management is now enabled for this operation. Flash Builder generates client code that can update data using a combination of the mapped operations.

Next: "Building the User Interface" on page 99

Building the User Interface

In Design View of the code editor, drag and drop components to create your user interface. This application uses the following components:

- An editable DataGrid to list employee records
- Buttons to delete records, commit changes to records and revert changes to records

Add the DataGrid and Buttons to the application

1 In Flash Builder, open the EmployeeServiceProject.mxml file, and then select Design to open Design View of the MXML editor.

2 From the Components view, drag a DataGrid component onto the Design Area and place it near the top.

   The DataGrid component is listed under Data Controls.

3 In the Flex Properties view, type "dg" for the ID property.

4 In the Flex Properties view, select the icon for Category View and edit the following property.

   - Expand Common, and for sortableColumns specify false.

     Because the database contains thousands of records, set sortable to false to avoid the overhead of sorting the records.

5 Drag four Buttons to the Design Area, lining them up beneath the DataGrid.

6 In Flex Properties view, select the icon for Standard View. For each Button, provide the following Labels and IDs:

Label	ID
Add	addButton
Delete	deleteButton
Revert	revertButton
Commit	commitButton

7 Save EmployeeServiceProject.mxml, then select Run > Run EmployeeServiceProject.

   After viewing the running application, close the application in the web browser.

Next: "Binding the UI components to data service operations" on page 99

Binding the UI components to data service operations

Configure the DataGrid to retrieve employee records, and add event handlers to the Buttons so the records can be updated or deleted.

Bind the getItems_paged() operation to the DataGrid

1 In the Data/Services view, select the getItems_paged() operation and drop it onto the DataGrid.

2 Click Configure Columns.

3 For each column, edit the headerText property to give the columns meaningful names.

Column	Value
emp_no	ID
gender	Gender
birth_date	Date of Birth
last_name	Last Name
hire_date	Hire Date
first_name	First Name

Generate event handlers for the Buttons

1 Select the Add button. Then, for the On Click field of the Add button, click the Pencil icon to generate an event handler for the button.

The MXML editor changes to Source View, placing the cursor in the generated event handler.

2 In the Script block, add the following import statement:

```
import services.employeeservice.Employee;
```

3 In the event handler body, type the following:

```
var e:Employee = new Employee();
e.first_name = "New";
e.last_name = "New";
e.birth_date = "2000-01-01";
e.hire_date = "2000-01-01";
e.gender = "M";
dg.dataProvider.addItem(e);
dg.verticalScrollPosition = dg.dataProvider.length -1;
```

As you type, Flash Builder content assist helps you view the available methods and values.

4 In Design View, add an On Click event handler for the Delete button and specify the following code:

```
employeeService.deleteItem(dg.selectedItem.emp_no);
```

5 Similarly, add an On Click event handler for the Revert button with the following code:

```
employeeService.getDataManager
    (employeeService.DATA_MANAGER_EMPLOYEE).revertChanges();
```

6 Add an On click event handler for the Commit button with the following code:

```
employeeService.commit();
```

Next: “Run the application” on page 100

Run the application

The application is complete. Save your changes and select Run > Run EmployeeServiceProject.

You can update and delete selected employees in the DataGrid. You can also add new employees.

Note: *When adding new employees, after selecting the Add button, scroll to the last item in the DataGrid to view the new entry. Because of the large data set, you have to scroll to the end to see the first new item that you add. After adding the first new item, the DataGrid scrolls to each newly added item.*

When adding dates, use this format: yyyy-mm-dd

Click the Revert button to undo any changes you made.

Click the Commit button to write all changes to the database.

Chapter 4: Migrating Flex 3 applications to Flex 4

When converting Flex 3 applications to Flex 4, you might encounter the following issues:

For additional information about the differences between Flex 3 and Flex 4, see Flex 4 Backwards Compatibility.

