Flex 4 in a day

About This Document

While it is impossible to cover all the aspects of any nontrivial framework or language in only 40 pages, we are confident that after reading this document you'll manage to find your way on Flex 4 projects. On the web there are many interesting articles on Flex and there is extensive documentation on the Adobe website. This document is not intended to duplicate that content. Instead, we have strived to keep it focused on what you need to know, so you have enough information to get started and you know where to look for more.

About the authors:

Mihai Corlan is a platform evangelist with Adobe Systems. For the past two years he's been traveling and speaking at conferences or partners and clients around the world. He focuses on Flex, AIR, and Flex Builder. You can follow him on his blog http://corlan.org or twitter http://twitter.com/mcorlan.

Alin Achim is a computer scientist with Adobe Systems. He is part of a team charged with building software used by Adobe employees. Flex (combined with various server side technologies) forms the backbone for the solutions they build.

Introduction

If you remember the transition from Flex 2 to Flex 3, you may recall that it was fairly nice and easy to follow. Compared to that, Flex 4 is a revolution: some aspects of the framework were changed and many new features were added.

The most notable new features are:
- A new UI component architecture named Spark that works along with the old one (MX). The Spark architecture separates the skinning and layout from the data and behavior.
- A new animation engine
- The addition of FXG and MXML graphics
- New MXML language features
- The addition of text controls based on the Text Layout Framework
- Charts, advanced data grid, and OLAP data grid are now part of the open source Flex SDK

MXML 2009

To enable the changes to the MXML language a new namespace was created. This chapter covers the new tags in MXML 2009 and the different namespaces available.

Namespaces

The first big change in Flex 4 is the separation of the namespace for the MXML language from the one for the components library. Thus, in your applications you now specify the namespace for the MXML language (2009 or 2006). You also specify the namespace for the Spark components library and MX components library if you want to use them. This separation is needed because there are components with the same name in both libraries (for example Button, Label, and so on).

You can still use the 2006 namespace, but you cannot use Flex 4 features (including the new components, new states engine, or layout schemes) with it. You also cannot mix the language namespaces in the same document. You can however, have some documents that use MXML 2009 and other documents that use MXML 2006.

Here are the main namespaces for Flex 4:

- MXML language declaration:
  ```
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  ```
- MX components (includes all of the components in the Flex mx.* packages, the Flex charting components, and the Flex data visualization components):
  ```
  xmlns:mx="library://ns.adobe.com/flex/mx"
  ```
- Spark components (includes all of the components in the Flex spark.* packages and the text framework classes in the flashx.* packages):
  ```
  xmlns:s="library://ns.adobe.com/flex/spark"
  ```
- Old Flex 3 and MX components library (when you don't want to use the new language features or the Spark components):
  ```
  xmlns:mx="http://www.adobe.com/2006/mxml"
  ```

For example:
```
<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:Style></fx:Style>

        <s:Button label="This is a Spark button!"/>
        <mx:Button label="This is a MX button!"
</s:Application>
```

If you work with Flash Catalyst and Illustrator or Photoshop to create skins for Flex applications you might see other namespaces as well; for example:

```
xmlns:ai="http://ns.adobe.com/ai/2009"
xmlns:d="http://ns.adobe.com/fxg/2008/dt"
xmlns:flm="http://ns.adobe.com/flame/2008"
```

These namespaces are either related to the FXG or used internally by Flash Catalyst, Flash Builder, Illustrator, or Photoshop.

States

One of the bigger enhancements in Flex 4 was made to simplify how states are managed. If you want to use states in a component or MXML document you use a `<s:States>` tag as a direct child of the root tag to declare any number of states, and then you can use the attributes `includeIn` and `excludeFrom` on any component to specify whether the component is present in a given state. You change the state using the property `currentState`. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<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="report"/>
            <s:State name="form"/>
    </s:states>

        <s:CheckBox label="Checkbox" includeIn="report, form"/>
        <sButton id="b1" label="Change state" click="currentState='State2';"/>
        <s:TextInput excludeFrom="form"/>
```

In addition, you can set the value for any attribute of an MXML component by appending the state name to the attribute with dot notation; for example:

```
        <s:CheckBox label.default ="Checkbox"
                    label.report="Report Name: "
                    label=""/>
```

Flex 4 also introduced the `stateGroups` attribute, which lets you group and apply two or more states together. For example:

```
<s:states>

<s:State name="A" stateGroups="G1"/>

<s:State name="B" stateGroups="G2"/>

<s:State name="C" stateGroups="G1"/>

</s:states>

…

<s:Button label="Button A" includeIn="G1"/>

<s:Button label="Button B" includeIn="G2"/>

<s:Button label="Button C" includeIn="G1"/>
```

The code above produces the following results:

When currentState = A or C

Button A

Button C
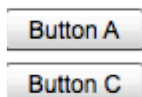
When currentState = B

Button B
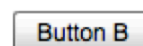
New MXML Tags

In addition to the existing MXML tags (`Binding`, `Component`, `Metadata`, `Model`, `Script`, `Style`, `XML`, `XMLList`) there are six new tags in Flex 4: `Declarations`, `Definition`, `DesignLayer`, `Library`, `Reparent`, and `Private`.

Declarations

In Flex 4, any components you declare that are not UI components must be placed inside of a `Declarations` tag; for example:

```
<?xml version="1.0"?>
<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:HTTPservice id="myService"/>
        </fx:Declarations>

        <s:Button label="My Button" />
</s:Application>
```

If you don't use place your non-UI components inside a `Declarations` tag, you'll get an error at compile time.

Library

Use the `<fx:Library>` tag to define named graphic `<fx:Definition>` children (see `Definition` below). 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.

Definition

You use an `<fx:Definition>` tag inside an `<fx:Library>` tag when you want to declare a UI component inside of a MXML file and use the component somewhere else in the file. This is the equivalent of defining the component in a separate MXML or ActionScript file.

Each `Definition` created must have a `name` attribute defined. This is needed to use the component in your code. Defining a component using the `Definition` tag doesn't automatically instantiate that component and add it to the display list.

Here is an example:

```
<?xml version="1.0"?>
<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:Library>

        <mx:Canvas>
                <fx:MySquare x="0" y="0" height="20" width="20"/>
                <fx:MySquare x="25" y="0" 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 example above, the new class is a subclass of mx.graphics.Group. The scope of this class is limited to the document, so you should treat it as a private ActionScript class.

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; for example:

```
<?xml version="1.0"?>
<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:Canvas>
            <fx:MySquare x="0" y="0" height="20" width="20"/>
            <fx:MySquare x="25" y="0" height="20" width="20"/>
    </mx:Canvas>

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


Reparent

You use the `<fx:Reparent>`tag to change the parent container of a component as part of a change of view state:

```
<fx:Reparent target="targetComp" includeIn="stateName">
```

`Target` specifies the target component, and `includeIn` specifies a view state. When the current view state is set to *stateName*, the target component becomes a child of the parent component of the `<fx:Reparent>` tag. Here is an example:

```
<s:states>
        <s:State name="Parent1"/>
        <s:State name="Parent2"/>
</s:states>

<mx:HDividedBox height="25%" width="100%" borderStyle="inset">
        <mx:VBox id="VB1">
                <mx:Button id="setCB" includeIn="Parent1"/>                    </mx:VBox>
        <mx:VBox id="VB2">
```

```
                    <fx:Reparent target="setCB" includeIn="Parent2"/>                    </mx:VBox>
</mx:HDividedBox>
```

The `<fx:DesignLayer>` tag is for internal use only. Flash Catalyst uses this tag, for example.

Two-way Data Binding

With *two-way data binding* a variable is bound to a property of an object, such that any time one of them is updated the other is automatically updated. When the value of the variable changes the object automatically updates its property, and when the object property changes, the variable is updated automatically.

