Developing Applications for
ADOBE® LIVECYCLE® MOSAIC 9.5
Contents

Chapter 1: Overview
Software requirements ................................................................. 1

Chapter 2: Creating Flex tiles
Creating a Flex project ............................................................... 3
Walkthrough creating a Flex tile .................................................. 4
Using the Alert control in module tiles ........................................ 6
Importing Flex tile samples into Flash Builder ............................... 6
Communicating between Flex tiles .............................................. 7
Using Runtime Shared Libraries (RSLs) ......................................... 10

Chapter 3: Creating HTML Tiles
Walkthrough creating an HTML/URL tile ..................................... 12
Walkthrough creating an HTML/JavaScript tile .............................. 13
Communicating between HTML tiles ......................................... 15

Chapter 4: Creating Catalogs
Adding metadata to a catalog ..................................................... 21
Adding categories to a catalog .................................................... 22
Adding assets to a catalog ........................................................ 23
Creating policy-protected catalogs ............................................. 28
Including Runtime Shared Libraries (RSLs) in a catalog .................. 28
Deploying a catalog ................................................................. 29

Chapter 5: Creating Applications
Creating an application .............................................................. 31
Using application layouts ......................................................... 38
Creating application styles and skins ......................................... 42
Creating default views and panels .............................................. 47
Setting when tiles are loaded .................................................... 48
Creating policy-protected applications ....................................... 50
Deploying the application ........................................................ 51

Chapter 6: Creating and Assigning Policies
Asset resource IDs ................................................................. 52
Assigning policies to an application .......................................... 55
Assigning policies to a catalog .................................................. 56
Deploying policies ................................................................. 58

Chapter 7: Creating Services
Interface library ................................................................. 59
Service library ................................................................. 60
Using services within a Flex tile ................................................. 61

Last updated 3/31/2011
# Contents

Adding services to a catalog ................................................................. 62  
Adding interfaces and services to an application .................................. 65  

**Chapter 8: Deploying assets to the server**
Create project folder structure .............................................................. 67  
Create a build.xml file ........................................................................ 68  
Deploying to a Mosaic server ................................................................. 70  
Removing applications, catalogs, and policies from the Mosaic server .......... 70  

**Chapter 9: Debugging**
Deploying the debug version of Mosaic server ...................................... 73  
Debugging Flex tiles ............................................................................. 73
Chapter 1: Overview

Welcome to Developing Applications for Adobe® LiveCycle® Mosaic 9.5. Mosaic is a tool for developing composite rich Internet applications and delivering intuitive, personalized workspaces to end users. This document provides detailed information about how to perform the following tasks:

- Create tiles
- Create catalogs of tiles and assets
- Create applications
- Create and assign policies to Mosaic assets
- Deploy catalogs, applications, and policies to the Mosaic server

For a detailed discussion of the Mosaic architecture, see the Mosaic technical guide available from www.adobe.com/go/learn_lcmosaic_docs_9.

Software requirements

Before you begin, ensure that you have the required software installed.

Developing tiles, catalogs, and applications
To create Mosaic tiles, catalogs, and applications, ensure that the following software is installed:

- Adobe Flash® Builder™ 4 Standard, Flash Builder 4 Professional, or Flash Builder 4 Plug-in for Eclipse
- Adobe Flex SDK 3.3 or later (does not support module tile development or custom skinning)
- Adobe Flex SDK 4.1.0, build 16076
- Java™ 1.6
- Apache Ant 1.7 or later (available from Apache)
- Mosaic installation media

You can download the Flex SDK at opensource.adobe.com.

For a list of the Flash Builder system requirements, see www.adobe.com/products/flex/systemreqs, and for Mosaic system requirements, see LiveCycle Mosaic 9.5 Getting Started.

Note: The term Flash Builder in this document refers to all versions of Flash Builder. Previously, Flash Builder was named Flex Builder.

Viewing applications
To access Mosaic applications using the desktop client, ensure that the following software is installed:

- Adobe AIR® 2.0.2 or later (available from Download AIR)

To access Mosaic applications using a web browser, users need one of the following browser applications and a compatible version of Adobe Flash®:

- Internet Explorer 7 or later
- Firefox 3.5.4 or later (for Windows), Firefox 3.6 or later (for Mac)
Overview

- Adobe Flash Player 10.1 or later

For a complete list of Mosaic system requirements, see LiveCycle Mosaic 9.5 Getting Started.

More Help topics
“Creating Flex tiles” on page 3
“Creating HTML Tiles” on page 12
“Creating Catalogs” on page 21
“Creating Applications” on page 30
“Creating and Assigning Policies” on page 52
“Creating Services” on page 59
“Deploying assets to the server” on page 67
“Debugging” on page 73
Chapter 2: Creating Flex tiles

Mosaic Flex tiles are derived from applications and modules built using Flex. Once ready, you add the tiles to catalogs to make the tiles available to Mosaic applications. The Mosaic applications reference catalogs as well as individual tiles to build the final application structure.

You can create tiles that communicate with each other through shared variables that are passed through a context. Tiles can also communicate using a publish-and-subscribe messaging model, enabled using either ActionScript or JavaScript.

Creating a Flex project

Before you begin, ensure that you have correctly configured Flash Builder with the Mosaic SWC files. For more information, see "Installing and Configuring the Development Environment" in LiveCycle Mosaic 9.5 Getting Started.

Create a Flex project:

1. In Flash Builder, click File > New > Flex Project.

   Note: While a Flex tile can contain a mixture of ActionScript and MXML, the default file for launching your tile must be MXML.

2. In the Project Name field, type a unique name for your project. For example, SampleTile.

3. For Application Type, ensure that Web is selected, and for Flex SDK Version, ensure that you select 3.4.0 or later (4.1.0 or later recommended). Click Finish.

   (Optional) If you copied the Mosaic SWC files from the install location to the frameworks folder of your Flex SDK, skip to step 10. Otherwise, continue to step 4.

4. In the Flex Navigator view, right-click the project, and then click Properties.

5. Click Flex Build Path.

6. On the Library Path tab, click Add SWC.

7. In the Add SWC dialog box, browse to the location of the mosaic-tile.swc Flex library, by default [Mosaic install directory]/sdks/flex/sdks/[Flex SDK directory]/frameworks/lib. Select the SWC file, and then click OK.

8. If you want to create a tile with custom skinning, repeat step 7 and add the mosaic-skin.swc Flex library, located by default in [Mosaic install directory]/sdks/flex/sdks/4.1.0/frameworks/lib.

9. Click OK.

10. In the Flex Navigator view, right-click the project, and then click Properties.

11. Click Flex Compiler.

12. Add the following to the Additional Compiler Arguments field:

    -includes=mx.managers.systemClasses.MarshallingSupport -static-link-runtime-shared-libraries=true

13. Click OK.
Walkthrough creating a Flex tile

Once you have created successfully created a Flex project for your tile, you can create tiles using MXML and ActionScript.

Create a Flex tile:
1. Ensure that the MXML editor displays the MXML file for your project.
2. In the MXML editor, make the following changes:
   - Depending on the type of tile you want to build, change the root tag to one of the following:
     - From `mx:Application` to `mc:Tile` (for a tile based on an application)
     - From `mx:Module` to `mc:ModuleTile` (for a tile based on a module)
   - Add the following namespace, `mc="com.adobe.mosaic.core.*"`, to the root MXML element.

The following examples show the modified MXML in Source mode within Flash Builder, for a tile and a module tile created using Flex SDK 4.1.0:

```xml
<?xml version="1.0" encoding="utf-8"?>
<mc:Tile layout="absolute"
   xmlns:s="library://ns.adobe.com/flex/spark"
   xmlns:mx="library://ns.adobe.com/flex/mx"
   xmlns:mc="com.adobe.mosaic.core.*"
/>

<?xml version="1.0" encoding="utf-8"?>
<mc:ModuleTile layout="absolute"
   xmlns:s="library://ns.adobe.com/flex/spark"
   xmlns:mx="library://ns.adobe.com/flex/mx"
   xmlns:mc="com.adobe.mosaic.core.*"
/>
```

Now you can add MXML and ActionScript to create the desired functionality for your tile.

Adding a Flex tile to a catalog

Once you have created a tile, add it to a catalog to make the tile available to Mosaic applications. For more information on adding Flex tiles to a catalog file, see “Adding assets to a catalog” on page 23.

Mosaic menu tile example

The following MXML provides an example of a tile based on a Flex module. The MenuTile sample is included in the samples\clientDashboard\tiles folder where Mosaic is installed. This tile creates a reusable menu of options that can be shared across Mosaic applications.
<?xml version="1.0" encoding="utf-8"?>
xmlns:s="library://ns.adobe.com/flex/spark"
xmlns:mx="library://ns.adobe.com/flex/mx"
xmlns:mc="com.adobe.mosaic.core.*"
layout="absolute"
backgroundAlpha="0"
height="23"
width="100%">
  <s:Button id="menuButton" label="Add Tiles" click="onAddTiles()"
    skinClass="MenuButtonSkin"/>

  <fx:Script>
    <![CDATA[
      import com.adobe.mosaic.om.interfaces.ICatalog;
      import com.adobe.mosaic.om.interfaces.ITile;
      import com.adobe.mosaic.om.interfaces.IPanel;
      import com.adobe.mosaic.om.interfaces.IView;
      import mx.controls.Alert;
      import mx.controls.Menu;
      import mx.events.MenuEvent;
      
      [Bindable] private var menuData:XML =
        <list>
          <menuitem label="OpenCases" data="OpenCases"/>
          <menuitem label="ClientInfo" data="ClientInfo"/>
          <menuitem label="BugTracker" data="BugTracker"/>
          <menuitem label="IssueStatistics" data="IssueStatistics"/>
        </list>;
      
      private function onAddTiles() : void {
        var menu:Menu = Menu.createMenu(this,menuData,false);
        menu.labelField = "@label";
        menu.addEventListener(MenuEvent.ITEM_CLICK, onMenuClick);
        var p:Point = new Point( menuButton.x, menuButton.y+menuButton.height );
        p = localToGlobal(p);
        menu.show( p.x, p.y );
      }
    ]]>
  </fx:Script>
</mc:ModuleTile>
Using the Alert control in module tiles

The Flex Alert control is a pop-up dialog box that can contain a message, a title, buttons, and an icon. By default, the alert is displayed in relation to the main window of its parent. For a Mosaic application, the parent's main window is the login window, which is hidden unless it is needed.

To use the Flex Alert class in a module tile, specify the parent object of the alert. This configuration ensures that the alert is displayed when the tile is running in the Mosaic desktop client. For example:

```
Alert.show("My message", "Alert title", Alert.OK, parentDocument as Sprite)
```

Using `parentDocument as Sprite` ensures that the target window is the Mosaic runtime window where the Mosaic application is displayed.

Importing Flex tile samples into Flash Builder

To assist you in creating new Flex tiles, you can import the Flex projects for all sample tiles included with Mosaic into Flash Builder. The sample tiles are located in the `samples\general\tiles` and `samples\max2009\tiles` folders where Mosaic is installed.

Import a Mosaic sample Flex tile project into Flash Builder:

1. Start Flash Builder.
2. Click File > Import > Flex Project.
3. Select Project folder, and specify the folder location of a sample tile. For example, `C:\Mosaic\samples\general\tiles\TileGame`.
4. Click Finish.
Communicating between Flex tiles

You can use two forms of communication to pass values and content between tiles:

- **contexts**
- **messaging**

For most situations, using contexts to pass attribute values between tiles is more desirable, because the attribute values persist on the server. This persistence allows tiles added to the application after the attribute value is set to retrieve the value.

Messaging allows you to submit a payload, which can be any serializable object, to all event listeners set to listen for the event. The payload does not persist on the server, but is ideal for passing single instance data across tiles.

*Note: Messages are broadcast throughout a Mosaic application, and are not restricted to particular views.*

Using contexts

All tiles within a Mosaic application exist within two contexts:

- **application**
- **view**

The application context represents all tiles throughout a Mosaic application, regardless of the view in which the tiles appear. The view context applies only to tiles within the same view. Tiles use these contexts to set attribute values and pass those values between tiles.