Namespaces

Flex 4 uses a new namespace. The old namespace (sometimes referred to as the "2006 namespace") is as follows:

```
xmlns:mx="http://www.adobe.com/2006/mxml"
```

The new namespace (sometimes referred to as the "2009 namespace") is composed of three namespace definitions, as follows:

```
xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:mx="library://ns.adobe.com/flex/mx"
xmlns:s="library://ns.adobe.com/flex/spark"
```

To use the new namespace, replace the old namespace with the new ones at the top of your MXML documents.

You can use all the Flex 3 components and language elements with the 2009 namespace.

You can still use the 2006 namespace and compile your applications with the Flex 4 compiler, but if you want to use the new features of Flex 4, such as layouts, effects, components, and FXG, convert your applications to the 2009 namespace. In addition, if you want to copy and paste sample code, you should remove prefixes for framework tags in your MXML files.

Different files used by the same application can have different namespaces (either 2006 or 2009), but you cannot have both namespaces in the same file.

For more information, see "Namespaces" on page 1.

Declarations

Non-default, non-visual properties must be wrapped in the `<fx:Declarations>` tag in Flex 4 applications. For example, in Flex 3 you could declare a String as a child of the Application tag without specifying that it was not to be added to the display list:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- migration/DeclarationsExampleFails.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:String id="str">The quick brown fox.</mx:String>
    <mx:Label text="{str}"/>
</mx:Application>
```

In Flex 4, you must wrap the String in a `<fx:Declarations>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- migration/DeclarationsExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <fx:Declarations>
        <fx:String id="str">The quick brown fox.</fx:String>
    </fx:Declarations>
    <s:Label text="{str}"/>
</s:Application>
```

Effects, validators, and formatters are other examples of non-visual items that should be placed inside a `<fx:Declarations>` tag. Note, however, that states and transitions continue to exist outside of the `<fx:Declarations>` tag.

Other classes that must be wrapped in a `<fx:Declarations>` tag include Array, XMLList, XML, and HTTPService.

Default properties for custom components

Custom components now support the use of a default property for the root of an MXML document tag. Previously, you had to explicitly define default properties in your MXML-based custom components. Now, the immediate child tags that are not values for properties declared on the base class will be interpreted as values for the default property.

This might change the way your MXML-based custom components are interpreted by the compiler.

In the following example, if the default property takes an array, then the two strings are added to the default property's array. They are not declared as new properties of the MyComponent sub-class, as they would have been under Flex 3.

```
<MyComponent xmlns="http://ns.adobe.com/mxml/2009">
    <String>A value</String>
    <String>Another value</String>
</MyComponent>
```

Loading SWF files

The signature of the `ModuleLoader.loadModule()` method has changed from:

```
public function loadModule():void
```

To:

```
public function loadModule(url:String, bytes:ByteArray):void
```

The behavior of the method should not change, though, because the default values of the new arguments is `null`.

Also, the SWFLoader class has a new property, `loadForCompatibility`, and a new method, `unloadAndStop()`.

These changes were added in Flex 3.2, but if you are migrating an application from Flex 3 to Flex 4, you might encounter them.

More Help topics

Developing and Loading Sub-applications

Charting

The `direction` style property of the GridLines class has been renamed to `gridDirection`.

In addition, when you use any non-display object to define chart data or to define the appearance of a chart, be sure to move that into an `<fx:Declarations>` block. This also includes HTTPService and other remote object tags that get chart data.

States

View states let you vary the content and appearance of a component or application, typically in response to a user action. Flex 4 lets you specify view states using a new inline MXML syntax, rather than the syntax used in Flex 3. In the new syntax, the AddChild, RemoveChild, SetProperty, SetStyle, and SetEventHandler classes have been deprecated and replaced with MXML keywords.

For information about the new states model, see View states.

HTML wrapper

The HTML wrapper has changed significantly from Flex 3. For example, the generated files now include the swfobject.js file rather than the AC_OETags.js file. The SWF file embedding is based on the SWFObject project. For a complete description of the changes, see "HTML wrappers" on page 25.