In Flex 3, two-way data binding was implemented 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:

- Inline declaration using the `@{bindable_property}` syntax; for example:
  ```
  <s:TextInput id="t1" text="@{t2.text}"/>

      <s:TextInput id="t2"/>
  ```
- In MXML; for example:
  ```
  <fx:Binding source="a.property" destination="b.property" twoWay="true"/>
  ```

MXML Graphics and FXG

Among the biggest changes introduced in Flex 4 is the addition of MXML graphics and support for FXG. While both of these concepts deal with drawing vector based graphics, they are not the same.

MXML Graphics

MXML graphics are a collection of classes that you can use to define interactive graphics—that is, graphics that can be changed at runtime. This is not a new language, but an addition to the MXML language and Flex framework. These classes are part of the Flex SDK (most of them are part of the *mx.graphics* and *spark.primitives* packages) and the MXML compiler maps each MXML graphic tag to a corresponding ActionScript class.

MXML graphic tags can be added as children of the root Application tag or any other container or group. They have an implicit depth order: each tag is rendered on top of the previous sibling. Because each tag is backed by an ActionScript class, you can create graphics using ActionScript instead of MXML.

MXML graphics define:
- Graphics and text primitives
- Fills, strokes, gradients, and bitmaps
- Filters, masks, alphas, transforms, and blend modes

Here is a list of MXML graphic tags and their ActionScript implementations (please note that the s namespace is for the Spark component library):

`<s:BitmapFill>`	`mx.graphics.BitmapFill`
`<s:BitmapImage>`	`spark.primitives.BitmapImage`
`<s:ColorTransform>`	`flash.geom.ColorTransform`
`<s:Ellipse>`	`spark.primitives.Ellipse`
`<s:GradientEntry>`	`mx.graphics.GradientEntry`
`<s:Graphic>`	`spark.primitives.Graphic`
`<s:Line>`	`spark.primitives.Line`
`<s:LinearGradient>`	`mx.graphics.LinearGradient`
`<s:LinearGradientStroke>`	`mx.graphics.LinearGradientStroke`
`<s:Path>`	`spark.primitives.Path`
`<s:RadialGradient>`	`mx.graphics.RadialGradient`
`<s:RadialGradientStroke>`	`mx.graphics.RadialGradientStroke`
`<s:Rect>`	`spark.primitives.Rect`
`<s:SolidColor>`	`mx.graphics.SolidColor`
`<s:SolidColorStroke>`	`mx.graphics.SolidColorStroke`
`<s:RichText>`	`spark.primitives.RichText`
`<s:Transform>`	`mx.geom.Transform`
`<s:BevelFilter>`	`spark.filters.BevelFilter`
`<s:BlurFilter>`	`spark.filters.BlurFilter`
`<s:ColorMatrixFilter>`	`spark.filters.ColorMatrixFilter`
`<s:DropShadowFilter>`	`spark.filters.DropShadowFilter`
`<s:GlowFilter>`	`spark.filters.GlowFilter`
`<s:GradientGlowFilter>`	`spark.filters.GradientGlowFilter`
`<s:GradientBevelFilter>`	`spark.filters.GradientBevelFilter`
`<s:ShaderFilter>`	`spark.filters.ShaderFilter`

You can write the MXML graphics code using a code editor or you can use a tool such as Flash Catalyst to generate the code. Typically, you use MXML graphics to create the skins for Flex 4 components.

For example, you can use MXML graphics to draw this shape:

Here is the code:

```xml
<?xml version="1.0" encoding="utf-8"?>
<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:Group x="50" y="50">
        <s:Rect height="75" radiusX="5.62472" radiusY="5.99958" width="75" x="0.5"
y="0.5">
            <s:fill>
                <s:SolidColor color="#494848"/>
            </s:fill>
            <s:stroke>
                <s:SolidColorStroke caps="none" joints="miter" miterLimit="10"
weight="1"/>
            </s:stroke>
            <s:filters>
                <s:GlowFilter alpha="0.56" blurX="7.5" blurY="7.5"
color="#777777" inner="true" quality="2" strength="2"/>
            </s:filters>
        </s:Rect>
        <s:Group x="22" y="28">
            <s:Line x="0" xFrom="37" y="14"/>
            <s:Path data="M 36.785 11.654 L 23.489 11.654 L 22.785 11.654 L 22.785
0 L 0 13.154 L 22.785 26.309 L 22.785 14.654 L 23.489 14.654 L 36.785 14.654 L 36.785 11.654
Z" winding="nonZero" x="0.215" y="0.346">
                <s:fill>
                    <s:SolidColor color="#FFFFFF"/>
                </s:fill>
            </s:Path>
        </s:Group>
    </s:Group>

</s:Application>
```

FXG

FXG is an XML-based graphics interchange format for the Adobe Flash Platform. You can use Adobe Illustrator and Adobe Photoshop to create graphical assets, export them as FXG, and use them in Flex.

You can't use the FXG language inside of MXML documents/components; instead you define separate .FXG files and then reference these files inside of MXML. The root tag for an FXG document is always `<Graphic>` and there can be only one `<Graphic>` node in a FXG document. For example you could define AlphaMaskComponent.fxg as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- /comps/AlphaMaskComp.fxg -->
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
    <Ellipse width="400" height="200" maskType="alpha">
        <mask>
            <Group>
```

```
            <Rect width="100" height="100">
                    <fill>
                            <SolidColor alpha="0.1"/>
                    </fill>
            </Rect>
        </Group>
    </mask>
    <fill>
            <SolidColor color="#FF00FF"/>
    </fill>
  </Ellipse>
</Graphic>
```

You would then use it in MXML like this:

```
<?xml version="1.0" encoding="utf-8"?>
<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:AlphaMaskComponent/>

</s:Application>
```

Most FXG elements have MXML graphic equivalents, although FXG elements support only a subset of the attributes supported by MXML graphics tags.

Here are some differences and similarities between FXG and MXML graphics:

- Both are used to create graphics in Flex applications.
- FXG is compiled (the FXG rendering model follows the Flash Player 10 rendering model very closely, and the compiler optimizes FXG directly into SWF tags. understood natively by Flash Player). MXML graphics are rendered at runtime (the Flex compiler transforms them into ActionScript, which is transformed into ActionScript bytecode). Thus FXG is highly optimized when compared to the same graphic drawn using MXML graphics.
- FXG uses its own namespace (`http://ns.adobe.com/fxg/2008`), whereas MXML graphics use the namespace of the containing document (usually the Spark namespace).
- Because FXG is highly optimized at compile time, it can't be changed at runtime. Thus if you want to use it for skinning a button you won't be able to add different colors for the background for each state (over, disabled, up, and down). Likewise, no data binding or event handling is available in FXG.

For more information, see [FXG and MXML graphics](#).

CSS

Flex 4 has significantly improved support for styling Flex applications using CSS. Within the CSS file, you need to specify the namespace of the Flex library you want to customize using CSS. For example, the following CSS file sets the styles for the MX Button and Spark Button:

```
/* CSS file */
@namespace s "library://ns.adobe.com/flex/spark";
@namespace mx "library://ns.adobe.com/flex/mx";

s|Button {
        color:#ffffff;
}

mx|Button {
        color:#ffffff;
}
```

You can use multiple classes for the same component. For example, if you've defined two CSS classes called *buttonOk* and *buttonGreen,* you can apply styles to both of them using the following code:

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

        .buttonOK {
                font-size: 12;
        }
        .buttonGreen {
                color: #00ff00;
        }
</fx:Style>
<s:Button styleName="buttonOk buttonGreen"/>
```

You can also define and apply styles using ID selectors; for example:

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

        #myButton {
                color:#ffffff;
        }
</fx:Style>
<s:Button id="myButton"/>
```

You can use descendant selectors. For example, you can apply a style only to buttons inside of a Spark Panel:

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

        s|Panel s|Button {
                color:#ffffff;
        }
</fx:Style>
<s:Panel>
        <s:Button label="White label"/>
</s:Panel>
<s:Button  label="Black label"/>
```