Both the `Tile` and `ModuleTile` classes, the base classes for Mosaic tiles, define a protected property `mosaicApp`, which is of type `IApplication`. In this way, you can use `mosaicApp` to retrieve and set information about a Mosaic application. The following ActionScript demonstrates how to reference the application and view level contexts from a tile to set or get an attribute value:

```actionscript
/* application */
mosaicApp.context.setAttribute(name, value);
mosaicApp.context.getAttribute(name);

/* view */
this.parentView.context.setAttribute(name, value);
this.parentView.context.getAttribute(name);
```

When setting a context using ActionScript objects and interfaces, note that only primitive ActionScript objects and interfaces defined in an interface library are valid. The interfaces must be entirely independent of the Flex SDK as they are shared among tiles that could be compiled with different Flex SDK versions. The supported ActionScript primitive types are:

- Array
- Date
- int
- Number
- Object
- String
- uint
Creating Flex tiles

- XML
- XMLList

**Note:** This list only applies to tiles. ModuleTiles can use **ant** Flex SDK class because ModuleTiles are not loaded into individual ApplicationDomains.

The following example demonstrates an example using the global context where the name and value of the attributes are linked to objects in the tile itself.

```xml
        xmlns:mc="com.adobe.mosaic.core.*"
        width="310"
        height="327"
        layout="absolute">

<mx:Script>
<![CDATA[
    private function setContext():void{
        mosaicApp.context.setAttribute(txtName.text,txtValue.text);
    }
    private function getContext():void{
        var s:String = mosaicApp.context.getAttribute(txtName.text);
        txtOutput.text += "Global Context Attribute: " + txtName.text + " = " + s + "\n";
    }
]]>
</mx:Script>

<mx:Button x="10" y="102" label="Get Global Context" click="{getContext()}"/>
<mx:TextArea id="txtOutput" height="185" bottom="10" right="10" left="10" editable="false"/>
<mx:Button x="10" y="72" label="Set Global Context" click="{setContext()}"/>
<mx:TextInput x="118" y="10" id="txtName"/>
<mx:Label x="10" y="12" text="Context Attribute"/>
<mx:TextInput x="118" y="38" id="txtValue"/>
<mx:Label x="10" y="40" text="Context Value"/>
</mc:Tile>
```

You can view a working example of contexts in the Basic example included with Mosaic. You can view the source for the sample in the samples\general\tiles\Basic folder where Mosaic is installed. For more information on the Mosaic ActionScript API for contexts, see [LiveCycle Mosaic 9.5 ActionScript API Reference](#).

**Using messaging**

Apart from using contexts, the other method for communicating between tiles is through the `com.adobe.mosaic.core.events.Message` class. Conceptually, you create a system of broadcasting from tiles and then add event listeners to allow tiles to receive broadcast information. Messages are sent out across an entire Mosaic application.

For example, consider the `MessageSender` component from the Basic sample application included with Mosaic:
Creating Flex tiles

In this example, the sendMessage method broadcasts a message to all other tiles in an application with the values of the txtNamespace, txtName, and txtPayload objects. However, for the broadcast to be of value, other tiles in the application must have an event listener added that receives the broadcast. Continuing this example, the MessageReceiver component from the Basic sample application included with Mosaic adds appropriate event listeners to tiles in the Basic application:
In this example, broadcast messages from the MessageSender object are ignored until the addListener method is called through a user clicking the Button object.

Through this mechanism of broadcasting and listening, you can create inter-tile communication that gives users a much more dynamic experience.

For more information on the Mosaic ActionScript APIs, see LiveCycle Mosaic 9.5 ActionScript API Reference.

Using Runtime Shared Libraries (RSLs)

One way to reduce the size of a Mosaic application SWF file is by externalizing shared assets into stand-alone files. These files can then be downloaded separately and cached on the client. These shared assets are available to any number of applications at runtime, but are transferred only once to the client. These shared files are known as Runtime Shared Libraries or RSLs. For more background information on RSLs, see Using Flex 4.

Mosaic supports RSLs with respect to Flex tiles by including the RSLs inside a catalog. For more information, see “Including Runtime Shared Libraries (RSLs) in a catalog” on page 28.

Note: Always compile Flex tiles with absolute RSL URLs. This step is required for the tile to function correctly in the Mosaic desktop client.

More Help topics
“Overview” on page 1
“Creating HTML Tiles” on page 12
“Creating Catalogs” on page 21
“Creating Applications” on page 30
“Creating and Assigning Policies” on page 52
“Creating Services” on page 59
“Deploying assets to the server” on page 67
“Debugging” on page 73
Chapter 3: Creating HTML Tiles

HTML tiles allow you to do the following in a Mosaic application:

- Integrate web pages you do not own or control.
- Extend HTML functionality of web pages you control through the Mosaic JavaScript libraries.

You can create two kinds of HTML tiles: URL references to existing web pages and HTML pages extended with JavaScript.

Walkthrough creating an HTML/URL tile

Creating URL tiles involves adding tile:TileClass elements to a catalog file.

Create a URL tile:
1. Using a text or XML editor, open a descriptor.xml file that describes a Mosaic catalog.
2. Add a tile:TileClass child element to the tile:TileClassList element. Ensure that the uri attribute for the tile:Content child element is a web-based URL, and that the contentType attribute is set to text/html.
3. Save the catalog. The tile is now ready to reference in an application.

Note: Deploy the catalog to a Mosaic server to make the tiles available to applications. For more information on deploying, see "Deploying assets to the server" on page 67.

For example, the following tile:TileClass entry represents the Google Finance tile available in the Brokerage sample application included with Mosaic.

```xml
<tile:TileClass catalog="sampleCatalog"
  label="Google Finance"
  name="GoogleFinance"
  maxHeight="1200"
  maxWidth="1200"
  minHeight="400"
  minWidth="400"
  maintainAspectRatio="true"
  initialHeight="400"
  initialWidth="400">
  <crx:Metadata>
    <crx:Description>Google Finance</crx:Description>
    <crx:Category>Component</crx:Category>
    <crx:Tag>component</crx:Tag>
    <crx:Tag>generic</crx:Tag>
  </crx:Metadata>

  <tile:Content contentType="text/html"
    uri="http://www.google.com/finance?client=ob&amp;q=${application.symbol}"/>
</tile:TileClass>
```
Note: Do not use a HTML/URL tile as a shortcut for displaying a Flex tile. Specifying a URL to an application created using Flex can result in the tile displaying incorrectly. The tile does not resize content to fit the area designated to tile. In addition, in some cases, portions of the referenced application are not visible.

More Help topics
“Adding tiles to a panel” on page 36

Walkthrough creating an HTML/JavaScript tile

In addition to HTML tiles that reference a URL, you can create HTML tiles that use JavaScript for more dynamic experiences. Mosaic also includes a JavaScript API that provides functions related to tile features. Include the mosaicBridge.js library with your JavaScript tile to have access to the functions provided as part of the Mosaic JavaScript API. The mosaicBridge.js library is located in the sdk/javascript folder of the Mosaic installation file set.

For more information on the Mosaic JavaScript APIs, see LiveCycle Mosaic 9.5 JavaScript API Reference.

Create an HTML/JavaScript tile:
1. Create an HTML file.
2. Copy the mosaicBridge.js library from [Mosaic installation directory]/sdk/javascript into the same directory as your HTML file. Ensure to include both files in the WAR file that you deploy to the application server.
3. Add a script element for the JavaScript you want to add to your HTML file, as well as any additional HTML.

The following example shows an example of an HTML tile with a single JavaScript function that sends a message using the sendMessage function:

```html
<html>
<head>
<script type="text/javascript" src="mosaicBridge.js"></script>
<script>
    function sendMessage(){
        var o = new Object();
        o.nameSpace = "myNameSpace"
        o.name = "Name";
        o.payload = "This is a payload";
        mosaicApp.sendMessage(o);
    }
</script>
</head>
<body>
<h3>Example</h3>
<input type="button" value="Send Message" onclick="sendMessage()" />
</body>
</html>
```

Now you can add HTML and JavaScript to create the desired functionality for your tile.

Adding an HTML tile to a catalog

Once you have created a tile, add it to a catalog to make the tile available to Mosaic applications. For more information on adding Flex tiles to a catalog file, see “Adding assets to a catalog” on page 23.
Configuring applications to work with HTML plug-in content

An HTML tile can include plug-in content, such as a Java applet or a PDF document. This plug-in content can hide all or part of the Mosaic application interface. For example, if Flash Player is running in the HTML tile, it is displayed on top of any Add View dialog boxes the Mosaic application displays. This problem happens only when you use the application in a web browser and not with the Mosaic desktop client.

Use the hasPlugin attribute to hide HTML tiles whenever the Mosaic application displays a pop-up dialog box or menu. The HTML tile is displayed behind the application background, which prevents the pop-up content from being obstructed.

The hasPlugin attribute is available for TileClass elements (when you create catalogs). It can have one of the following values:

- **none**  Specifies that the HTML tiles do not have plug-in content.
- **other**  Hides an HTML tile when the application is running in a web browser and displays pop-up content. It does not change the behavior of a Flex tile or applications running in the desktop client. This value is the default, and the behavior when you are using an earlier version of Mosaic or do not specify the hasPlugin attribute.
- **AdobeReader**  The application adjusts the display of HTML tiles in the same way as with **other**. However, by identifying the plug-in as Adobe Reader, the application design will be able to take advantage of future functionality.
- **AdobeFlashPlayer**  The application adjusts the display of HTML tiles in the same way as with **other**. However, by identifying the plug-in as Adobe Flash Player, the application design will be able to take advantage of future functionality.

The following catalog XML illustrates how you can use the hasPlugin attribute:
The behavior for not using the attribute and using other is the same. As a result, the application hides both the SimpleContext1 and SimpleContext3 HTML tiles when it displays pop-up content. The tile SimpleContext2, which uses a value of none, remains visible when pop-up content is displayed.

**Communicating between HTML tiles**

HTML tiles can broadcast to other tiles and listen to broadcasts from other tiles to create a more dynamic user experience. This feature is similar to “Communicating between Flex tiles” on page 7.

You can use two forms of communication to pass values and content between tiles:

- contexts
- messaging
For most situations, using contexts to pass attribute values between tiles is more desirable. When you use contexts, the attribute values persist on the server when the user saves a view. This persistence allows tiles added to the application after the attribute value is set to retrieve the value.

Messaging allows you to submit a payload, which can be any serializable object, to all event listeners set to listen for the event. The payload does not persist on the server, but is ideal for passing single instance data across tiles.

*Note*: Messages are broadcast throughout a Mosaic application, and are not restricted to particular views.

### Using contexts

All tiles within a Mosaic application exist within two contexts:

- application
- view

The application context represents all tiles throughout a Mosaic application, regardless of the view in which the tiles appear. The view context applies only to tiles within the same view. Tiles use these contexts to set attribute values and pass those values between tiles.

You can use the `mosaicApp` variable to retrieve and set information about a Mosaic application, and `parentView` variable to retrieve and set information about a view. The following JavaScript demonstrates how to reference the application and view level contexts from a tile to set or get an attribute value:

```javascript
/* application */
function setContext()
    var context = mosaicApp.getContext();
    context.setAttribute(name, value);
}

/* view */
function setContext()
    var context = parentView.getContext();
    context.setAttribute(name, value);
}
```

When setting a context that references ActionScript objects and interfaces, note that only primitive ActionScript objects and interfaces defined in an interface library are valid. In JavaScript, these typed interfaces come across as object types and not with their particular "Class" definitions. Therefore, treat these objects as dynamic types. The interfaces must be entirely independent of the Flex SDK as they are shared among tiles that could be compiled with different Flex SDK versions. The supported ActionScript primitive types are:

- Array
- Date
- int
- Number
- Object
- String
- uint
- XML
- XMLList
Note: This list only applies to tiles. ModuleTiles can use any Flex SDK class because ModuleTiles are not loaded into individual ApplicationDomains.