Also, the default output of the HTML templates is to center the SWF file on the page. In Flex 3, the default was to align the SWF file to the upper left corner of the page.

Layouts

If you use the Spark Application class instead of the MX Application as your MXML file's root tag, the default layout is absolute (BasicLayout). Previously, the default layout was vertical. To change the layout in the Spark Application class, use the `<s:layout>` child tag and specify one of the layout classes, BasicLayout, HorizontalLayout, or VerticalLayout.

The following example sets the Spark Application class's layout to HorizontalLayout, which is the same as the default setting for the Application class in a Flex 3 application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- layouts/HorizontalLayoutExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark">
     <s:layout>
          <!-- HorizontalLayout positions controls horizontally. -->
          <s:HorizontalLayout/>
     </s:layout>

     <s:Button label="Click Me"/>
     <s:Button label="Click Me"/>
</s:Application>
```

The constraint styles (`baseline`, `top`, `bottom`, `left`, `right`, `horizontalCenter`, and `verticalCenter`) have been duplicated as properties of the UIComponent class. While you can still set them as styles, you should now set them as properties.

The coordinate space of UIComponent layout-related properties have been changed from post-scale to pre-transform. When you change the scale on a UIComponent, there will be a slight difference in positioning and sizing. The difference is small, but might be noticed in text anti-aliasing. The difference might increase with the number of changes to the scale properties.

Watch for differences any time you set the scale on a UIComponent and you rely on one of these properties. For example, custom containers that arrange children might need to be updated to support scaled children.

The following table describes the coordinate space differences between Flex 3 and Flex 4:

Properties	**Flex 3 behavior**	**Flex 4 behavior**
`height` `width`	Coordinate space is post-scale in regards to the last committed scale on UIComponent.	Coordinate space is pre-transform (and pre-scale).

`explicitWidth`
`explicitHeight`
`explicitMinWidth`
`explicitMinHeight`
`explicitMaxWdith`
`explicitMaxHeight`
`measuredWidth`
`measuredHeight`
`measuredMinWidth`
`measuredMinHeight`
	Coordinate space is pre-scale during the `measure()` method, but post-scale after the `measure()` method is called.
	
	In Flex3, the coordinate space also depended on the scale. Changing the scale would change the value of the properties.
		Coordinate space for properties is always the same, doesn't change over time, and is not dependent on other properties. This change affects scaled objects. For example, if you wrote a custom MX container that supported scaled items, you might need to call the `Container.getLayoutItemAt()` method or use the PostScaleAdaptor class to compensate for the changes. Another cases where you might be affected is if you try to match the size of one display object to another.

You can prevent the Flex 4 compiler from applying the coordinate space differences that are described in this section. To do this, set the value of the `compatibility-version` compiler option to `3.0.0`, as the following example shows:

```
mxmlc ... -compatibility-version=3.0.0 ...
```

For more information, see About Spark layouts.

Binding

The binding expression `@{`*`ref_value`*`}` is recognized as a two-way binding expression in Flex 4.

In Flex 3, a binding expression like the following was interpreted as a one-way binding:

```
<mx:String id="str">test</mx:String>
<mx:Text text="@{str}"/>
```

Flex 3 set the Text component's `text` property to "test", but the value of the String component did not change if the Text component's `text` property changed.

In Flex 4, this example is interpreted as a two-way binding expression. A change in the String component updates the `text` property of the Text component, and a change in the `text` property of the Text component updates the String component.

You can prevent the Flex 4 compiler from enforcing the two-way binding. To do this, set the value of the `compatibility-version` compiler option to `3.0.0`, as the following example shows:

```
mxmlc ... -compatibility-version=3.0.0 ...
```

In addition, you can no longer bind the `name` property in a state. The behavior was changed to support the new states model in Flex 4, where the `name` property behaves like `id` and is used at compile time. As a result, the `name` property cannot be set at run time.

For more information, see Data binding.

Filters

Filter tags in MXML now use Flex filters rather than Flash filters. In Flex 3, when you used a filter tag in MXML, the compiler used a class in the flash.filters.* package. The Flash filters have the array properties `colors`, `ratios` and `alphas`.

In Flex 4, the same filter tag in MXML uses a class from the spark.filters.* package. As a result, you must follow the new syntax for these filter classes. For example, the spark.filters.GradientBevelFilter, spark.filters.GradientFilter, and spark.filters.GradientGlowFilter classes have an array property called `entries` that contain GradientEntry instances. The GradientEntry class defines the `color`, `ratio`, and `alpha` properties.

This change affects applications that use tags for the BevelFilter, BlurFilter, ColorMatrixFilter, ConvolutionFilter, DisplacementMapFilter, DropShadowFilter, GlowFilter, GradientBevelFilter, and GradientGlowFilter classes.

One of the features of these new filter classes is that you do not need to reinstate the `filters` property on a component when a property of one of these filters changes; the component automatically picks up that change. This is different from Flex 3, in which any changes to filter properties had to be followed by setting the `filters` property on the component again.

You must be sure to import classes from the flash.filters.* package that you use in your Spark filters. For example, if you use a Spark BevelFilter, but specify the BitmapFilterQuality.HIGH constant for it's quality property, you must import the BitmapFilterQuality class, as the following example shows:

```
import spark.filters.*;
import flash.filters.BitmapFilterType;
import flash.filters.BitmapFilterQuality;

myBevelFilter = new BevelFilter(5, 45, color, 0.8, 0x333333, 0.8, 5, 5, 1,
    BitmapFilterQuality.HIGH, BitmapFilterType.INNER, false);
```

Component rendering

When you compile your applications with the Flex 4 compiler, you might notice several changes to the appearances of the MX components.

The NumericStepper control is one pixel wider when you set its `fontFamily` style property to a specific font.

Rounded corners of some components are slightly different. The difference is very slight, but panels, buttons and other components with rounded corners are affected. This is a difference in Flash Player 10 from Flash Player 9.

Menus are shifted one or two pixels to the right. In Player 9, the background of a Menu bled over the left border by one to two pixels. In Player 10, the left border of a Menu has cleaner lines.

With embedded fonts, some letters in the DateField control appeared shifted. This is no longer true.

Styles

Changes to the Flex 4 CSS implementation require some migration when using type and class selectors in your CSS. In addition, Flex 4 emphasizes the use of styles in the skin classes associated with controls rather than with the controls themselves. As a result, some style properties that you could previously set on the control must now be set by subclassing the skin.

Namespaces

CSS in Flex 4 now requires a namespace for type selectors so that the compiler can disambiguate between classes of the same name in different packages. For example, there is a Button class in the mx.controls package and a Button class in the spark.components package.

The supported namespaces are as follows:

- `library://ns.adobe.com/flex/spark`
- `library://ns.adobe.com/flex/mx`

The "spark" namespace is for Spark components. The "mx" namespace is for MX components. You apply a namespace by providing a prefix and then specifying the prefix in the type selector, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- styles/NamespaceIdentifierExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:myComps="*">
     <s:layout>
        <s:VerticalLayout/>
     </s:layout>

     <fx:Style>
        @namespace s "library://ns.adobe.com/flex/spark";
        @namespace mx "library://ns.adobe.com/flex/mx";

        s|Button {
            fontSize:16;
        }

        mx|VBox {
            color:red;
        }
     </fx:Style>

    <mx:VBox>
        <!-- This Spark button is red, and has a fontSize of 16. -->
        <s:Button label="Click Me, Too"/>
    </mx:VBox>
</s:Application>
```

If you do not specify a namespace for an ambiguous type selector, the compiler returns a warning.

Class selectors

In Flex 4, CSS selectors have more functionality, and therefor the processing of those selectors now handles some situations differently. In Flex 3, class selectors that used different subjects but the same `styleName` were applied to all components whose `styleName` matched, in a last-in basis. Now, Flex processes the entire subject, including the class, so that components match on a `styleName` and any other portions of the selector predicate.