You can use pseudo selectors; for example:

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


        s|Button:over {
                color: #000000;
```

```
        }

        s|Button:up {
                color: #ffffff;
        }
</fx:Style>
<s:Button label="My Button"/>
```

To apply style sheets at runtime, compile the CSS file into a SWF and load this SWF using the StyleManger `loadStyleDeclaration()` method. Here are the basic steps to achieve this in Flash Builder:

- Create a CSS file; for example myStyle.css
- In Flash Builder right-click the file in the Package Explorer and select *Compile CSS to SWF*. You will find a myStyle.swf file inside the bin-debug folder.
- When you want to load this style at runtime, use `loadStyleDeclarations`; for example:
  `styleManager.loadStyleDeclarations("myStyle.swf");`
  The first parameter is the path of the SWF file and the second (optional) parameter specifies whether the style is applied immediately after it is loaded (this is the default behavior).

Flex Themes

A theme defines the look and feel of an application. It can define a limited set of styles, for example the font size and family, or it can redefine all Flex UI components using a mix of CSS, images, and skins.

The default theme for a Flex 4 application is called Spark. Thus all Flex UI components in Flex 4 have this theme by default, including the MX components (Flex 3 components). You can change the theme using the `theme` compiler argument. If you use Flash Builder 4, you can use the Properties dialog box to apply themes to an existing project and to add new themes.
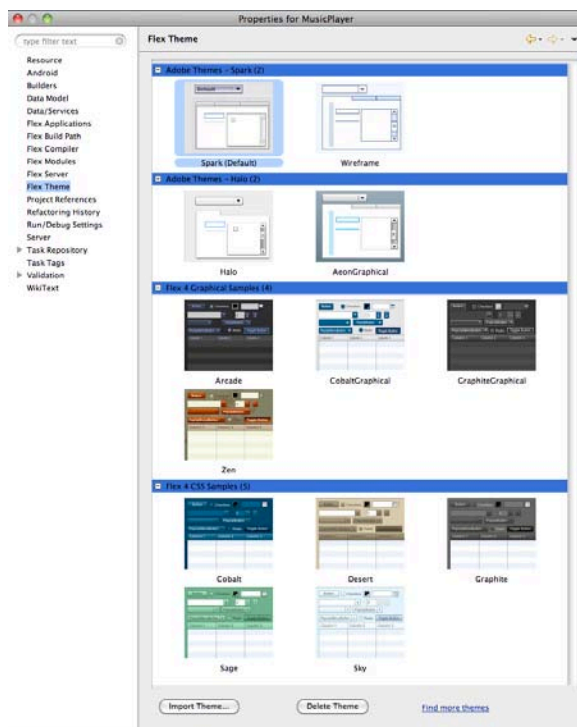
Flash Builder offers a number of themes to choose from, and if you want, you can create your own themes. It is possible to apply multiple themes to the same project. However, if the themes you apply set the appearance for the same element, then, as with CSS, the last theme applied takes precedence.

If you want the MX components to have the Flex 3 look and feel, then you can use the Halo theme. Note that Spark components will use the same skin as the one defined by the Spark theme.

You can find the code for the built-in themes in the `Flex 4 SDK/frameworks/themes/` folder.

Creating a Flex Theme

In general, themes take the form of a SWC file, which comprises a CSS file and skinning assets. The skinning assets can be images (JPEG, GIF, PNG) or programmatic assets compiled into an SWF file.

You can also create a theme is by using only a CSS file as is without compiling it into a SWF file. Alternatively, you can have a CSS file and a SWF file (this is how the AeonGraphical skin was created).

For example, consider a situation in which you've created two programmatic skins, SpecialButtonSkin and SpecialCheckBoxSkin, in MXML. You've also created the following CSS file, which references these class names:

```
//mycss.css
@namespace s "library://ns.adobe.com/flex/spark";
s|Button {
       skinClass: ClassReference("SpecialButtonSkin");
}
s|CheckBox {
       skinClass: ClassReference("SpecialCheckBoxSkin");
}
s|panel {
       color:#ff0000;
}
```

You could then use the command line component compiler (*compc*) with the `include-file` and `include-classes` options to include the CSS, image assets, and skins. Use `o` option to specify the output theme SWC file; for example:

```
compc    -source-path c:/projects/flex/mytheme
         -include-file mycss.css c:/projects/flex/mytheme/mycss.css
         -include-classes  SpecialButtonSkin SpecialCheckBoxSkin
         -o c:/projects/flex/MyTheme.swc
```

Instead of passing all these arguments on the command line, you can create a *Flex Config* file and pass it as an argument to the `load-config` option to the *compc* command.

```
compc –load-config myconfig.xml
```

The following Flex configuration file can be used in place of the command line options in the example above:

```
<?xml version="1.0"?>
<flex-config>
        <output>MyTheme.swc</output>
        <include-file>
                 <name>mycss.css</name>
                 <path>c:/projects/flex/mytheme/mycss.css</path>
        </include-file>
        <include-classes>
                 <class> SpecialButtonSkin </class>
                 <class> SpecialCheckBoxSkin</class>
        </include-classes>
</flex-config>
```

You can also invoke the component compiler to make a theme SWC file by creating a Library Project in Flash Builder.

For more information see, see [Creating a theme SWC file](#).

Applying a Theme

You can apply a theme using the `theme` option and the MXML compiler; for example:

```
mxmlc -theme="C:\Program Files\Adobe\Flex\sdks\4.0.0\frameworks\themes\Halo\halo.swc"
MyApp.mxml
```

Alternatively, you can open the Flash Builder Properties dialog box for the project you want to customize and select the theme you want. (If it is not already there, you can import the theme before applying it.)

One final note: You can't apply a theme by just adding it to the *libs* folder. If you want to apply themes at runtime, read the CSS chapter to learn how (basically you use a SWF instead of a SWC and call the StyleManager `loadStyleDeclarations` method).
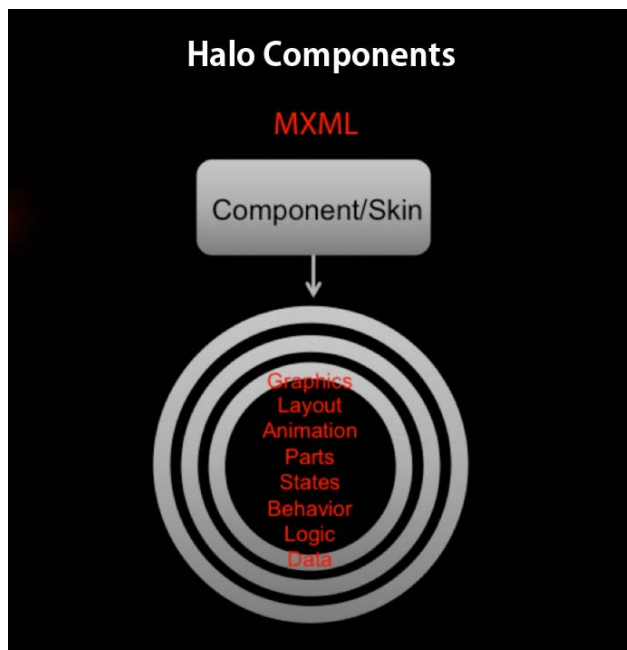
Spark Components

Many of the features presented so far can be seen as evolutionary features: small improvements or additions. The Spark components are definitely a revolutionary feature. The new Spark component architecture, introduced in Flex 4, is a significant departure from the previous MX component model of Flex 3 and Flex 2.

In a nutshell, Spark components introduce a clear separation between the logic and data of a component and its visual appearance (skin). To support this separation, every Spark component has at least two classes.

One class (typically written in ActionScript) gives the component MXML tag name (for example, Button) and encapsulates the core behavior of the component: defining what events the component dispatches, the data represented by the component (for a button this could be the label), and managing subcomponents and states (for a button these could be up, down, over, and disabled).

Coupled with this class there is a skin class that manages everything related to how the component is rendered on the screen including layout, animation, changing appearance in different states, transitions, and representing the data. Typically you create a skin class using MXML graphics, FXG, or both.

From a tooling perspective, you can use a text editor to create the skin class or you can use the new workflows enabled by Flash Catalyst and Illustrator or Photoshop to create the skins visually.
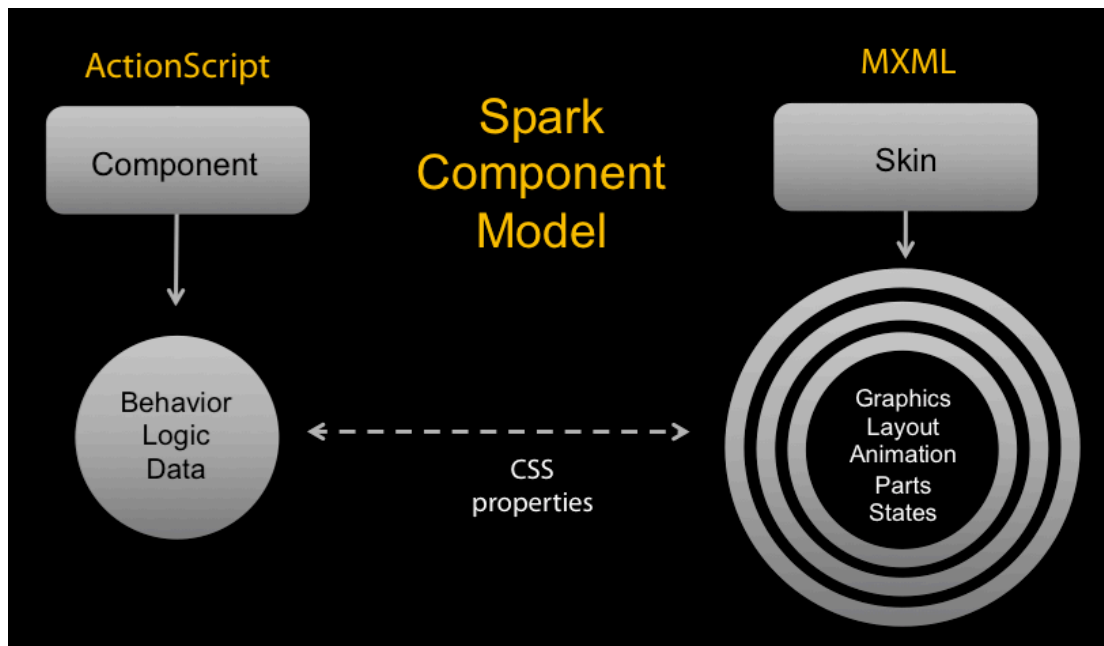
Flex 4 applications require Flash Player 10 or higher. Flash Player 10 introduced the new Flash Text Engine (FTE) and Pixel Bender, features used by Flex 4.

Here is a list of Spark components:

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

In the list above you'll notice that there is not a Spark component for each MX component. This is something Adobe is working on to address in future Flex SDK releases. In the same time there will not be a Spark equivalent for each MX component. Still, in Flex 4 you have access to all the Flex 3 components and you can mix these with Flex 4 components on the same project as you wish.

Like MX components, Spark components are subclasses of the *UIComponent* class. However, a set of new classes that inherit from *UIComponent* were introduced in Flex 4, and these new classes form the base classes for any Spark component.

To create a Spark component that is skinnable, create a subclass of *spark.components.supportClasses.SkinnableComponent*.

To create a Spark container that is skinnable, you subclass *spark.components.SkinnableContainer*. There are Spark containers that are not skinnable, for example the Group container.

Two other important containers are *DataGroup* and *SkinnableDataContainer*. These containers can be used to display data. For example, if you want to create a list you can use *SkinnableDataContainer*. If you don't need a skin, then you can use *DataGroup*.

All Spark containers support children that implement the *IVisualElement* interface– Spark components, MX components, and *spark.primitives.supportClasses.GraphicElement* classes implement this interface. Thus you can add UI components (Spark or MX), MXML graphics tags, or FXG to a Spark container.

To create a skin for a Spark container, typically you create an MXML file that has a Group with the `id` set to `contentGroup` and some MXML graphics tags or FXG to create the look and feel.

Finally, to create a skin for a Spark component, subclass the *spark.components.supportClasses.Skin* class.

The Skinning Contract – Skinning Spark Components

Separating the component's logic and behavior from its appearance in two different classes is good; however these classes must interact to provide a rich user experience. A beautiful skin for a button that doesn't react to mouse over or click events would not be very interesting.

In order to enable interactivity between the component and its skin, a contract exists between these two entities. The skinning contract has these three parts: Data, Parts, and States.

The following illustrates how these parts are related for a simple Spark component, the Button.

Button component	**Skin component**

Data:
   [Bindable]
   public function set label   `<s:Label text="{hostComponent.label}"/>`

Parts:
   [SkinPart(required="false")]
   public var labelDisplay:TextBase;   `<s:Label id="labelDisplay"/>`

States:
   [SkinState("up")]   `<s:State name="up"/>`
   [SkinState("over")]   `<s:State name="over"/>`
   [SkinState("down")]   `<s:State name="down"/>`
   [SkinState("disabled")]   `<s:State name="disabled"/>`

Data

The component class holds the data. For a button this could be the label string:
```
[Bindable]
public function set label
```

The skin class defines a *Label* component in order to display the data on the screen. It can read the data from the component class using the `hostComponent` variable:
```
<s:Label text="{hostComponent.label}"/>
```

Parts

The component class defines the skin parts and whether they are required or optional; for example:
```
[SkinPart(required="false")]
public var labelDisplay:TextBase;
```

 The skin class implements the skin parts (or it can choose to ignore them if they are optional):
```
<s:Label id="labelDisplay"/>
```

States

The component class defines the states supported:
```
[SkinState("up")]
[SkinState("over")]
[SkinState("down")]
[SkinState("disabled")]
```

The skin class can choose to react to the state changes or not. At a minimum, it has to declare the host component's states:
```
<s:State name="up"/>
<s:State name="over"/>
```

```
<s:State name="down"/>
<s:State name="disabled"/>
```

You can set a relationship between a skin class and the components that can be skinned with it. For example, you can add the following in the skin class:

```
<fx:Metadata>
        [HostComponent("spark.components.Button")]
</fx:Metadata>
```

Creating Spark Components and Skins

There are two methods you can use to create a Spark component and skin:
- Start from scratch and subclass *SkinnableComponent* and *Skin* classes.
- Extend an existing Spark component or skin.

As an example, below is a custom Spark component that can be used to display the network connectivity status.

Here is the code for the component:

```
package org.corlan {
        import spark.components.supportClasses.SkinnableComponent;
        import spark.components.supportClasses.TextBase;

        //the states defined by the skin
        [SkinState("connected")]
        [SkinState("disconnected")]
        [SkinState("disabled")]

        public class ConnectionStatus extends SkinnableComponent {

                private var _content:String;
                private var _enabled:Boolean = true;
                private var _connected:Boolean = false;

                [SkinPart(required="true")]
                public var labelDisplay:TextBase;

                public function ConnectionStatus() {
                        super();
                }

                public function set label(val:String):void {
                        content = val;
                }

                public function get label():String {
                        return (content != null) ? content.toString() : "";
                }

                public function set connected(val:Boolean):void {
                        _connected = val;
                        invalidateState();
                }

                public function get connected():Boolean {
                        return _connected;
                }

                override public function set enabled(val:Boolean):void {
                        _enabled = val;
                        super.enabled = val;
                }
```

```actionscript
                private function set content(val:String):void {
                        _content = val;
                        // Push to the optional labelDisplay skin part
                        if (labelDisplay)
                                labelDisplay.text = label;
                }

                private function get content():String {
                        return _content;
                }

                override protected function partAdded(partName:String, instance:Object):void {
                        super.partAdded(partName, instance);