The following example demonstrates an example using the application context where the name and value of the attributes are linked to objects in the tile itself.

```html
<html>
<head>
    <script type="text/javascript" src="mosaicBridge.js"></script>
    <script>
        function appendLine(info){
            document.getElementById('txtOutput').value += info + "\n";
        }
        function setContext(){
            mosaicApp.getContext().setAttribute(document.contextInfo.name.value,document.contextInfo.value.value);
        }
        function getContext(){
            var o = mosaicApp.getContext().getAttribute(document.contextInfo.name.value);
            appendLine("Attribute: " + document.contextInfo.name.value + " = " + o);
        }
        function addWatcher(){
            mosaicApp.getContext().addAttributeWatcher(document.contextInfo.name.value,onWatcher);
        }
        function onWatcher(event){
            appendLine("Watcher: " + event.source + " " + event.oldValue + " -> " + event.newValue);
        }
        function sendMessage(){
            var o = new Object();
            o.nameSpace = document.messageInfo.namespace.value;
            o.name = document.messageInfo.name.value;
            o.payload = document.messageInfo.payload.value;
            mosaicApp.sendMessage(o);
        }
        function addMessageListener(){
            mosaicApp.addMessageListener(document.messageInfo.namespace.value,
            document.messageInfo.name.value, onMessage);
        }
        function onMessage(message){
            appendLine("Message: " + message.nameSpace + ":" + message.name + " = " + message.payload);
        }
    </script>
</head>
<body>
```
In this example, a user clicks the submit button to add a listener to the tile. Users click the button object to broadcast messages to other objects on this tile, as well as other tiles in the application.

You can view a working example of contexts in the BasicHTML example included with Mosaic. You can view the source for the sample in the samples\general\tiles\BasicHTML folder where Mosaic is installed. For more information on the Mosaic JavaScript API for contexts, see LiveCycle Mosaic 9.5 JavaScript API Reference.

### Setting when content from contexts is refreshed

The contextWatcherPolicy attribute allows you to stop an HTML tile from refreshing content provided by a context if the tile is not currently displayed. This configuration can improve the performance of your Mosaic application.

The contextWatcherPolicy attribute is available for TileClass elements (when you create catalogs). It can have one of the following values:

- **alwaysProcess** - Whenever a context changes, the application refreshes the HTML tile with the new values or content. This value is the default and the behavior when you are using an earlier version of the Mosaic client.
- **ignoreUnlessDisplayed** - The application refreshes the HTML tile content provided by contexts only if the tile is currently displayed.

In this example, contextWatcherPolicy is included in the TileClass element for the GoogleFinance sample application:

```xml
<tile:TileClass maxHeight="1200" maxWidth="1200" minHeight="400" minWidth="400" maintainAspectRatio="true" initialHeight="400" initialWidth="400" label="GoogleFinance" name="GoogleFinance" catalog="sampleCatalog" contextWatcherPolicy="ignoreUnlessDisplayed">
    <crx:Metadata>
        <crx:Description>Google Finance</crx:Description>
        <crx:Category>Component</crx:Category>
        <crx:Tag>component</crx:Tag>
        <crx:Tag>generic</crx:Tag>
    </crx:Metadata>
    <tile:Content contentType="text/html" url="http://www.google.com/finance?client=ob&amp;q=${application.symbol}"/>
</tile:TileClass>
```
Using messaging

You can pass messages between tiles using the `sendMessage` function. Conceptually, you create a system of broadcasting from tiles and then add event listeners to allow tiles to receive broadcast information. Messages are sent out across an entire Mosaic application.

For example, consider the following example from the BasicHTML sample application included with Mosaic:

```html
<html>
<head>
<script type="text/javascript" src="mosaicBridge.js"></script>
<script>

function appendLine(info){
    document.getElementById('txtOutput').value += info + "\n";
}

function setContext(){
    mosaicApp.getContext().setAttribute(document.contextInfo.name.value,document.contextInfo.value.value);
}

function getContext(){
    var o = mosaicApp.getContext().getAttribute(document.contextInfo.name.value);
    appendLine("Attribute: " + document.contextInfo.name.value + " = " + o);
}

function addWatcher(){
    mosaicApp.getContext().addAttributeWatcher(document.contextInfo.name.value,onWatcher);
}

function onWatcher(event){
    appendLine("Watcher: " + event.source + " " + event.oldValue + " -> " + event.newValue);
}

function sendMessage(){
    var o = new Object();
    o.nameSpace = document.messageInfo.namespace.value;
    o.name = document.messageInfo.name.value;
    o.payload = document.messageInfo.payload.value;
    mosaicApp.sendMessage(o);
}

function addMessageListener(){
    mosaicApp.addMessageListener(document.messageInfo.namespace.value,
    document.messageInfo.name.value, onMessage);
}

function onMessage(message){
    appendLine("Message: " + message.nameSpace + ":" + message.name + " = " + message.payload);
}

</script>
</head>
<body>
```
Through this mechanism of broadcasting and listening, you can create communication between tiles that gives users a much more dynamic experience.

**Note:** You cannot pass array, Boolean, or date objects between HTML tiles, or between HTML tiles and Adobe Flex tiles. The JavaScript API for Mosaic does not currently support the transmission of these objects as payloads for messages between tiles.

For more information on the Mosaic JavaScript APIs, see [LiveCycle Mosaic 9.5 JavaScript API Reference](#).

**More Help topics**
- “Overview” on page 1
- “Creating Flex tiles” on page 3
- “Creating Catalogs” on page 21
- “Creating Applications” on page 30
- “Creating and Assigning Policies” on page 52
- “Creating Services” on page 59
- “Deploying assets to the server” on page 67
- “Debugging” on page 73
Chapter 4: Creating Catalogs

A catalog is an XML file that defines a set of assets that are available to Mosaic applications on the server. Mosaic assets include tiles, style sheets, runtime shared libraries (RSLs), services, resources, and applications.

In general, when you create a catalog you follow these steps:

1. Create the following folder structure for the tiles and other assets you want to deploy to the server. It is not necessary to create folders for asset types that you are not using.
   - [ProjectName]/catalogs
   - [ProjectName]/catalogs\Interfaces
   - [ProjectName]/catalogs\Resources
   - [ProjectName]/catalogs\RSLs
   - [ProjectName]/catalogs\Services
   - [ProjectName]/catalogs\Stylesheets
   - [ProjectName]/catalogs\Tiles
   - [ProjectName]/catalogs\Tiles\[tilename] (repeat for each tile in the catalog)

2. Create an XML file named descriptor.xml in a file editor.

3. Add tiles, categories, and metadata to the file in XML.

4. Save the descriptor.xml file to the [ProjectName]/catalogs folder.

5. Deploy the catalog, and the associated tiles, to the Mosaic server. For more information on deploying, see “Deploying assets to the server” on page 57.

The following example illustrates an empty catalog named sample_catalog. The value of the name attribute corresponds to the value of the name attribute in the mosaic-catalog Ant task used to deploy the catalog. Similarly, the value of the name is used as the value of the catalog attribute of TileReference elements in an application. The value of the name attribute must be unique across all catalogs available on the Mosaic server.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog:Catalog name="[mosaic-catalog Ant task name]" label="Sample Catalog"
xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/"
xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/"/>
</catalog:Catalog>
```

Note: The example above illustrates the Mosaic namespaces and prefixes used in all Mosaic sample catalog files.

Adding metadata to a catalog

You can add metadata at both the catalog and asset levels in using the Metadata element. However, provide at least a description, using the Description element, for both catalogs and tiles.

The following example extends the sample_catalog example by adding metadata that describes the HTML tile and the catalog.
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog:Catalog name="[mosaic-catalog Ant task name]" label="Sample Catalog"
 xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/">
 xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
 xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
 xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
 xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/">
 <crx:Metadata>
   <crx:Description>Sample Catalog</crx:Description>
 </crx:Metadata>

 <catalog:Categories>
   <catalog:Category name="HTML_Tiles"/>
 </catalog:Categories>

 <tile:TileClassList>
   <tile:TileClass
     maxHeight="1200"
     maxWidth="1200"
     minHeight="400"
     minWidth="400"
     maintainAspectRatio="true"
     initialHeight="400"
     initialWidth="400"
     label="AdobeDotCom"
     name="Adobe.com"
     catalog="sampleCatalog">
     <crx:Metadata>
       <crx:Description>Adobe.com</crx:Description>
       <crx:Category>HTML_Tiles</crx:Category>
       <crx:Tag>component</crx:Tag>
     </crx:Metadata>
     <tile:Content contentType="text/html" uri="http://www.adobe.com"/>
   </tile:TileClass>
 </tile:TileClassList>
</catalog:Catalog>

For more information on the child elements and attributes of the Metadata element, see LiveCycle Mosaic 9.5 XML Schema Reference.

Adding categories to a catalog

To assist organizing your tiles, you can create categories within a catalog. Then, you associate tiles with a particular category. For example, you could create categories to store tiles for each Mosaic application separately, or create a category explicitly for HTML tiles.

The following example extends the sample_catalog example by adding a single category for HTML tiles and adds a single HTML tile associated with the category.
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog:Catalog name="sample_catalog" label="Sample Catalog"
 xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/"
 xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
 xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
 xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
 xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/>

<catalog:Categories>
  <catalog:Category name="HTML_Tiles"/>
</catalog:Categories>
</catalog:Catalog>

Tiles can be associated with multiple categories, or none. For more information on the attributes of the Category element, see LiveCycle Mosaic 9.5 XML Schema Reference.

Adding assets to a catalog

You add Mosaic assets to a Mosaic catalog by adding their definition to the descriptor.xml file that defines the catalog. Mosaic assets include tiles, style sheets, services, and images, as well as panels, and views you want to reuse across applications.

Add a Flex tile to a catalog:
1  Open a catalog descriptor.xml file in a text or XML editor.
2  Add a new tile:TileClass element as a child of the tile:TileClassList element in the catalog, and specify the appropriate information for your tile.

   Note: The value of the name attribute of the tile:TileClass element is the filename of the tile excluding the filename extension.

The following example adds the TileTimer tile from the TileGame sample application to a catalog:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog:Catalog name="sample_catalog" label="Sample Catalog"
xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/"
xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/>

<tile:TileClassList>
  <tile:TileClass initialHeight="296"
    initialWidth="214"
    label="Tile Timer"
    name="TileTimer"
    <crx:Metadata>
      <crx:Description>Simple Tile Game</crx:Description>
      <crx:Category>Component</crx:Category>
      <crx:Tag>component</crx:Tag>
      <crx:Tag>menu</crx:Tag>
    </crx:Metadata>
    <tile:Content loadAs="default"
      contentType="application/x-shockwave-flash"
      uri="/mosaic/catalogs/sample_catalog/tiles/TileTimer/content"/>
  </tile:TileClass>
</tile:TileClassList>
</catalog:Catalog>

Because the TileTimer tile is based on an application built using Flex, for the tile:Content element, the value of loadAs is "default". For tiles based on Flex modules, set loadAs to "module".

Add an HTML tile to a catalog:
1 Open a catalog descriptor.xml file in a text or XML editor.
2 Add a new <tile:TileClass> element as a child of the <tile:TileClassList> element in the catalog. Then, specify the appropriate information for your tile.
   Note: HTML tiles do not have a corresponding file to deploy to the server. Assets that do not have a corresponding file to deploy to the server can use any value for the name attribute. However, this value must be unique among all assets of the same type within the catalog.

The following example extends the sample_catalog example by adding a single HTML tile that displays the Adobe website.
Add a panel to a catalog:
1 Open a catalog descriptor.xml file in a text or XML editor.
2 Add a new view:PanelClass element as a child of a view:PanelClassList element in the catalog, and specify the appropriate information for your tile.
Creating Catalogs

Add a view to a catalog:

1. Open a catalog descriptor.xml file in a text or XML editor.
2. Add a new view:ViewClass element as a child of a view:ViewClassList element in the catalog, and specify the appropriate information for your view.

The following example extends the sample_catalog example by adding several new views:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog:Catalog name="sample_catalog" label="Sample Catalog"
 xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/"
 xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
 xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
 xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
 xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/">
  <view:ViewClassList>
    <view:ViewClass name="Financial Services" label="Financial Services">
      <crx:Metadata>
        <crx:Description>A sample workspace, contains at least one panel.</crx:Description>
        <crx:Category>Finance</crx:Category>
        <crx:Tag>business</crx:Tag>
      </crx:Metadata>
      <crx:Context/>
      <view:Content>
        <view:View>
          <view:PanelReference name="Calculators" catalog="sample_catalog"/>
        </view:View>
      </view:Content>
    </view:ViewClass>
  </view:ViewClassList>
</catalog:Catalog>
```
Add a style sheet to a catalog:
1. Open a catalog descriptor.xml file in a text or XML editor.
2. Add a new view:StyleSheetClass element as a child of a view:StyleSheetClassList element in the catalog, and specify the appropriate information for your style sheet.

The following example extends the sample_catalog example by adding a single style sheet with customized navigation styles:
After you add a style sheet to a catalog, you can reference styles from the style sheet in applications. For more information, see “Default styles” on page 42.