In Flex 3, the following class selectors would result in a red label for the Button control and red text for the Text control. The last definition was used by all components whose `styleName` matched "myStyleName".

```
Button.myStyleName { color:green; }
Text.myStyleName { color:red; }
```

In Flex 4, the label of the Button control is green and the text in the Text control is red.

For more information about advanced CSS in Flex 4, see "Advanced CSS" on page 19.

Style properties

MX components supported a large number of styles that are no longer supported in Spark components. In MX, skinning was done as a last resort. In Spark, skinning is a primary workflow. MX components relied on styles to handle items that really belong in the skin.

The following list describes some of the differences:

- Layout-related styles: Examples include `labelPlacement`, `headerHeight`, and `verticalGap`. These styles are now properties in the skin and/or layout.
- State-driven styles: Examples include `disabledColor`, `backgroundDisabledColor`, and `textRollOverColor`. These styles are replaced with stateful styles/attributes in the skin or CSS pseudo selectors.
- Drawing-related styles: Examples include `fillColors`, `highlightAlphas`, and `cornerRadius`. These styles are replaced with FXG markup in the skins.

Theme specific styles

Some component styles are now applicable only if you are using the correct theme. For example, the `borderAlpha` style of the mx.containers.Panel container is only applicable when using the Halo theme. The `symbolColor` style of the spark.compronents.CheckBox control is only applicable when using the Spark theme.

If a style is only applicable for a certain theme, that theme is listed in the style description in the ActionScript 3.0 Reference for the Adobe Flash Platform.

StyleManager

To access the StyleManager, you use the `styleManager` property of the current application. This is because each application can have its own instance of the StyleManager, rather than sharing the same one across multiple applications.

In addition, when calling the `styleManager.getStyleDeclaration()` method, you must now specify the fully-qualified package name of the target class, rather than just the class name.

The following example shows these differences:

```
<?xml version="1.0"?>
<!-- migration/StyleManagerExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">

    <fx:Script><![CDATA[
        public function initApp():void {
            /* New method of accessing the StyleManager. */

styleManager.getStyleDeclaration("spark.components.Button").setStyle("fontSize",15);

            /* Old method; no longer works. */
            StyleManager.getStyleDeclaration("Button").setStyle(setStyle("fontSize",15);
        }
    ]]></fx:Script>

    <s:Button id="myButton" label="Click Me"/>

</s:Application>
```

Fonts

How you embed fonts depends on which set of components you are using. Using embedded fonts with Flex 4 components do not require additional coding. Using embedded fonts in an application where Flex 4 and Flex 3 components are used might require you to either change the `textFieldClass` or `textInputClass` styles of the Flex 3 component or embed the font for each type of component.

For more information, see Embedding Fonts with MX components.

In addition, Flex no longer supports the `local()` method of embedding a font by name. You must now use the `src()` method in your `@font-face` rules, as the following example shows:

```
 @font-face {
    src:url("../assets/MyriadWebPro.ttf");
    /* You can no longer do this: */
    /* src:local("Myriad Web Pro"); */
    fontFamily: myFontFamily;
    embedAsCFF: true;
}
```

Globals

Setting and getting properties on the top-level Application object is different in Flex 4.

ApplicationGlobals.application has been replaced with FlexGlobals.topLevelApplication. Using Application.application is deprecated. FlexGlobals.topLevelApplication is the recommend alternative. Application.application will return null if the top-level application is not of type Application or a subclass of Application.

To set properties, such as styles or event listeners, on the Application class, import the FlexGlobals class, and use the `topLevelApplication` property, as the following example shows:

```
import mx.core.FlexGlobals;
FlexGlobals.topLevelApplication.addEventListener( KeyboardEvent.KEY_UP, keyHandler);
```

Graphics

The Stroke class has been deprecated. Use SolidColorStroke instead. For example, change this:

```
var s:Stroke = new Stroke(0x001100,2);
```

to this:

```
var s:SolidColorStroke = new SolidColorStroke(0x001100,2);
```

The LinearGradient class's angle property has been renamed to rotation.