                        if (instance == labelDisplay) {
                                // Push down to the part only if the label was explicitly set
                                if (_content)
                                        labelDisplay.text = label;
                        }
                }

                override protected function getCurrentSkinState():String {
                        if (!_enabled)
                                return "disabled";

                        if (_connected)
                                return "connected";
                        else
                                return "disconnected";
                }

                private function invalidateState():void {
                        //invalidateProperties();
                        invalidateSkinState();
                }
        }
}
```

Here is the code for the associated skin class:

```xml
<?xml version="1.0" encoding="utf-8"?>
<s:SparkSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:s="library://ns.adobe.com/flex/spark"
     alpha.disabled="0.5">

        <fx:Metadata>
                [HostComponent("org.corlan.ConnectionStatus")]
        </fx:Metadata>

    <s:states>
        <s:State name="disconnected"/>
        <s:State name="connected" />
        <s:State name="disabled" />
    </s:states>

    <s:Group>
                <s:layout>
                        <s:HorizontalLayout/>
                </s:layout>
                <s:Label id="labelDisplay"
                                textAlign="center"
                                verticalAlign="middle"
                                maxDisplayedLines="1"
                                horizontalCenter="0" verticalCenter="1" color="#ff0000"
color.connected="#00ff00"/>
    </s:Group>
</s:SparkSkin>
```

Here is an example of code that uses this component:

```actionscript
private function changeConnStatus():void {
        connStatus.connected = connStatus.connected;
}

<corlan:ConnectionStatus id="connStatus"
```

```
                          label="Connection Status" skinClass="org.corlan.ConnectionStatusSkin"/>
```

Spark Components and Skins Lifecycle

So how does the skin class communicate with the component class? The component class must implement the `getCurrentSkinState()` method. This method returns the state that the skin should display. To push the change to the skin, invoke the component class `invalidateSkinState()` method. In turn, the skin will call `getCurrentSkinState()`and will change its state accordingly.

What about injecting the data from the component to the skin (for example the label string for a button)? There are two different ways to achieve this. The skin can pull the data using the `hostComponent` variable. Alternatively, the component object can push the data into the skin using the `skinPart`. If you want to use the latter method, you might want to override the `partAdded()` method (you can see this approach used in the previous example).

```
        Component            Skin

           Constructor(), …
        commitProperties()
                 loadSkin()    Constructor()


   Skin.hostComponent = this
            addChild(skin)    initialize()
            findSkinParts()
               partAdded()
       getCurrentSkinState()    currentState = …
  measure()/updateDisplayList()/    commitProperties()/measure()/
            updateComplete        updateDisplayList()/updateComplete


   createDynamicPartInstance(),
   removeDynamicPartInstance(),
          getDynamicPartAt(),
           numDynamicParts().
```

For an example of adding a new skin state in a component that extends an existing one, see Justin Shacklette's Drawer Component in Flex 4.

For more on this topic, see Introducing skinning in Flex 4.

New Layout Engine

In Flex 3, the layout was tightly coupled with the container inside the individual controls. Each container type exposed a certain layout behavior. Components that shared the same functionality were often duplicated to achieve different layout behavior. Also, it was particularly difficult to make components that worked with collections display their children differently.

In Flex 4, this issue was solved by decoupling the layout from the component. Spark components can define their layout declaratively. The layouts can support both Spark and MX components in addition to FXG graphic primitives. Layouts can even be changed at runtime.

The following additional features are also available to developers:
1. Layout logic is abstracted in separate classes inheriting from *LayoutBase* (the superclass).
2. The *ILayoutElement* interface has been created to allow layout classes to manage the containers to which they apply.
3. Spark data containers have layout virtualization inherited from the base class *DataGroup*. Virtualization is intended to minimize the footprint and the startup time for containers with large numbers of items. When using virtualization, the measurements of the *DataGroup* container will be approximate based on the layout dimensions of the first element in the *itemRenderer* or on the dimensions of a typical element specified through the `typicalItem` property.
4. For Spark containers, the display order is decoupled from the child order.

As a direct consequence of these features, the stock layouts support 2D transformations and smooth per-pixel scrolling. Post-layout transforms allow developers to specify position properties without affecting the layout. Also, custom layouts can be created very easily and 3D transformations can be incorporated in these layouts.

Existing Spark Layouts

Flex ships with several layout classes that can be used with Spark containers, including BasicLayout, HorizontalLayout, TileLayout, and VerticalLayout. The layout classes are defined in the spark.layout package.

BasicLayout
The BasicLayout class uses absolute positioning. The children can be positioned either by specifying coordinates or by using constraints.
This layout is not supported by the Spark list-based controls. Do not use this layout with a Spark List control or any of its subclasses (such as ButtonBar, ComboBox, DropDownList, or TabBar). Also, the BasicLayout class does not support virtual layout.

Here is an example of BasicLayout:

```
<Group>
        <layout>
                <BasicLayout/>
        </layout>
        <Label text="content" baseline="40"/>
</Group>
```

HorizontalLayout

The *HorizontalLayout* class arranges the layout elements in a horizontal sequence, left to right, with optional gaps between the elements and optional padding around the elements.

During the execution of the `measure()` method, the default size of the container is calculated by accumulating the preferred sizes of the elements, including gaps and padding. When the `requestedColumnCount` property is set to a value other than `-1`, only the space for that many elements is measured, starting from the first element.

Here is an example of *HorizontalLayout*:

```
<s:List dataProvider="{objectsList}" itemRenderer="components.CustomerRendered" width="200">
        <s:layout>
                <s:HorizontalLayout clipAndEnableScrolling="true"  />
        </s:layout>
</s:List>
```

TileLayout

The *TileLayout* class arranges layout elements in columns and rows of equally-sized cells. The *TileLayout* class uses a number of properties that control orientation, count, size, gap, and justification of the columns and the rows as well as element alignment within the cells.

Here is an example of *TileLayout*:

```
<s:List dataProvider="{objectsList}" itemRenderer="components.CustomerRendered" width="200"
height="50">
        <s:layout>
                <s:TileLayout />
        </s:layout>
</s:List>
```

VerticalLayout

The *VerticalLayout* class arranges the layout elements in a vertical sequence, top to bottom, with optional gaps between the elements and optional padding around the sequence of elements.

As with *HorizontalLayout*, during the execution of the `measure()` method, the default size of the container is calculated by accumulating the preferred sizes of the elements, including gaps and padding. When `requestedRowCount` is set, only the space for that many elements is measured, starting from the first element.

Here is an example of *VerticalLayout*:

```
<s:List dataProvider="{objectsList}" itemRenderer="components.CustomerRendered">
        <s:layout>
                <s:VerticalLayout gap="15" />
        </s:layout>
</s:List>
```

Creating Custom Spark Layouts

To create a custom Spark layout, you need to inherit from the base class *LayoutBase* and
override the `updateDisplayList` method.
When a layout is assigned to a container, the target property of the layout will receive a
reference to that container. The custom layout may decide on the positioning and the
size of the elements inside the container. For such a scenario, special attention has to be
paid to the type of container. If the container supports virtualization (for example, if it's
a *DataGroup* or List), finding a certain element in the container has to be done through
the `getVirtualElementAt` method of the target object. If the container does not support
virtualization, the `getElementAt` method has to be used.

It's fairly easy to resize and position elements using the *ILayoutElement* API; for
example:

```
public class Horizontal3DLayout extends LayoutBase
{
        override public function updateDisplayList(width:Number, height:Number):void
        {
                var numElements:int = target.numElements;
                var targetWidth:Number = target.width;
                var middleHeight:Number = target.height / 2;

                for (var i : int = 0; i < numElements; i++)
                {
                        var element:ILayoutElement = target.getVirtualElementAt(i);
                        element.setLayoutBoundsSize(NaN, NaN, false);

                        var elWidth:Number = NaN;
                        var elHeight:Number = NaN;