More Help topics
LiveCycle Mosaic 9.5 XML Schema Reference
“Creating Services” on page 59

Creating policy-protected catalogs

To restrict access to Mosaic assets, you create policies that define which user roles are entitled to access the assets. Mosaic assets include applications, catalogs, tiles, style sheets, views, panels, and resources. You create roles and assign users to them using LiveCycle Administration Console.

For more information about creating policies and assigning them to applications, see “Creating and Assigning Policies” on page 52.

Including Runtime Shared Libraries (RSLs) in a catalog

There are two locations where you can include Flex tile RSLs inside a catalog:

• In the tile directory with the generated SWF file (for example, [ProjectName]/[CatalogName]/Tiles/[tilename]/)
• In the catalog RSLs folder (for example, [ProjectName]/[CatalogName]/RSLs)

You do not add references to the RSLs to the catalog descriptor.xml. The RSLs are included in the catalog archive by adding the files to the appropriate folder, described above.

*Note:* Always compile Flex tiles with absolute RSL URLs. This step is required for the tile to function correctly in the Mosaic desktop client.

## Deploying a catalog

Once you have created a catalog descriptor.xml file, deploy the catalog to the Mosaic server. After you deploy the catalog, the catalog and its associated assets are available to Mosaic applications.

For more information on deploying catalogs to the Mosaic server, see “Deploying assets to the server” on page 67.

**More Help topics**

“Overview” on page 1

“Creating Flex tiles” on page 3

“Creating HTML Tiles” on page 12

“Creating Applications” on page 30

“Creating and Assigning Policies” on page 52

“Creating Services” on page 59

“Deploying assets to the server” on page 67

“Debugging” on page 73
Chapter 5: Creating Applications

The basic function of a Mosaic application design is to organize, lay out, and create functionality for a set of component applications called tiles. The tiles you add to panels provide the user with interactive content, such as an application created using Flex or an HTML page. The following image illustrates the Brokerage sample application included with Mosaic:

A. Contacts tile (Flex)  B. Portfolio tile (Flex)  C. Stock Information tile (HTML)

To create a Mosaic application, you perform the following general steps:

1. Create an application
2. Add views to the application
3. Add panels to the views
4. Add tiles to the panels
5. Deploy the application

The shell is a container for one or more views. It can provide shared functionality such as a menu that allows users to add, save, and load customized views. For example, you can create one shell for employee self-service and another for a trading desktop.

Although the shell and panel add a layout and shared functionality to tiles, you create tiles independently from the application.
Mosaic provides a set of sample applications. You can use the sample application shells, views, and layouts as a starting point for your own applications. You can also develop your own application elements to give your applications a custom appearance.

For more information on creating tiles, see “Creating Flex tiles” on page 3 or “Creating HTML Tiles” on page 12, and for information on creating catalogs, see “Creating Catalogs” on page 21. For more information on the XML structure of the elements involved in creating applications, see LiveCycle Mosaic 9.5 XML Schema Reference.

Creating an application

The central component of a Mosaic application is a shell.

The shell provides the menus that allow a user to access and save Mosaic views. It also determines what appears to users and what they can do by:

- Displaying view elements that are related to the user's current activity, including customized tiles.
- Providing a menu that allows the user to add and save views and panels.

You select features and a layout for a shell when you design an application. For example, you can create a shell that includes a view organizer or a menu that allows users to add tiles to panels.

Create an application:

Create an application XML file in a text editor. The application XML file uses the following basic structure:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<app:Application name="Default" label="Default"
 xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
 xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
 xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
 xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/"
 xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/">
  <app:Shell name="Default" label="Default">
    <catalog:CatalogReference name="sample_catalog" uri="sample_catalog"/>
  </app:Shell>
</app:Application>
```

The application includes namespace references to the Mosaic namespaces, and an object hierarchy beginning with a Shell element that contains at least one CatalogReference object.

The Application element label attribute specifies the text to display in the title bar of the browser client. If this attribute is not used or has no value, the title bar displays the default content. For example, some web browsers display the current URL in the title bar.

Note: You can use the Default application included in the samples\general\applications folder where you installed Mosaic as a starting point for creating your own applications.

Adding a view

The views you define for an application are the default views that appear when users access the application. Depending on your application design, users can create custom views and save them to use later.

Views are child elements of the ViewManager element. The view manager determines the size of the views, and, optionally, can include a Layout child element to determine the structure of panels in the view.
You can add ViewManager elements to app:Shell elements or view:Panel elements.

The optional Organizer element adds a view organizer and menu that allow users to save and manage their views. The menu also includes an option that generates a URL for the current view. Users can use this URL to share the view with other Mosaic users who can access the application.

**Add a view to an application:**

❖ In an application XML file in a file editor, add a ViewManager element as a child element of the Application element. Then, add one or more View elements as children of the ViewManager element:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app:Application label="Basic" name="Basic"
    xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/"
    xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
    xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
    xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
    xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/>

<crx:Metadata>
    <crx:Description>This application demonstrates inter-tile communication using the Mosaic Tile SDK.</crx:Description>
</crx:Metadata>
<app:Shell label="Basic" name="Basic">
    <catalog:CatalogReference uri="sample_catalog" name="sample_catalog"/>
    <view:ViewManager height="100%" width="100%">
        <view:View height="100%" width="100%" label="Tile Communication">
            <view:Panel label="Simple Context" styleName="PanelInShellStyle">
                <view:Layout numColumns="2" name="DynamicColumnLayout"/>
                <tile:TileReference label="SetAndGetContext" name="SetAndGetContext" catalog="sample_catalog" />
                <tile:TileReference label="ContextWatcher" name="ContextWatcher" catalog="sample_catalog" />
                <tile:TileReference label="SimpleContext" name="SimpleContext" catalog="sample_catalog" width="100%" height="50%"/>
            </view:Panel>
        </view:View>
    </view:ViewManager>
    <view:View height="100%" width="100%" label="Complex Communication">
        <view:Panel label="Complex Context">
            <view:Layout numColumns="2" name="DynamicColumnLayout"/>
            <tile:TileReference label="ComplexContextSetGet" name="ComplexContextSetGet" catalog="sample_catalog" />
        </view:Panel>
    </view:View>
</app:Shell>
```
Adding a view organizer
❖ In an application XML file in a file editor, add an Organizer element as a child element of a Shell or Panel element.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<app:Application name="ClientDashboard" label="Client Dashboard"
 xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
 xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
 xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
 xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/"
 xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/">
  <crx:Metadata>
    <crx:Description>This application shows how to use Views and Panels to construct a Mosaic application. The Tiles within this application demonstrate the Mosaic Tile SDK.</crx:Description>
  </crx:Metadata>
  <app:Shell name="ClientDashboard" label="Client Dashboard" styleName="BannerShellStyle">
    <view:Layout name="VerticalLayout" verticalGap="-2"/>
    <catalog:CatalogReference name="dashboard_catalog" uri="dashboard_catalog"/>
    <tile:TileReference height="25" label="Menu" name="MenuTile" catalog="dashboard_catalog"/>
    <view:Panel width="100%" height="100%" styleName="PanelInShellStyle">
      <view:Organizer width="290" height="100%"/>
      <view:ViewManager width="100%" height="100%"/>
```
Adding a panel to a view

You use panels to organize tiles in a view. For example, a panel is useful if the view contains multiple tiles that do not fit onto the screen, which requires the user to scroll the browser window. The default panel definition uses the FlowLayout layout, which arranges tiles top-to-bottom and left-to-right.

Add a panel to a view:
❖ In an application XML file in a file editor, add a view:Panel element as a child element of a view:View element:
Add a panel to a shell:
You can add view:Panel elements as direct children of app:Shell elements, that is, without using views. Panels added directly as children of app:Shell elements must explicitly set width and height attribute values:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<app:Application name="ClientDashboard" label="Client Dashboard"
xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/"
xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/">
  <crx:Metadata>
    <crx:Description>The Mosaic Application demonstrated at Adobe MAX 2009. This application shows how to use Views and Panels to construct a Mosaic application. The Tiles within this application demonstrate the Mosaic Tile SDK.</crx:Description>
  </crx:Metadata>
  <app:Shell name="ClientDashboard" label="Client Dashboard" styleName="BannerShellStyle">
    <view:Layout name="VerticalLayout" verticalGap="-2"/>
    <catalog:CatalogReference name="dashboard_catalog" uri="dashboard_catalog"/>
    <tile:TileReference height="25" label="Menu" name="MenuTile" catalog="dashboard_catalog"/>
    <view:Panel width="100%" height="100%" styleName="PanelInShellStyle">
      <view:Layout name="HorizontalLayout" horizontalGap="0"/>
      <view:Organizer width="290" height="100%"/>
      <view:ViewManager width="100%" height="100%"/>
    </view:Panel>
  </app:Shell>
</app:Application>
```
Adding tiles to a panel

Tiles are independently developed Flex or HTML applications, or URL references, that add interactive content to panels. You can add any tile included in the application’s catalog to a panel. The tile’s properties and panel layout determine the tile’s appearance and location. When you add a tile to a panel, you can also set additional properties. For example, you can specify whether users can delete the tile from the panel when they are customizing their views.

Add a tile to a panel:
❖ In an application XML file in a text editor, add at least one tile:TileReference elements as children of a view:Panel element:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app:Application label="Basic" name="Basic"
    xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/"
    xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
    xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
    xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
    xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/">
    ...
    </view:ViewManager>
</app:Application>
```
Sharing a saved view

After you save a view in your deployed application, you can generate a link to the view. This option is helpful if you want to share your design with a reviewer or any other user who has access to the application.

To access the link, click the Show View Link option in the view menu of your deployed application. The view menu is available when you include an Organizer element in the application design.

If your tile programmatically saves a view, you can create a link to it using the format


**server:port** Name and port number of the Mosaic server where the application is deployed

**root path** Root folder of the project that contains the application

**application name** Name of the application that you created the view in

**view ID** Universally unique identifier for the view

Tiles can retrieve the unique identifier for a saved view by looping through the `mosaicApp.userViews` array. Specify the saved view using its label, its description, or both.

In the following example, the first line saves the view data to the first index position of the array. A local variable (sampleView) holds the data for the view and an iterant (myView) stores the values for the view. The `nodeName` property accesses the unique identifier.
mosaicApp.views[0].save("<view label>", "<view description>", onSuccess, onFailure);
function onSuccess(evt:Event)
    var sampleView:IView;
    for each (var myView:IView in mosaicApp.userviews) {
        if (myView.nodeLabel == "<view label>") {
            sampleView = myView;
        }
    }
    var savedViewUUID = sampleView.nodeName;
}
function onFailure(evt:Event) {
}

Using application layouts

Layouts are optional elements that you can use to arrange the contents of an application shell, views, view managers, and panels. Mosaic includes pre-defined layouts to allow you to structure objects in these parent elements in various ways. You can also nest container elements, such as view:Panel, with different layouts to create your desired application structure.

For example, the following XML describes the Basic sample application. This example uses the layout named DynamicColumnLayout to structure the contents of two separate panels within the application:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app:Application label="Basic" name="Basic"
    xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/"
    xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
    xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
    xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
    xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/>

    <crx:Metadata>
        <crx:Description>This application demonstrates inter-tile communication using the Mosaic Tile SDK.</crx:Description>
    </crx:Metadata>
    <app:Shell label="Basic" name="Basic">
        <catalog:CatalogReference uri="sample_catalog" name="sample_catalog"/>
        <view:ViewManager height="100%" width="100%">
            <view:View height="100%" width="100%" label="Tile Communication">
                <view:Panel label="Simple Context" styleName="PanelInShellStyle">
                    <view:Layout numColumns="2" name="DynamicColumnLayout" catalog="sample_catalog" />
                    <tile:TileReference label="SetAndGetContext" name="SetAndGetContext" catalog="sample_catalog" />
                    <tile:TileReference label="ContextWatcher" name="ContextWatcher" catalog="sample_catalog" />
                    <tile:TileReference label="SimpleContext" name="SimpleContext" catalog="sample_catalog" width="100%" height="50%" />
                </view:Panel>
                <view:Layout numColumns="2" name="DynamicColumnLayout" catalog="sample_catalog" />
                <tile:TileReference label="ComplexContextSetGet" name="ComplexContextSetGet" catalog="sample_catalog" />
            </view:View>
        </view:ViewManager>
    </app:Shell>
</app:Application>
```
The following table describes the pre-defined layouts available in Mosaic. These layouts define the physical layout and orientation of their child objects. You specify these layouts using the `name` attribute of the `view:Layout` element. Layouts can be used as children of `view:Panel`, `view:View`, `view:ViewManager`, and `app:Shell` elements within an application.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Valid Parent Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsoluteLayout</td>
<td><code>view:Panel</code></td>
<td>Mimics the Flex Canvas container. Components can be placed using x and y properties or top, left, bottom, and right constraints. Components can also have their width and height values explicitly set, or set relative to the parent container.</td>
</tr>
<tr>
<td></td>
<td><code>app:Shell</code></td>
<td></td>
</tr>
<tr>
<td>ColumnLayout</td>
<td><code>view:Panel</code></td>
<td>Mimics the Flex HBox container. All components are sized to take up the entire height and have equal width. Adding more components reduces the width of all components.</td>
</tr>
<tr>
<td></td>
<td><code>app:Shell</code></td>
<td></td>
</tr>
<tr>
<td>DynamicColumnLayout</td>
<td><code>view:Panel</code></td>
<td>(Default layout for <code>view:Panel</code> elements) This layout places all of its children in equal-sized columns. The <code>numColumns</code> attribute (default: 3) can set the number of columns. Once a row is filled, the next row begins. But unlike TileLayout, only the width of the children are set to the column width. The height of a child can either be explicitly given on the tag or taken from the metadata stored in the catalog.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DynamicRowLayout</td>
<td><code>view:Panel</code></td>
<td>This layout places all of its children in equal-sized rows. The <code>numRows</code> attribute (default: 3) can set the number of rows. Once a column is filled, the next column begins. Unlike TileLayout, only the height of the children are set to the row height. The widget of a child can either be explicitly given on the tag or taken from the metadata stored in the catalog.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FlowLayout</td>
<td><code>view:Panel</code></td>
<td>Similar to tile, except the cells are not of equal size. Each row can have a different number of components, depending on their size. A direction can be specified - either horizontal or vertical.</td>
</tr>
<tr>
<td></td>
<td><code>app:Shell</code></td>
<td></td>
</tr>
<tr>
<td>HDividedBoxLayout</td>
<td><code>view:Panel</code></td>
<td>Mimics the HDividedBox Flex container. Components are laid out horizontally with a gripper in between to resize them.</td>
</tr>
<tr>
<td></td>
<td><code>app:Shell</code></td>
<td></td>
</tr>
</tbody>
</table>
Creating Applications