Ant tasks

The templates attribute has been removed from the html-wrapper ant task. The following two new attributes have been added:

- `express-install` (default false): When set to true playerProductInstall.swf will be copied with the html file.
- `version-detection` (default true)

The `express-install` attribute takes precedence over the version-detection attribute. If `express-install` is true, version-detection will also be treated as true.

For more information, see Using the html-wrapper task.

Repeater

When using the Repeater control in Flex 4 applications, you must do the following:

- Wrap the Repeater control in a MX container such as VBox, HBox, Panel, or Canvas. You cannot wrap the Repeater control in a Spark container.
- If you use non visual classes such as Array, ArrayCollection, or Model to define repeated elements, put those inside a Declarations tag.
- Avoid putting Spark text primitives Label and RichText inside a repeater; these controls require a container and are not based on UIComponent.

RSLs

The framework RSL is linked by default. This means that when you build an application SWF file, classes in the framework.swc file are not included in the application SWF file. The classes are instead loaded from the framework RSL before the main application loads. In Flex 3 the default was to statically link the classes. Using the framework RSL was optional.

The advantage of using the framework RSL is smaller application SWF sizes and faster download times. The downside is increased memory usage because all the framework classes are loaded in the RSL, not just the classes that your application requires.

For more information, see Using the Framework RSLs.

Deferred Instantiation

Flex 4 containers do not support the `queued` creation policy. As a result, you cannot add Flex 4 containers or their contents to the creation queue.

In addition, Spark containers use the `createDeferredContent()` method to instantiate components that have been deferred. MX containers continue to use the `createChildren()` or `createComponentFromDescriptors()` methods.

Another difference between Flex 3 and Flex 4 is that the `creationPolicy` property is now an *inheritable* property. If you do not explicitly set it on a container, the value will be inherited from the parent container.

For more information, see “Deferred instantiation” on page 27.

Drag and drop

As part of a drag-and-drop operation, the drag initiator writes the drag data to an instance of the DragSource class. In previous releases of Flex, the list-based controls, except for Tree, wrote that data with a format String of `"items"`. In Flex 4, the format string is `"itemsByIndex"`.

For the Tree control, the format string of the drag data is `"treeItems"`, as it was in previous releases of Flex.

In previous releases of Flex, the drag data was written to the DragSource object as an Array. In Flex 4, the drag data is written as a Vector of type Object.

For more information, see Drag and drop.

AdvancedDataGrid

In the previous version of Flex, you used the SummaryField class to create summary data for an AdvancedDataGrid control. You now use the SummaryField2. The SummaryField2 class is new for Flex 4 and provides better performance than SummaryField.

In the previous version of Flex, you used the GroupingCollection class with the AdvancedDataGrid control. You now use the GroupingCollection2 class. The GroupingCollection2 class provides better performance than GroupingCollection.

For more information, see AdvancedDataGrid control.

Versioning

Flex 4 applications require that Flash Player be version 10.0 or later. As a result, the `target-player` compiler argument only accepts the value 10.0.

The `compatibility-version` compiler argument accepts the values 3.0 and 4.0.

Modules

When loading a module with the `load()` method, you should pass the main application's module factor. This idenfities the parent's StyleManager to the module.

For example:

```
 info = ModuleManager.getModule("movies/ColumnChartModule.swf");
info.addEventListener(ModuleEvent.READY, modEventHandler);
info.load(null, null, null, moduleFactory);
```

For more information about modules, see Modular Applications.

Sub-applications

When using sub-applications, you must ensure that your main application was compiled with the same or later version of the compiler than the sub-applications that it loads.

In addition, when compiling applications that will be loaded by other applications, you should specify the following compiler argument:

```
includes=mx.managers.systemClasses.MarshallingSupport
```

This includes all sub-applications and the main applications that load them.

For more information about sub-applications, see Developing and Loading Sub-applications.