                        if (i < numElements/2)
                        {
                                element.setLayoutMatrix(new Matrix(1.6, 0, 0, 1.6, 0, 0), true);
                                elWidth = element.getLayoutBoundsWidth(true);
                                elHeight = element.getLayoutBoundsHeight(false);
                                element.setLayoutBoundsPosition(i * elWidth , middleHeight +
elHeight, true);
                        }
                        else
                        {
                                element.setLayoutMatrix(new Matrix(0.9, 0, 0, 0.9, 0, 0), true);
```

```
                                elWidth = element.getLayoutBoundsWidth(true);

                                elHeight = element.getLayoutBoundsHeight(false);

                                element.setLayoutBoundsPosition(targetWidth – i * elWidth ,
middleHeight – elHeight, true);
                        }
                }
        }
}
```

For more information on Spark layouts, the online documentation is a good place to
start, especially the topics covering renderers, skins, *LayoutBase*, *ILayoutElement*,
GroupBase, *DataGroup*, and virtualization. Also, see the following resources:

- Spark layouts with Flex 4
- Flex 4 & Custom Layouts
- Flex 4 Deep Dive: UI and Dev
- Differences between Flex 3 and Flex 4

New Animation Engine

All of the new Flex 4 effects are subclasses of the *Animation* class, which is a subclass of the *Effect* class. This allows the Flex 4 framework to expose a parallel hierarchy of effects that does not break compatibility with the old ones, since the effects from the Flex 3 framework are subclasses of the *TweenEffect* class. The functionalities exposed by the *Animation* class are very similar to those exposed by the Tween class in Flex 3.

The new classes for animation are grouped under *spark.effects*.

The animation can be a change in position (performed by the Move effect), a change in size (performed by the Resize effect), a change in visibility (performed by the Fade effect), or any other animation used by effects or performed directly using the Animation class.

When defining animation effects, you typically create an instance of the *Animate* class, or of a subclass of Animate. This creates an Animation instance when the `play()` method is invoked. The *Animation* instance accepts start and end values, duration, and optional parameters such as easer and interpolator objects.

```
<s:Animate id="mover" target="{button}" duration="1000">
<s:SimpleMotionPath property="x" valueFrom="0" valueTo="100" />
<s:SimpleMotionPath property="y" valueTo="100" />
<s:SimpleMotionPath property="width" valueBy="20" />
</s:Animate>
…
<s:Button id="button" click="mover.play()"/>
```

The Animation class is a timing engine. This class calculates properties based on the properties passed to it, it does not animate objects by itself. Instead an animation target is passed to it, which is responsible for consuming the new properties and for animating objects.

Here is an example:

```
<s:Elastic id="elastic" />
<s:Animation id="animation" animationTarget="{new utilities.PropertyAnimator(rect)}"
easer="{elastic}" duration="1200">
<s:SimpleMotionPath property="x" valueFrom="0" valueTo="200" />
<s:SimpleMotionPath property="y" valueFrom="0" valueTo="200" />
</s:Animation>
```

Built-in Effects

Flex 4 include several subclasses of the *Animate* class to handle the most frequently used effects. These effects are responsible for providing the required properties to the *Animate* class and performing any related functionality to animate the targeted object. The names of these effects describe the type of animation performed by each of them.

- Resize effect – changes the width, height, or both dimensions of a component. When starting values for the resize are not provided, the current values of the object are used. Specifying any two of the `from`, `to` and `by` properties will allow the engine to calculate the third property; for example:

```
<s:Resize id="resizer" target="{rect}" widthTo="200" heightTo="200" />
...
<s:Button id="resizeBtn" label="Resize effect" click="resizer.play()" />
```

- Transform effects (*Move*, *Rotate*, *Scale*) – these effects only work on subclasses of *UIComponent* and *GraphicElement*. Transform effects operate relative to the transform center of the target. By default this is the upper-left corner of the target. To play the transform around the center point of the target, set the `autoCenterTransform` property to `true`; for example:

```
<s:Move id="mover" target="{rect}" xTo="200" yTo="200" />
<s:Rotate id="rotator" target="{rect}" angleTo="120" autoCenterTransform="true" />
<s:Scale id="scaler" target="{rect}" scaleXBy="2" scaleYBy="2" autoCenterTransform="true" />
…
<s:Button id="moveBtn" label="Move effect" click="mover.play()" />
<s:Button id="rotateBtn" label="Rotate effect" click="rotator.play()" />
<s:Button id="scaleBtn" label="Scale effect" click="scaler.play()" />
```

- *Fade* effect –animates the `alpha` property of a component.

```
<s:Fade id="fader" target="{rect}" alphaTo="0.5" />
…
<s:Button id="fadeBtn" label="Fade effect" click="fader.play()" />
```

- *AnimateColor* effect – animates a change in the `color` property over time, interpolating between the given `colorFrom` and `colorTo` values on a per-channel basis.

```
<s:AnimateColor id="colorEffect" target="{myGradient}" colorFrom="0x0000FF" colorTo="0xFF0000"
repeatCount="2" repeatBehavior="reverse" />
…
<s:Button id="clrBtn" label="Color effect" click="colorEffect.play()" />
```

- *Keyframe* animation – animates between keyframes with the help of the *MotionPath* and *Keyframe* classes. The *Keyframe* objects, specified as part of a *MotionPath*, are key/value pairs that a specified property will pass through over the duration of an animation; for example:

```
<s:Animate id="colorKeyframes" target="{myGradient}">
            <s:MotionPath property="alpha">
                  <s:Keyframe time="500" value="1.0" />
```

```
                    <s:Keyframe time="500" value="0.3" />

                    <s:Keyframe time="1100" value="0.0" />

                    <s:Keyframe time="300" value="0.3" />

                    <s:Keyframe time="300" value="0.5" />

                    <s:Keyframe time="300" value="1.0" />

            </s:MotionPath>

            <s:MotionPath property="color">

                    <s:Keyframe time="1000" value="0x00FF00" />

                    <s:Keyframe time="2000" value="0xCCCCCC" />

            </s:MotionPath>

        </s:Animate>
…
<s:Button id="clrKeyframesBtn" label="Keyframe animate effect" click="colorKeyframes.play()"
/>
```

Advanced Effects

Flex 4 supports more advanced effects, including 3D effects, pixel shader effects, chaining effects, and more.

- 3D effects

  3D effects add support for the z-axis when performing an animation. Increasing the z value will make the object appear farther away from the viewer, while decreasing the value will have the opposite result, moving the object towards the viewer visually. As with the 2D transform effects, using the 3D transform effects may modify the layout of the parent container for the animated object. To override this behavior, set the `applyChangePostLayout` property of the effect to `false`. Here is an example of 3D effects:

```
<s:Move3D id="moverBis" duration="600" xBy="-200" zBy="-200" autoCenterTransform="true"
repeatCount="2" repeatBehavior="reverse" target="{rect}" />

<s:Rotate3D id="rotatorBis" target="{rect}" angleXFrom="0" angleXTo="180" angleYFrom="0"
angleYTo="180" duration="1000" autoCenterTransform="true" />

<s:Scale3D id="scalerBis" target="{rect}" scaleXBy="-.25" />

...
<s:Button id="moveBtnBis" label="3D Move effect" click="moverBis.play()" />

<s:Button id="rotateBtnBis" label="3D Rotate effect" click="rotatorBis.play()" />

<s:Button id="scaleBtnBis" label="3D Scale effect" click="scalerBis.play()" />
```

- Pixel shader effects

  These effects are used to apply animation to a target object that has a bitmap representation for the state before the animation starts and after the animation completes.

  Working with Image objects is obvious, but the shader effects extend their capability to animate any Flex component. To achieve that, these effects capture a bitmap image of the component before the animation and one after and apply the animation between those two images. Here is an example:

```
[Embed(source="assets/Blue hills.jpg")]

private const ImgA:Class;


[Embed(source="assets/Sunset.jpg")]

private const ImgB:Class;

...