Note: It is recommended that you do not add a Layout child element to the ViewManager element. The default layout provides the capability to add new views to the application.

Certain layouts described in the table above use specific attributes that set parameter values associated with the layout. The following table lists attributes and maps them to their parent layout:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Applies to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>columnAlign</td>
<td>String</td>
<td>Specifies how to justify the fully visible columns to the container width. Valid values are: left, justifyUsingGap, and justifyUsingWidth. Default value is &quot;left&quot;.</td>
<td>TileLayout</td>
</tr>
<tr>
<td>columnWidth</td>
<td>Number</td>
<td>Width of individual columns, in pixels. By default, this value is not set.</td>
<td>TileLayout</td>
</tr>
</tbody>
</table>
| horizontalAlign | String  | The horizontal placement of items within a container. Values are "left", "right", "center", and "justify". | TileLayout
                                                                          HorizontalLayout
                                                                          VerticalLayout
For example, the Basic sample application included with Mosaic demonstrates the `numColumns` attribute used by the `DynamicColumnLayout`:

```xml
<view:Layout numColumns="2" name="DynamicColumnLayout"/>
```
In addition to attributes used by specific layouts, some layouts also bestow positioning attributes on their children. The following table lists the attributes that are passed to child elements, and maps those attributes to their parent layout:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description</th>
<th>Applies to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>bottom</td>
<td>Number</td>
<td>The offset from the bottom edge of the container.</td>
<td>AbsoluteLayout</td>
</tr>
<tr>
<td>height</td>
<td>Number</td>
<td>The height of the object.</td>
<td>All layouts except RowLayout and DynamicRowLayout.</td>
</tr>
<tr>
<td>left</td>
<td>Number</td>
<td>The offset from the left edge of the container.</td>
<td>AbsoluteLayout</td>
</tr>
<tr>
<td>right</td>
<td>Number</td>
<td>The offset from the right edge of container.</td>
<td>AbsoluteLayout</td>
</tr>
<tr>
<td>top</td>
<td>Number</td>
<td>The offset from the top edge of the container.</td>
<td>AbsoluteLayout</td>
</tr>
<tr>
<td>width</td>
<td>Number</td>
<td>The width of the object.</td>
<td>All layouts except ColumnLayout and DynamicColumnLayout.</td>
</tr>
<tr>
<td>x</td>
<td>Number</td>
<td>The horizontal position of an object as measured from the container’s left side.</td>
<td>AbsoluteLayout</td>
</tr>
<tr>
<td>y</td>
<td>Number</td>
<td>The vertical position of an object as measured from the container’s top side.</td>
<td>AbsoluteLayout</td>
</tr>
</tbody>
</table>

### Creating application styles and skins

A skin is a component that covers the visual area of another component within an application. In Mosaic, a skin sets border and background colors, as well as transparency values. Skins can vary in terms of their complexity. Some are static visual user interface components, while others are more complex, containing controls that manipulate application component behaviors.

Mosaic is built using Flex 4 and the Spark component library. Every Mosaic runtime component has its own skin and set of styles. In addition, you can create custom skins and styles.

In general, when creating your own custom styles and skins, you follow these steps:

1. Create a web-based Flash Builder project.
2. Create a CSS file with your custom styles, and compile the CSS as a SWF file.
3. Add the SWF file to a catalog.
4. Create a Mosaic application that contains a catalog:CatalogReference element to identify the catalog, and a view:StyleReference tag to reference the style sheet (SWF file).
5. If necessary, add styleName attributes on those application XML elements that you want to reference your custom styles.

**Note:** Mosaic relies on the Spark skinning capabilities introduced with Flex 4. Familiarize yourself with Spark skinning by viewing Using Flex 4 before creating application skins for Mosaic.

### Default styles

Each Mosaic component uses a default style defined in the default style sheet (CSS). You can view the default Mosaic CSS in the [Mosaic installation directory]/sdk/css folder.
The following table maps the Mosaic application components to their default styles. When you create your custom CSS, you can use the default naming, or create your own.

<table>
<thead>
<tr>
<th>Component</th>
<th>Default Style Name</th>
<th>Default Layout</th>
<th>Alternate Styles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell</td>
<td>ShellStyle</td>
<td>AbsoluteLayout</td>
<td>n/a</td>
</tr>
<tr>
<td>ViewManager</td>
<td>TabbedViewManagerStyle</td>
<td>StackLayout</td>
<td>PlainViewManagerStyle</td>
</tr>
<tr>
<td>View</td>
<td>TabbedViewStyle</td>
<td>StackLayout</td>
<td>PlainViewStyle</td>
</tr>
<tr>
<td>Panel</td>
<td>PlainPanelStyle</td>
<td>DynamicColumnLayout</td>
<td>TabbedPanelStyle, PanelInShellStyle</td>
</tr>
<tr>
<td>Tile</td>
<td>TileStyle</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

When creating a custom CSS, you can choose to use the same style names to overwrite default styles, or create new styles. If you use existing style names, you do not have to explicitly specify the styles using the styleName attribute in your application XML definition. If you use custom style names, then to implement it in your application, specify the style using the styleName attribute on the corresponding element.

For example, the following CSS entry defines the default style for the view:Shell element:

```css
.shellStyle {
   skin-class: ClassReference('com.adobe.mosaic.skins.aqua.ShellSkin');
    background-alpha: 1;
}
```

To use the existing style, but change its definition, you update values for the existing class. For example, you could create a custom ActionScript or MXML skin class and reference it using the skin-class attribute:

```css
.shellStyle {
    skin-class: ClassReference('com.mycompany.styles.skins.MySkin');
    background-alpha: 1;
}
```

Or you could create a custom CSS and define a new class:

```css
.myShellStyle {
    skin-class: ClassReference('com.mycompany.styles.skins.MySkin');
    background-alpha: 1;
}
```

**Note:** If you create a custom CSS for a Mosaic application based on an HTML page, do not use the style background-alpha. If you use this style and run the application in a browser, the application content is not displayed unless the cursor is inside the display area.

If you choose to define a new class, add a reference to the style using the styleName attribute on the appropriate application XML element. For example:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app:Application label="Basic" name="Basic"
    xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/
    xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/
    xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/
    xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/
    xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/>

<crx:Metadata>
<crx:Description>This application demonstrates inter-tile communication using the
Mosaic Tile SDK.</crx:Description>
</crx:Metadata>

<app:Shell label="Basic" name="Basic" styleName="MyShellStyle">
<catalog:CatalogReference uri="sample_catalog" name="sample_catalog"/>
$view:ViewManager height="100%" width="100%">
    <view:View height="100%" width="100%" label="Tile Communication">
        <view:Panel label="Simple Context" styleName="PanelInShellStyle">
            <view:Layout numColumns="2" name="DynamicColumnLayout"/>
            <tile:TileReference label="SetAndGetContext" name="SetAndGetContext" catalog="sample_catalog"/>
            <tile:TileReference label="ContextWatcher" name="ContextWatcher" catalog="sample_catalog"/>
            <tile:TileReference label="SimpleContext" name="SimpleContext" catalog="sample_catalog" width="100%" height="50%"/>
        </view:Panel>
    </view:View>
</view:ViewManager>
</app:Shell>
</app:Application>

If you create a custom CSS, do the following before viewing the application:

- Add the style sheet to a catalog
- Deploy all related resources to a Mosaic server

For more information on adding custom style sheets to a catalog, see “Adding assets to a catalog” on page 23.

**Alternative styles**

Mosaic also provides an alternative style sheet named NavigationStyles.css. It includes the view style
ComboBoxViewStyle, which allows users to switch between panels using a combo box instead of tabs. Switch to a
combo box when you create applications in which the view and panel tabs can become crowded and make the
application hard to use.

You can view NavigationStyles.css in the [Mosaic installation directory]/samples/general/stylesheets/NavigationExample/src folder. The sample application Style Example
references the style sheet (as a compiled SWF file) and style using the StylesheetReference and styleName attributes:
<app:Shell name="Navigation" label="Style Example">
  <catalog:CatalogReference name="sample_catalog" uri="sample_catalog"/>
  <view:StylesheetReference name="NavigationStyles" catalog="sample_catalog"/>
  <view:ViewManager width="100%" height="100%">
    <view:View label="View 1" width="100%" height="100%" styleName="ComboBoxViewStyle">
      <view:Panel label="Panel 1" width="100%" height="100%">
        <view:Layout name="FlowLayout" />
        <tile:TileReference catalog="sample_catalog" name="RetrieveData" label="RetrieveData"/>
        <tile:TileReference catalog="sample_catalog" name="ComplexContextSetGet" label="ComplexContextSetGet"/>
        <tile:TileReference catalog="sample_catalog" name="ComplexContextWatcher" label="ComplexContextWatcher"/>
        <tile:TileReference catalog="sample_catalog" name="SetAndGetContext" label="SetAndGetContext"/>
      </view:Panel>
      <view:Panel label="Panel 2" width="100%" height="100%">
        <view:Layout name="FlowLayout" />
        <tile:TileReference catalog="sample_catalog" name="ContextWatcher" label="ContextWatcher"/>
        <tile:TileReference catalog="sample_catalog" name="MessageSender" label="MessageSender"/>
        <tile:TileReference catalog="sample_catalog" name="MessageReceiver" label="MessageReceiver"/>
      </view:Panel>
    </view:View>
  </view:ViewManager>
</app:Shell>

Linking skins to application components

When you create a skin, you link it with a component or set of components to have that skin display in an application at runtime. You create the link between skin and component using the HostComponent metadata element in the MXML definition of the skin.

For example, the following MXML is an abbreviated version of the DropDownNavigatorSkin from the Navigation sample included with Mosaic:
For Mosaic components, you associate the skin with either the IMosaicContainer interface or the IMosaicComponent interface. Associating with an interface allows the skin to work with several different components, as long as each component implements the same interface.

Once you declare the HostComponent metadata, your custom skin can refer to properties and methods linked to that HostComponent. The reference is made using the built-in hostComponent property that is provided to all Spark skins. For example, the default skin used by view:View elements, TabLayoutBarSkin, uses the ButtonBar Flex component as its HostComponent. TabLayoutBarSkin then uses data-binding between its dataProvider attribute and hostComponent.contentList to display a set of tabs showing the contents of the hostComponent.

For more information on the IMosaicContainer and IMosaicComponent interfaces, see ActionScript 3.0 Reference for the Adobe Flash Platform. The ActionScript reference also has information on the TabLayoutBarSkin used by default view:View elements within Mosaic applications.

### Compiling a custom style sheet

To use your Flex project with the Mosaic skinning SDK, link them in Flash Builder. The skinning SDK is located in the [Mosaic installation directory]/sdk/flex/sdks/4.1.0/frameworks/lib folder. The SDK contains the default skin classes for the default Mosaic aqua theme, as well as the IMosaicContainer and IMosaicComponent interfaces.

You can create a style sheet SWF file for use with Mosaic using Flash Builder to compile a CSS to a SWF file.

#### Creating a style sheet SWF file using Flash Builder:

1. Start Flash Builder.
2. Select File > New > Flex Project.
3. Type a project name, assign a workspace, set the Flex SDK version to 3.4.0 or a later version of the SDK, and then click Next.
4. Click Finish.
5. Add a CSS containing your Mosaic custom styles, as well as any assets referenced (such as images), to your Flash Builder project.
6. Right-click the CSS file and ensure that the Compile CSS to SWF option is selected.
Creating default views and panels

Default views and panels are new views and panels added by an end user to an application. Initially, these default views and panels are blank, but you can create default view and panel definitions within an application. In this way, when users add views or panels to an application at runtime, those views and panels contain specific content automatically.

To create default views and panels, add the app:Defaults element to an existing application definition. For example, the following application XML adds default views with default panels to the Basic sample application included with Mosaic:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<app:Application name="Basic" label="Basic"
xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/"
xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/">
<crx:Metadata>
<crx:Description>Basic Application</crx:Description>
</crx:Metadata>
<app:Defaults>
<view:Panels>
<view:Panel width="100%" height="100%" tileChrome="full">
<catalog:CatalogReference name="sample_catalog" uri="sample_catalog"/>
<tile:TileReference catalog="sample_catalog" name="ComplexContextSetGet" label="ComplexContextSetGet"/>
</view:Panel>
</view:Panels>
<view:Views>
<view:View label="Two Panel View" width="100%" height="100%">
<view:Panel label="Panel 1" width="100%" height="100%" tileChrome="full">
<catalog:CatalogReference name="sample_catalog" uri="sample_catalog"/>
<tile:TileReference catalog="sample_catalog" name="MessageSender" label="MessageSender"/>
</view:Panel>
<view:Panel label="Panel 2" width="100%" height="100%" tileChrome="full">
<catalog:CatalogReference name="sample_catalog" uri="sample_catalog"/>
<tile:TileReference catalog="sample_catalog" name="MessageReceiver" label="MessageReceiver"/>
</view:Panel>
</view:View>
</view:Views>
</app:Defaults>
<app:Shell name="Basic" label="Basic">
<catalog:CatalogReference name="sample_catalog" uri="sample_catalog"/>
</app:Shell>
</app:Application>
```
A default panel contains tile:TileReference elements indicating the specific tiles to display. Similarly, a default view contains view:Panel and tile:TileReferences elements. Ensure that you reference the appropriate catalogs using the catalog:CatalogReference within the view:View, view:Panel, and tile:TileReference definitions.

The structure of the app:Defaults element and its children are important. For example, if a user adds a default view and selects one that contains no child view:Panel elements, the view displays without any panels. This result occurs even if the app:Defaults element contains default panels as children. Views must enable users to add new panels to allow adding default panels. For more information on the XML structure of these elements, see LiveCycle Mosaic 9.5 XML Schema Reference.

### Setting when tiles are loaded

Use the loadPolicy attribute to choose when an application loads a Flex-based tile.

Loading a tile involves retrieving all the logic, styles, and content associated with the tile from the Mosaic server. If your application has large or complex tiles, or many tiles, loading them all at startup can increase how long it takes to open. The loadPolicy attribute allows you to specify which tiles are not loaded until the user opens a view in which they are displayed.

The loadPolicy attribute is available for TileClass elements when you create catalogs. It is also available for TileReference elements when you define the contents of a view or panel in an application. It can have one of the following values:

- **all** When the user opens an application, the tile is created, even if it is not displayed in the current view or panel. This value is the default.
- **auto** The tile is created only when the tile's view or panel is displayed.
Tiles can inherit their loadPolicy setting from their parent components. For example, a tile is defined in a catalog with a loadPolicy value of all. When an application that uses this tile is opened, the tile is loaded. However, if the same tile has a loadPolicy value of auto, the catalog setting is ignored. A tile with a loadPolicy value of auto loads only when the panel is displayed.

The following catalog XML illustrates the loadPolicy attribute. The value is disregarded if a child component sets loadPolicy for the tile:

```xml
<tile:TileClass initialHeight="300" initialWidth="600" label="openIssues" name="openIssues" catalog="sampleCatalog" fitContent="true" loadPolicy="auto">
  <crx:Metadata>
    <crx:Description>Display Stock information for a set of symbols</crx:Description>
    <crx:Category>Component</crx:Category>
    <crx:Tag>component</crx:Tag>
    <crx:Tag>menu</crx:Tag>
  </crx:Metadata>
  <tile:Content loadAs="default" contentType="application/x-shockwave-flash" uri="/mosaic/catalogs/sample_catalog/tiles/openIssues/content"/>
</tile:TileClass>
```

In the following example of application XML, Open Issues and Order History are large tiles. By setting loadPolicy to auto, those tiles are not loaded until the user opens the view that contains them (Customer Service).
Creating policy-protected applications

To restrict access to Mosaic assets, you create policies that define which user roles are entitled to access the assets. Mosaic assets include applications, catalogs, tiles, style sheets, views, panels, and resources. You create roles and assign users to them using LiveCycle Administration Console.

For more information about creating policies and assigning them to applications, see “Creating and Assigning Policies” on page 52.
Deploying the application

Once you have created an application file, deploy the application to a Mosaic server in order for the application to be available for end users.

For more information on deploying catalogs to the Mosaic server, see “Deploying assets to the server” on page 67.

More Help topics
“Overview” on page 1
“Creating Flex tiles” on page 3
“Creating HTML Tiles” on page 12
“Creating Catalogs” on page 21
“Creating and Assigning Policies” on page 52
“Creating Services” on page 59
“Deploying assets to the server” on page 67
“Debugging” on page 73
Chapter 6: Creating and Assigning Policies

To restrict access to Mosaic assets, you create policies that define which user roles are entitled to access the assets. Mosaic assets include applications, catalogs, tiles, style sheets, views, panels, and resources. You create roles and assign users to them using LiveCycle Administration Console. (For more information, see “Installing and Configuring the Production Environment” in LiveCycle Mosaic 9.5 Getting Started.)

Note: Make sure that you have a thorough knowledge of writing XACML policies before you attempt to create policies for Mosaic.

A generic Mosaic policy file contains the following structure:

```xml
<crx:PolicyClass
    name="string"
    label="string">
    <crx:Metadata>
    <crx:Content>
        <xacml:Policy>
        </xacml:Policy>
    </crx:Content>
</crx:PolicyClass>
```

The Policy element defines an XACML policy as described by http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-policy-schema-os.xsd.

Default policy
Mosaic includes a default XACML policy file, mosaic_default_asset_policy.xml, located in the WEB-INF/classes/config/mosaic/xacml folder of the mosaic.war file. This policy file defines all of the default system policies with respect to what users and roles are allowed to perform actions on Mosaic resources.

Protected asset types
Mosaic allows policies to control access to the following assets:

- Applications
- Catalogs
- Panel templates
- Saved views
- Shell templates
- Style sheet templates
- Tile templates
- View templates

Asset resource IDs

Assets stored in the Mosaic policy store are accessible using asset type identifiers. These identifiers include hostname and port values to allow you to create references to multiple XACML servers (if necessary).
The following table lists the protected asset types with the associated identifiers supported by Mosaic:

<table>
<thead>
<tr>
<th>Asset</th>
<th>Identifier</th>
<th>Asset Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application List</td>
<td>mosaic://[hostname]/[port]/applications</td>
<td>crx:ApplicationList root</td>
</tr>
<tr>
<td>Application</td>
<td>mosaic://[hostname]/[port]/applications/[application-name]</td>
<td>crx:Application root</td>
</tr>
<tr>
<td>Saved View List</td>
<td>mosaic://[hostname]/[port]/applications/views</td>
<td>crx:ViewClassList root</td>
</tr>
<tr>
<td>Saved View</td>
<td>mosaic://[hostname]/[port]/applications/views/[view-name]</td>
<td>crx:ViewClass root</td>
</tr>
<tr>
<td>Catalog List</td>
<td>mosaic://[hostname]/[port]/catalogs</td>
<td>crx:CatalogList root</td>
</tr>
<tr>
<td>Catalog</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]</td>
<td>crx:Catalog root</td>
</tr>
<tr>
<td>Tile List</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/tiles</td>
<td>crx:TileClassList root</td>
</tr>
<tr>
<td>Tiles</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/tiles/[tile-name]</td>
<td>crx:TileClass (root local tile content, either SWF file or other file)</td>
</tr>
<tr>
<td>Style sheet List</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/stylesheets</td>
<td>crx:StylesheetClassList root</td>
</tr>
<tr>
<td>Style sheets</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/stylesheets/[stylesheet-name]</td>
<td>crx:StylesheetClass (root local style sheet content, either SWF file or CSS file)</td>
</tr>
<tr>
<td>Panel List</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/panels</td>
<td>crx:PanelClassList root</td>
</tr>
<tr>
<td>Panels</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/panels/[panel-name]</td>
<td>crx:PanelClass root</td>
</tr>
<tr>
<td>View Template List</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/views</td>
<td>crx:ViewClassList root</td>
</tr>
<tr>
<td>View Templates</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/views/[view-name]</td>
<td>crx:ViewClass root</td>
</tr>
<tr>
<td>Policy List</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/policies</td>
<td>crx:PolicyClassList root</td>
</tr>
<tr>
<td>Policies</td>
<td>mosaic://[hostname]/[port]/catalogs/[catalog-name]/policies/[policy-name]</td>
<td>crx:PolicyClass (root XACML XML file)</td>
</tr>
</tbody>
</table>
Internal asset entities

For each asset identifier, the associated metadata entity is defined according to the following table:

<table>
<thead>
<tr>
<th>Element</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolicyReference</td>
<td>[asset-identifier]/PolicyReference</td>
</tr>
<tr>
<td>Metadata/OwnedBy</td>
<td>[asset-identifier]/Metadata/OwnedBy</td>
</tr>
<tr>
<td>Metadata/CreatedBy</td>
<td>[asset-identifier]/Metadata/CreatedBy</td>
</tr>
<tr>
<td>Metadata/CreatedDate</td>
<td>[asset-identifier]/Metadata/CreatedDate</td>
</tr>
<tr>
<td>Metadata/ModifiedBy</td>
<td>[asset-identifier]/Metadata/ModifiedBy</td>
</tr>
<tr>
<td>Metadata/ModifiedDate</td>
<td>[asset-identifier]/Metadata/ModifiedDate</td>
</tr>
<tr>
<td>Metadata/Category</td>
<td>[asset-identifier]/Metadata/Category</td>
</tr>
<tr>
<td>Metadata/Tag</td>
<td>[asset-identifier]/Metadata/Tag</td>
</tr>
<tr>
<td>Metadata/Description</td>
<td>[asset-identifier]/Metadata/Description</td>
</tr>
</tbody>
</table>

For example, the following policy file snippet restricts changing the crx:OwnedBy entity to the current owner:

```xml
<Resource>
  <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:regexp-string-match">
    <AttributeValue
      DataType="http://www.w3.org/2001/XMLSchema#string">^mosaic://[^#]*#xpointer(/crx:Metadata/crx:OwnedBy)</AttributeValue>
    <ResourceAttributeDesignator
      DataType="http://www.w3.org/2001/XMLSchema#string"AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
  </ResourceMatch>
</Resource>
```

Classifying metadata elements

Mosaic assets can contain metadata elements that hold information related to the content. These metadata elements are classified with different levels of protection as follows:

- **current owner** Modifiable by the user specified in the OwnedBy element.
- **system** Not modifiable by any users.
- **user** Modifiable by any user with write permissions to the asset.

No one can modify the elements classified as "system" data. Elements classified as "current owner" are modifiable by the user specified in the OwnedBy element. Elements classified as "user" are modifiable by any user with "Write" permission on the asset.

- Category (current owner)
- CreatedBy (system)
- CreatedDate (system)
- Description (user)
- ModifiedBy (system)
- ModifiedDate (system)
Assigning policies to an application

Once you create and deploy a policy file, explicitly reference the policy within the descriptor of each asset to which the policy applies. If you do not reference the policy in this way, the system policy is applied.

The crx:PolicyReference element must appear immediately following the required crx:Metadata element. Policies can control virtually any asset which contains a crx:Metadata element.

Assign a policy to an application:
❖ Open an application XML file in a file editor, and add a PolicyReference element as a child of the root element:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app:Application label="Basic" name="Basic"
   xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/"
   xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
   xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
   xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
   xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/">
  <crx:Metadata>
    <crx:Description>This application demonstrates inter-tile communication using the Mosaic Tile SDK.</crx:Description>
  </crx:Metadata>
  <crx:PolicyReference name="MyPolicy"/>
</app:Application>
```

Assign a policy to an application:
❖ Open an application XML file in a file editor, and add a PolicyReference element as a child of the root element:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app:Application label="Basic" name="Basic"
   xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/"
   xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
   xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
   xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
   xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/">
  <crx:Metadata>
    <crx:Description>This application demonstrates inter-tile communication using the Mosaic Tile SDK.</crx:Description>
  </crx:Metadata>
  <crx:PolicyReference name="MyPolicy"/>
</app:Application>
```

Last updated 3/31/2011
Assigning policies to a catalog

Assign a policy to a catalog:
❖ Open a catalog XML file in a file editor, and add a PolicyReference element as a child of the root element. For example, the following XML source illustrates adding a policy named MyPolicy to the catalog descriptor.xml for the Basic sample application:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog:Catalog name="sample_catalog" label="Sample Catalog"
    xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/"
    xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
    xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
    xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
    xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/>

    <crx:Metadata>
        <crx:Description>Sample Catalog</crx:Description>
    </crx:Metadata>

    <crx:PolicyReference name="MyPolicy"/>

    <catalog:Categories>
        <catalog:Category name="Component"/>
        <catalog:Category name="Utility"/>
    </catalog:Categories>

</catalog:Catalog>
```
Assign a policy to assets in a catalog:

❖ Open a catalog XML file in a file editor. Add a crx:PolicyReference element as a child of the root class element for each asset that you want to manage with the policy. Add the crx:PolicyReference element immediately following the crx:Metadata element. For example, the following XML source illustrates adding a policy named MyPolicy to assets. The assets are a subset contained in the catalog descriptor.xml for the Basic sample application:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<catalog:Catalog name="sample_catalog" label="Sample Catalog"
 xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/"
 xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
 xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
 xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
 xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/">

<crx:Metadata>
  <crx:Description>Sample Catalog</crx:Description>
</crx:Metadata>
<catalog:Categories>
  <catalog:Category name="Component"/>
  <catalog:Category name="Utility"/>
</catalog:Categories>

<view:StylesheetClassList>
  <view:StylesheetClass name="NavigationStyles" catalog="sampleCatalog">
    <crx:Metadata>
      <crx:Description>An example of an alternate navigation control for Views and ViewManagers.</crx:Description>
    </crx:Metadata>
    <crx:PolicyReference name="MyPolicy"/>
    <view:Content uri="/mosaic/catalogs/sampleCatalog/Stylesheets/NavigationStyles/content" contentType="application/x-shockwave-flash"/>
  </view:StylesheetClass>
</view:StylesheetClassList>

<tile:TileClassList>
  <tile:TileClass initialHeight="400" initialWidth="680" label="Default" name="Default" catalog="sampleCatalog" fitContent="true">
    <crx:Metadata>
      <crx:Description>Component</crx:Description>
    </crx:Metadata>
    <crx:PolicyReference name="MyPolicy"/>
    <tile:Content loadAs="default" contentType="application/x-shockwave-flash" uri="/mosaic/catalogs/sample_catalog/tiles/Default/content"/>
  </tile:TileClass>
</tile:TileClassList>
```
Deploying policies

Once you have created a policy file and referenced it in an application, deploy the policy to a Mosaic server. The policy takes effect after you deploy it to the server.

For more information on deploying catalogs to the Mosaic server, see “Deploying assets to the server” on page 67.

More Help topics
“Overview” on page 1
“Creating Flex tiles” on page 3
“Creating HTML Tiles” on page 12
“Creating Catalogs” on page 21
“Creating Applications” on page 30
“Creating Services” on page 59
“Deploying assets to the server” on page 67
“Debugging” on page 73
Chapter 7: Creating Services

Services, in the context of Mosaic, are essentially tiles that do not have any user interface. Using a service has the following advantages:

- Provides a centralized method for defining and initializing values used by tiles
- Ensures that tiles always load with a set of data, regardless of the order in which the tiles load within an application
- Performs tasks that are initialized before any tiles are loaded in the Mosaic application

For example, you can create a service that enables tiles to access and submit information using LiveCycle Data Services.

A service is composed of two libraries:

- Interface Library: De-couples the definition of an object from its actual implementation. Tiles reference the interfaces to access functions provided by the service.
- Service Library: Linked to the interfaces, classes in the service library perform functions, such as data lookup, but do not include any user interfaces.

Note: The mockStockService sample included with Mosaic is used as an example throughout. The mockStockService sample is located in the [Mosaic installation directory]/samples/stockDataService folder.

Interface library

For example, the mockStockService sample contains the following interfaces compiled as the StockDataInterfacesLibrary.swf (located in the [Mosaic installation directory]/samples/stockDataService/catalogs/Interfaces folder):

```swfl
package com.adobe.mosaic.samples.services.stockdata
{
    public interface IStockData
    {
        function get symbol():String;
        function get volume():Number;
        function get price():Number;
        function get relatedStocks():Vector.<IStockData>;
    }
}
```

```swfl
package com.adobe.mosaic.samples.services.stockdata
{
    public interface IStockDataService
    {
        function getQuote(symbol:String):IStockData;
    }
}
```

The purpose of these interfaces is to provide an implementation of the stockDataService for tiles to reference.

Create an interface library:

1. In Flash Builder, click File > New > Flex Library Project.
In the Project Name field, type a unique name for your project. For example, SampleInterfaceLibrary.

For Application Type, ensure that Web is selected, and for Flex SDK Version, ensure that you select 3.4.0 or later, and then click Finish.

In the Flex Navigator view, right-click the project, and then click Properties.

Click Flex Compiler.

If you are using Flex SDK 4.1.0, add the following to the Additional Compiler Arguments field:

- includes=mx.managers.systemClasses.MarshallingSupport -static-link-runtime-shared-libraries=true

Click OK.

When finished, create your interface classes.

To add your interface library to a catalog, you must compile your interface library project SWC file as a SWF file. You can create an ANT build.xml file that converts your SWC file to a SWF file. For an example, see the example build.xml located in the [Mosaic installation directory]/samples/stockDataService/interfaces/StockDataInterfaceLibrary folder.

### Service library

The service library performs the functions, such as data lookup, the results of which you want to make available to tiles within your application. For example, the mockStockService sample contains several classes compiled as the StockDataService.swf (located in the [Mosaic installation directory]/samples/stockDataService/services/StockDataService folder). These classes collectively perform a service of data lookups. Among these classes, the StockDataService class keeps a list of stock information, and updates it when a symbol is requested from a tile:

```typescript
package com.adobe.mosaic.samples.services.stockdata.impl
{
    import com.adobe.mosaic.samples.services.stockdata.IStockData;
    import com.adobe.mosaic.samples.services.stockdata.IStockDataService;

    import flash.events.TimerEvent;
    import flash.utils.Dictionary;
    import flash.utils.Timer;
    import mx.utils.NameUtil;

    public class StockDataService implements IStockDataService
    {
        private var _data:Dictionary = new Dictionary(true);
        private var _timer:Timer = new Timer(1000);

        public function StockDataService(server:String="http://localhost",param:String="15")
        {
            _data["ADBE"] = new StockData("ADBE",5.61,36.15);
            _data["GOOG"] = new StockData("GOOG",4.37,531.64);
            _data["MFST"] = new StockData("MFST",63.65,31.11);
        }

        public function getQuote(symbol:String):IStockData
        {
```
DEVELOPING APPLICATIONS FOR LIVECYCLE MOSAIC 9.5

Creating Services

```
for (var k:String in _data) {
    _data[k] = changeStock(_data[k]);
}

return _data[symbol];
}
private function changeStock(s:IStockData):IStockData{
    var ns:IStockData = new
    StockData(s.symbol, getNewPrice(s.price), getNewVolume(s.volume));
    return ns;
}
private function getNewVolume(n:Number):Number{
    return int((Math.random() + n)*100)/100;
}
private function getNewPrice(n:Number):Number{
    var d:Number = ((Math.round(Math.random() * 10) % 2) == 0)?1:-1;
    return int((n + (d * Math.random())*100)/100;
}
```

Create a service library:
1 In Flash Builder, click File > New > Flex Library Project.
2 In the Project Name field, type a unique name for your project. For example, SampleServiceLibrary.
3 For Application Type, ensure that Web is selected, and for Flex SDK Version, ensure that you select 3.4.0 or later, and then click Finish.
4 In the Flex Navigator view, right-click the project, and then click Properties.
5 Click Flex Compiler.
6 If you are using Flex SDK 4.1.0, add the following to the Additional Compiler Arguments field:
   -includes=mx.managers.systemClasses.MarshallingSupport -static-link-runtime-shared-libraries=true
7 Click OK.

When finished, create your service classes.

To add your service library to a catalog, you must compile your service library project SWC file as a SWF file. You can create an ANT build.xml file to convert your SWC file to a SWF file. For an example, see the build.xml located in the [Mosaic installation directory]/samples/stockDataService/services/StockDataService folder.

Using services within a Flex tile

Once you have created a service library with a corresponding interface library, you can reference the interfaces you created in Flex tiles. For example, the tile named StockDataTable, included with the mockStockService sample, uses the interfaces from the interface library to reference stock data:
Adding services to a catalog

To make interface and service libraries available within an application, you add them to a catalog definition. You also ensure that the SWF files for both libraries are available within the catalog folder structure of your Mosaic project.

Add interfaces and services to a catalog:
1. Copy the corresponding SWF files into the following folders of your Mosaic project:
   - `[ProjectName]\catalogs\Interfaces`
   - `[ProjectName]\catalogs\Services`
2 Update the descriptor.xml for the catalog to reference the appropriate interfaces and services.

- For each tile that references an interface to a service, add references to the interface using the tile:Interface element. (The tile:Interface element is enclosed in a tile:Depends parent element.) Use the library and interfaceName attributes to specify the interface you want to make available to the tile.
- Add a tile:InterfaceLibraryList element with tile:InterfaceLibrary child elements for each interface you want to include in the catalog.
- Add a tile:ServiceLibraryList element with tile:ServiceLibrary child elements for each service you want to include in the catalog.

For example, the following catalog XML definition describes the mockStockData sample included with Mosaic:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<catalog:Catalog name="sampleCatalog" label="Stock Data Service Sample Catalog"
    xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/
    xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/
    xmlns:panel="http://ns.adobe.com/Mosaic/Panel/1.0/
    xmlns:shell="http://ns.adobe.com/Mosaic/Shell/1.0/
    xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/
    xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/">

    <crx:Metadata>
        <crx:Description><![CDATA[
The Mock Stock Data Service application.]]></crx:Description>
    </crx:Metadata>
    <catalog:Categories>
        <catalog:Category name="Services"/>
    </catalog:Categories>

    <tile:TileClassList>
        <tile:TileClass name="StockDataTable" label="StockDataTable"
            catalog="sampleCatalog" initialWidth="400" initialHeight="400">
            <crx:Metadata>
                <crx:Description><![CDATA[This tile has a description. It can me plain text or contain HTML.]]></crx:Description>
                <crx:Category>Finance</crx:Category>
            </crx:Metadata>
            <tile:Depends>
                <tile:Interface library="StockDataInterfaceLibrary" interfaceName="com.adobe.mosaic.samples.services.stockdata.IStockData"/>
                <tile:Interface library="StockDataInterfaceLibrary" interfaceName="com.adobe.mosaic.samples.services.stockdata.IStockDataService"/>
            </tile:Depends>
            <tile:Properties>
                <tile:Property type="com.adobe.mosaic.samples.services.stockdata.IStockDataService" name="stockDataService"/>
            </tile:Properties>
            <tile:Content
                url="/mosaic/catalogs/sampleCatalog/tiles/StockDataTable/content"
                contentType="application/x-shockwave-flash" loadAs="default"/>
        </tile:TileClass>

        <tile:TileClass name="StockDetails" label="StockDetails" catalog="sampleCatalog"/>
    </tile:TileClassList>
</catalog:Catalog>
```
DEVELOPING APPLICATIONS FOR LIVECYCLE MOSAIC 9.5