<s:CrossFade id="crossFade" bitmapFrom="{new ImgA().bitmapData}" bitmapTo="{new
ImgB().bitmapData}" target="{faded}" duration="2000" repeatCount="2" repeatBehavior="reverse"
/>

<s:Wipe id="wipe" bitmapFrom="{new ImgA().bitmapData}" bitmapTo="{new ImgB().bitmapData}"
target="{faded}" duration="2000" repeatCount="2" repeatBehavior="reverse" />

...

<s:Button id="crossFadeBtn" label="Cross fade effect" click="crossFade.play()" />

<s:Button id="wipeBtn" label="Wipe effect" click="wipe.play()" />
```

- *AnimateFilter*

  The *AnimateFilter* effect differs from the other effects because it animates properties of a filter applied to a target rather than the properties of the target itself. The filters are included in the same package; the common filters are *DropShadowFilter*, *GlowFilter*, *BlurFilter*, and *ShaderFilter*. Here's an example:

```
<s:AnimateFilter id="animF" target="{shadowRect}" bitmapFilter="{new
spark.filters.DropShadowFilter()}">

        <s:SimpleMotionPath property="color" valueFrom="0" valueTo="0x0000FF" />

        <s:SimpleMotionPath property="distance" valueFrom="0" valueTo="20" />

        <s:SimpleMotionPath property="angle" valueFrom="270" valueTo="360" />

</s:AnimateFilter>

...

<s:Button id="animFilterBtn" label="AnimateFilter effect" click="animF.play()" />
```

- Chaining effects (Parallel vs. Sequence)

  Effects can be chained to run either in parallel or in a sequence. New Spark implementations have been provided for Parallel and Sequence composite effects.

  The effects are used to do exactly what their name suggests. They either run child effects in parallel or in a sequence, one after the other.

```
<s:Sequence id="seq">

        <s:Move target="{rect}" xBy="200" />

        <s:Rotate target="{rect}" angleBy="360" autoCenterTransform="true" />

        <s:Move target="{rect}" xBy="-200" />

</s:Sequence>


<s:Parallel id="parallelEffect">

        <s:Rotate target="{rect}" angleTo="360" autoCenterTransform="true" />

        <s:AnimateColor target="{myGradient}" colorFrom="0x0000FF" colorTo="0xFF0000"
repeatCount="2" repeatBehavior="reverse" />

</s:Parallel>

...

<s:Button id="parallelBtn" label="Parallel effect" click="parallelEffect.play()" />
```

```
<s:Button id="seqBtn" label="Sequence effect" click="seq.play()" />
```

Spark effects support all the general events, such as `effectStart`, `effectStop`, and `effectEnd`. They also dispatch `effectRepeat` when the effect begins a new repetition and `effectUpdate` every time the effect updates a target.

You can change the speed of an animation by using an easing class with the effect. Flex supplies Spark easing classes–including Bounce, Elastic, Linear, Power, and Sine–in the *spark.effects.easing* package.

For more information, see the following resources:
- Using Adobe Flex 4
- Keyframe Animation in Flex 4
- Effects and Animation - Hello! Flex 4
- Effects in Adobe Flex 4 – Part 1: Basic effects
- Effects in Adobe Flex 4 – Part 2: Advanced graphical effects
- Creating a simple property animation in Flex 4
- The Animation Class in Flex 4

What's New in Adobe AIR 2

AIR 2 is the new release of the Adobe platform that enables developers to create Rich Internet Applications that are executable outside of the browser.
Since the initial release of AIR, there have been several language enhancements and bug fixes, but not many important new features until AIR 2. This version includes many exciting new features and performance improvements.

Native Processes

One of the most requested features by developers, the Native Processes feature allows AIR applications to start and interact with native processes. To have access to the native process, you must use a native application installer to install your application instead of the application installer built into the AIR runtime.

The *flash.desktop.NativeProcess* class allows an AIR application to launch, interact with, and terminate a process. Programmatic interaction is optional and asynchronous; it is achieved via standard streams. If the process cannot communicate through these streams, the AIR application cannot communicate with it.

The *flash.desktop.NativeProcessStartupInfo* class contains information regarding a process to be launched. An instance of this class is passed as an argument to the *NativeProcess.start()* function.

The *flash.events.NativeProcessExitEvent* is the standard event class dispatched by the native process when it exits. This event will contain the process exit code, so it can be used for further action.

Updated Version of WebKit

A new branch of the WebKit browser engine—the equivalent of the Safari 4.0.3 branch—was used for AIR 2.

Notable features are:
- Versioned WebKit behavior to not break AIR 1.5 compatibility
- Adding the `<HEAD>` tag to the document if none is present
- Adding error reporting for a XMLHttpRequest call to a nonexistent resource
- Zero-length XMLHttpRequest POST will not be transformed into a GET as in AIR 1.5
- Improved support for JavaScript, CSS3, the `canvas` tag, and `data:` URLs.

The new JavaScript engine (SquirrelFish Extreme) runs applications up to twice as fast as before without any code modification.
The performance benchmarks below show an impressive performance improvement from AIR 1.5 to AIR. The better results achieved by Safari are due the size optimizations that AIR 2 must comply with.

Performance benchmark using Google V8 and WebKit SunSpider:

Operating System / Benchmark	AIR 1.5.3	AIR 2	Safari 4.0.4
Windows XP / V8 (bigger is better)	158.6	1157.8	1509.4
Windows XP / SunSpider (lower is better)	3286.4	1625.4	666.2
Mac OS X 10.6 / V8 (bigger is better)	374.4	2522.8	2619
Mac OS X 10.6 / SunSpider (lower is better)	1758.8	608.2	374.4

Also, in terms of CSS3 support, AIR 2 shipped with these notable features:
- 2D transformations, animations, and transitions
- Scrollbar styles – proprietary CSS properties for skinning and configuring scrollbars
- Text column support – reflowing text in a container across an arbitrary number of columns
- Zoom – AIR2 supports the new zoom property

The New Networking API

AIR 2 brings several key networking enhancements including secure and server sockets, IPv6, and UDP support.

All networking classes, including socket related classes, support both IPv4 and IPv6. The new *flash.net.DatagramSocket* class provides the ability to connect with other hosts using the UDP internet protocol.

The *flash.net.SecureSocket* class allows applications to connect with trusted servers using SSL and TLS for encrypted communication. The socket will connect only if the operating system verifies that there are no issues with the certificate or the certificate chain.

An AIR application can act as a server using the *flash.net.ServerSocket* class. After adding a listener to the server socket, the code will be notified when a connection has been established through the *flash.events.ServerSocketConnectEvent* event.
Other improvements include support for interrogating DNS resource records (*flash.net.dns.DNSResolver*) and the network interfaces available on the host (*flash.net.NetworkInfo* and *flash.net.NetworkInterface*).

Better Operating System Integration

AIR 2 is more tightly integrated with the operating system and the file system through features such as detection of mass storage devices, the open document API, an improved printing API, and support for native installers.

Management of mass storage devices is handled through the *flash.filesystem.StorageVolumeInfo* class. The class represents each available device currently accessible on the computer and gives information on each device's properties. Access is provided to the root directory as a *flash.filesystem.File* object.

You can open documents with the default application associated with the file's type using the `openWithDefaultApplication` method of the *flash.filesystem.File* object. For security reasons, executable types cannot be opened, unless the application was deployed by a native installer.

In terms of cross-platform printing, AIR 2 added even more classes to its already extensive support. The new classes are grouped under the *flash.printing* package (as before) and cover specifying the available paper size values (the *PaperSize* class), the method of printing (*PrintMethod*), and options for the print dialog displayed to the user (*PrintUIOptions*).

Local Audio Recording

In AIR 2, you can access the microphone through the *flash.media.Microphone* class. This class allows an application to interrogate the device and set various properties. Moreover, it provides access to the sound stream, so the data can either be played or saved. Data sampling on the fly can be achieved by using a *SoundTransform* object.

Multi-touch and Gesture Support

With the addition of multi-touch and gesture support in AIR 2, an AIR application can respond to gestures such as pinch, scroll, rotate, scale, and two-finger tap. Almost all of the visual components in the Flex framework provide multi-touch and gesture support via special events. To find out if a device is multi-touch enabled, use the *flash.ui.Multitouch* class. Multi-touch and gesture APIs are also available with the prerelease of AIR for Android.

More AIR 2 Features
Here are some other notable features of AIR 2:
- Global exception handling as part of the Flash Player update
- Screen Reader Support (Windows only) – runtime dialog boxes can be read
- Database transaction savepoints – the SQLite database included with AIR provides support for savepoints within transactions. The flash.data.SQLConnection provides three new functions: `SQLConnection.setSavepoint()`, `SQLConnection.releaseSavepoint()`, and `SQLConnection.rollbackToSavepoint()` for working with savepoints.
- File promises – you can use this feature to create and handle references to files that do not exist yet. At some point in the process, the files have to be created.

For more information on new AIR 2 features (including demo apps) see the following resources:
- [What's New in Two](#)
- [What's new about HTML, HTML5, CSS, and JavaScript in AIR 2?](#)

- [Adobe AIR Team Blog](#)
- [Exploring the new file capabilities in Adobe AIR 2](#)
- [See what's new in AIR 2 - Features list](#)
- ["What's new in FP10.1 and AIR2" slides and source files](#)

- [Adobe AIR Team Blog](#)
- [Exploring the new file capabilities in Adobe AIR 2](#)
- ["What's new in FP10.1 and AIR2" slides and source files](#)

Text Layout Framework

The Text Layout Framework is a library built on top of the low level Flash Text Engine APIs introduced in Flash Player 10. This framework introduces print quality support for text in Flash Player/AIR (including support for right-to-left languages).

On top of this framework, Spark introduces three new "primitive" text components (Label, RichText, and RichEditableText) and two skinnable text components (TextInput and TextArea).

These components support:
- 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)

Working with Fonts

When you use device fonts, you avoid the increase in SWF size that goes along with embedding the fonts. However, it is a good idea to specify at the end of the font list a generic device font. For example, if Helvetica is not available on the device, then another San Serif font can be used. Here are the values to choose from: `_sans`, `_serif`, `_typewriter`. You can declare the font like this:

```
.myClass {
        fontFamily: Arial, Helvetica, "_sans";
}
```

If you choose to embed fonts—supported file types include TrueType fonts (.ttf), OpenType fonts (.otf), TrueType Collections (.ttc), Mac Data Fork Fonts (.dfont), and Mac Resource Fork TrueType Suitcases (which do not have a file extension) —you'll realize the following benefits:

- The client environment does not need the font to be installed.
- Embedded fonts can be rotated and faded.
- Embedded fonts are anti-aliased, which means that their edges are smoothed for better readability. This is especially apparent when the text size is large.
- Embedded fonts provide smoother playback when zooming.
- Text appears exactly as you expect.
- You can use the advanced anti-aliasing information that provides clear, high-quality text rendering in SWF files. Using advanced anti-aliasing greatly improves the readability of text, particularly when it is rendered at smaller font sizes.

You embed a font using either a font-face declaration in CSS or Embed metadata. If you want to limit the size of the SWF, use character ranges and include only the chars you'll use for a specific font-face.

For more information, see [Using Flex 4: Fonts](#).

Working with Flex 3 Components and Flex 4 Components in the Same Project

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

Migrating Flex 3 Projects to Flex 4

If you have a Flex 3 project that needs some of the new functionality in Flex 4, then you may decide to migrate the project from Flex 3 to Flex 4. This can (and should) be a painless process. Here are the main steps you have to take:

1. Change the namespace from the MXML 2006 namespace to the new set of namespaces:
   ```
   xmlns:fx="http://ns.adobe.com/mxml/2009"
   xmlns:mx="library://ns.adobe.com/flex/mx"
   xmlns:s="library://ns.adobe.com/flex/spark"
   ```
2. If your application has non-UI components defined in MXML (such as RemoteObject, HTTPService, and so on), move them inside an `<fx:Declarations>` tag.
3. If you use CSS, you have to add a namespace for type selectors (Button, Panel, and so on); for example:
   ```
   <fx:Style>
   @namespace s "library://ns.adobe.com/flex/spark";
   @namespace mx "library://ns.adobe.com/flex/mx";

   s|Button {
       fontSize:16;
   }
   mx|Button {
       color:red;
   }
   </fx:Style>
   ```
4. Target Flash Player 10 instead of Flash Player 9.
5. If you use states, switch from the Flex 3 States model to the Flex 4 States model.

For more information, see the following resources:
- [Transitioning an application from Flex 3 to Flex 4](#)
- [Adobe Flex 4Features and Migration Guide](#)
- [Moving existing Flex projects from Flex Builder 3 to Flash Builder 4](#)

New Features in Flash Builder 4

Flash Builder 4 introduces a number of improvements that make developing Flex application even faster than before, including:

- Support for states in Code view
- Improved debugger: use conditional breakpoints, the go to line command, watchpoints, and expression evaluations
- Improved profiler: see if an object will be garbage collected
- Call hierarchy: display all the callers of the selected variable, function/method, class, or interface
- Code generators: automatically generate event handlers, getters, and setters
- Data Centric Development wizards: generate the client service wrapper and data types based on the server services, with support for Java, ColdFusion, PHP, Web Services, and REST Services.
- Network monitor and test operation views
- Refactoring: move and rename code
- ASDoc content display: hover over code or use code hinting to display ASDoc information. Select code to display information in ASDoc panel.
- Code indentation, including support for copy and paste
- Command line builds: synchronize individual build settings with a nightly build environment
- Unit testing: generate and edit reusable FlexUnit tests that can be run from scripts or directly within Flash Builder. There is automation support for Spark components and Adobe AIR applications.

Additional Resources

Among the best places to learn about Flex are the [Flex Developer Center](#) on Adobe Developer Connection and [Adobe TV](#), where you can find all the recordings from Adobe MAX 2008 and later. Finally, be sure to install Tour de Flex. This is an AIR application that offers tons of code examples for Flex 3 and Flex 4, including third-party components. It is updated regularly.

Online documentation on Flex 4 and Flash Builder 4:
http://help.adobe.com/en_US/flex/using/WSa7fd4845e77b4e67406b2ad31276e10ce3d-8000.html

AdobeTV:
http://tv.adobe.com

Adobe Developer Connection:
http://www.adobe.com/devnet/flex

Tour de Flex:
http://www.adobe.com/devnet/flex/tourdeflex/

What's new in Flex 4:
http://www.adobe.com/devnet/flex/articles/flex4sdk_whatsnew.html

Differences between Flex 3 and Flex 4:
http://www.adobe.com/devnet/flex/articles/flex3and4_differences.html

A brief overview of the Spark architecture and component set:
http://www.adobe.com/devnet/flex/articles/flex4_sparkintro.html

What's new in Flash Builder 4:
http://andrewrshorten.wordpress.com/2010/03/23/whats-new-in-flash-builder-4-all-of-this/

Moving from Flex 3 to Flex 4:
http://www.adobe.com/devnet/flex/articles/flexbuilder3_to_flashbuilder4.html

Data-centric development with Flash Builder 4:
http://www.adobe.com/devnet/flex/articles/datacentric_development.html

New Effect Engine: http://www.adobe.com/devnet/flex/articles/flex4_effects_pt1.html
http://www.adobe.com/devnet/flex/articles/flex4_effects_pt2.html

Introducing Skinning in Flex 4:
http://www.adobe.com/devnet/flex/articles/flex4_skinning.html

Spark layouts with Flex 4:
http://www.adobe.com/devnet/flex/articles/spark_layouts.html