Creating Services

initialWidth="400" initialHeight="400">
<crx:Metadata>
<crx:Description><![CDATA[This tile has a description. It can be plain text or contain HTML.]]></crx:Description>
<crx:Category>Finance</crx:Category>
</crx:Metadata>
<tile:Depends>
<tile:Interface library="StockDataInterfaceLibrary"
interfaceName="com.adobe.mosaic.samples.services.stockdata.IStockData"/>
</tile:Depends>
<tile:Content uri="/mosaic/catalogs/sampleCatalog/tiles/StockDetails/content"
contentType="application/x-shockwave-flash" loadAs="default"/>
</tile:TileClass>

</tile:TileClassList>

<tile:InterfaceLibraryList>
<tile:InterfaceLibrary name="StockDataInterfaceLibrary">
<crx:Metadata>
<crx:Description>Stock Data Interfaces</crx:Description>
</crx:Metadata>
<tile:InterfaceList>
<tile:Interface
interfaceName="com.adobe.mosaic.samples.services.stockdata.IStockData"/>
<tile:Interface
interfaceName="com.adobe.mosaic.samples.services.stockdata.IStockDataService"/>
</tile:InterfaceList>
</tile:InterfaceLibrary>
</tile:InterfaceLibraryList>

<tile:ServiceLibraryList>
<tile:ServiceLibrary name="StockDataService">
<crx:Metadata>
<crx:Description>Mock Stock Data service</crx:Description>
</crx:Metadata>
<tile:ServiceClassList>
<tile:ServiceClass
className="com.adobe.mosaic.samples.services.stockdata.impl.StockDataService"
name="mockStockDataService" scope="singleton">
<tile:Implements>
<tile:Interface library="StockDataInterfaceLibrary"
interfaceName="com.adobe.mosaic.samples.services.stockdata.IStockDataService"/>
Adding interfaces and services to an application

Once you have included interfaces and services in a catalog, you can reference them within dependent tiles in an application XML definition. For example, the following is the application XML definition for the mockStockService sample included with Mosaic. Note the method of referencing the service using the tile:Property element (bolded):

```
</tile:Implements>
<tile:ConstructorArgs>
  <tile:Argument value="http://serverOne.corp.adobe.com"/>
  <tile:Argument value="arg2"/>
</tile:ConstructorArgs>
</tile:ServiceClass>

<tile:ServiceClass
  className="com.adobe.mosaic.samples.services.stockdata.impl.AsyncStockDataService"
  name="asyncStockDataService" scope="singleton" initMethod="startup">
  <tile:Implements>
    <tile:Interface library="StockDataInterfaceLibrary"
      interfaceName="com.adobe.mosaic.samples.services.stockdata.IStockDataService"/>
  </tile:Implements>
</tile:ServiceClass>
</tile:ServiceClassList>
</tile:ServiceLibraryList>
</catalog:Catalog>

For more information on the XML structure of these elements, see LiveCycle Mosaic 9.5 XML Schema Reference.
<app:Application name="mockStockService" label="mockStockService"
xmlns:view="http://ns.adobe.com/Mosaic/View/1.0/"
xmlns:catalog="http://ns.adobe.com/Mosaic/Catalog/1.0/"
xmlns:tile="http://ns.adobe.com/Mosaic/Tile/1.0/"
xmlns:crx="http://ns.adobe.com/Mosaic/CRXTypes/1.0/"
xmlns:app="http://ns.adobe.com/Mosaic/Application/1.0/">
  <crx:Metadata>
    <crx:Description>Application showing a Mock "service" connected to a Tile and strongly typed inter-tile communication.</crx:Description>
  </crx:Metadata>
  <app:Shell name="mockStockService" label="mockStockService">
    <catalog:CatalogReference name="stockDataService_catalog" uri="stockDataService_catalog"/>
    <view:Organizer visible="false"/>
    <view:ViewManager width="100%" height="100%">
      <view:View label="View 1">
        <view:Panel label="Drop-Down List Test" width="100%" height="100%">
          <view:Layout name="FlowLayout"/>
          <tile:TileReference name="StockDataTable" label="StockDataTable"
catalog="stockDataService_catalog">
            <tile:Properties>
              <tile:Property name="stockDataService">
                <tile:ServiceReference catalog="stockDataService_catalog"
library="StockDataService"
name="mockStockDataService"/>
              </tile:Property>
            </tile:Properties>
          </tile:TileReference>
          <tile:TileReference name="StockDetails" label="StockDetails"
catalog="stockDataService_catalog">
          </tile:TileReference>
        </view:Panel>
      </view:View>
    </view:ViewManager>
  </app:Shell>
</app:Application>

For more information on the XML structure of these elements, see LiveCycle Mosaic 9.5 XML Schema Reference.

More Help topics
  “Overview” on page 1
  “Creating Flex tiles” on page 3
  “Creating HTML Tiles” on page 12
  “Creating Catalogs” on page 21
  “Creating Applications” on page 30
  “Creating and Assigning Policies” on page 52
  “Deploying assets to the server” on page 67
  “Debugging” on page 73
Chapter 8: Deploying assets to the server

Once you have created tiles, catalogs, applications, and policies, deploy them to the Mosaic server for testing, and to make them available to end users.

- Create a project folder structure and copy any tile, catalog, application, and policy files into your folder structure (if necessary).
- Create an Ant build.xml file to reference your unique assets.
- Deploy the assets to the Mosaic server using the Ant Tasks provided with Mosaic. For more information on the Mosaic Ant tasks, see LiveCycle Mosaic 9.5 Server Task Reference.

Create project folder structure

The build scripts included with Mosaic are pre-built to recognize a specific project folder structure. You can view the general relationships between folders by viewing the samples folder, and its subfolders, included in the Mosaic installation file set.

Create a project folder structure:
1. Create a folder and assign it a unique name, for example MyProject.
2. Create the following subfolders:
   - MyProject\applications
   - MyProject\catalogs
   - MyProject\catalogs\Interfaces
   - MyProject\catalogs\Resources
   - MyProject\catalogs\RSLs
   - MyProject\catalogs\Services
   - MyProject\catalogs\Stylesheets
   - MyProject\catalogs\Tiles
   - MyProject\catalogs\Tiles\[tilename]\(repeat for each tile in the catalog)
   - MyProject\policies
3. Copy the following files to the corresponding folders:
   - Generated Flex tiles (SWF file) to MyProject\catalogs\Tiles\[tilename]
   - Image files referenced by the application or tiles to MyProject\catalogs\Resources
   - Generated style sheets (SWF file) to MyProject\catalogs\Stylesheets
   - Generated service interfaces (SWF file) to MyProject\catalogs\Interfaces
   - Generated service definitions (SWF file) to MyProject\catalogs\Services
   - Policy files (XML) to MyProject\policies
4. Create a descriptor.xml catalog file in the MyProject\catalogs folder. For information on creating a catalog, see “Creating Catalogs” on page 21.
When you have finished, create an Ant build.xml file to deploy the assets to the Mosaic server.

Create a build.xml file

The build.xml file facilitates the deployment of Mosaic tiles, catalogs, and applications to the server. The file itself is broadly divided into the following primary targets:

- **deploy-catalogs**
  References the catalog containing Flex and HTML tiles for use in Mosaic applications. When executed, the build.xml file creates a ZIP of the catalog and its associated resources and deploys the ZIP to the Mosaic server.

- **deploy-applications**
  References one or more Mosaic application files containing tiles available in the catalog referenced in the mosaic-catalog target.

- **deploy-policies**
  References one or more Mosaic policy files containing access privileges for Mosaic assets based on roles defined on the application server.

In addition, the build.xml file is prefaced with a taskdef element and three properties references:

```xml
<taskdef resource="mosaic.tasks" classpath="../bin/mosaicTasks.jar"/>
<property name="protocol" value="http"/>
<property name="server" value="localhost"/>
<property name="port" value="8080"/>
```

The taskdef element provides a reference to the mosaicTasks.jar file which defines the Mosaic Ant tasks for the build.xml to execute correctly. By default, the mosaicTasks.jar file is located in the bin folder of the Mosaic installation file set.

The server and port properties specify the servername and port number of the server hosting Mosaic. build.xml files are included in the samples\general, samples\clientDashboard, and samples\stockDataService folders of the Mosaic installation file set. These files are configured to deploy to the standalone Tomcat server provided with Mosaic. To deploy to a production server, update these values as necessary.

Create a build.xml file:

- Create a build.xml file in the root folder of your project. For example, in the MyProjects folder.

  For example, the following build.xml creates and deploys a single catalog named mycatalog, a single application named myapp, and a single policy named mypolicy. They are deployed to a Mosaic server named myserver.corp.abc.com using the credentials designer/password.

  ```xml
  <mosaic:deploy-catalog>
    <name>mycatalog</name>
  </mosaic:deploy-catalog>
  <mosaic:deploy-application>
    <name>myapp</name>
  </mosaic:deploy-application>
  <mosaic:deploy-policy>
    <name>mypolicy</name>
  </mosaic:deploy-policy>
  ```

  **Note:** The name attribute specifies a unique name ID for each asset on the Mosaic server. See the bolded name attributes below for examples.
<project name="Deploy projects to myserver" default="deploy">
  <taskdef resource="mosaic.tasks" classpath="../bin/mosaicTasks.jar" />
  <property name="server" value="myserver.corp.abc.com" />
  <property name="port" value="8080" />
  <property name="username" value="designer" />
  <property name="password" value="password" />

  <target name="deploy" depends="deploy-catalogs, deploy-applications, deploy-policies" />

  <target name="deploy-catalogs">
    <delete file="sample_catalog.zip" />
    <zip destfile="sample_catalog.zip">
      <zipfileset dir="catalogs" />
    </zip>
    <mosaic-catalog action="import"
      name="sample_catalog"
      file="sample_catalog.zip"
      property="response"
      server="${server}"
      port="${port}"
      username="${username}"
      password="${password}" />
    <echo message="${response}" />
  </target>

  <target name="deploy-applications">
    <mosaic-application action="import"
      name="MyApp"
      file="applications/myapp.xml"
      property="response"
      server="${server}"
      port="${port}"
      username="${username}"
      password="${password}" />
    <echo message="${response}" />
  </target>

  <target name="deploy-policies">
    <mosaic-policy action="import"
      name="MyPolicy"
      file="policies/mypolicy.xml"
      property="response"
      server="${server}"
      port="${port}"
      username="${username}"
      password="${password}" />
    <echo message="${response}" />
  </target>
</project>

More Help topics
LiveCycle Mosaic 9.5 Server Task Reference
Deploying to a Mosaic server

Once you have created a project folder structure and a build.xml file, deploy your Mosaic assets to the server. You deploy the assets by executing the build.xml file on the command line.

Deploy catalogs and applications to the Mosaic server:
1. Open a new Command Prompt dialog.
2. Switch to the root folder for your project, for example C:\MyProject.
3. To execute the default deploy target, type ant.
   - The build script attempts to deploy to the Mosaic server specified in the build.xml file.

Removing applications, catalogs, and policies from the Mosaic server

Once you have deployed applications, catalogs, tiles, and policies to a Mosaic server, you can remove those assets using Ant tasks.

Remove assets from the Mosaic server:
1. Update the Ant build.xml file that contains Mosaic deploy targets with delete (undeploy) actions for the Mosaic application, catalog, or policies that you want to remove. For example, the following build.xml source removes the Basic application, the sample ClientDashBoard catalog (and any related assets), and the policy named MyPolicy:
Deploying assets to the server

```xml
<project name="Remove assets from myserver" default="remove">
  <taskdef resource="mosaic.tasks" classpath="../bin/mosaicTasks.jar" />
  <property name="server" value="myserver.corp.abc.com" />
  <property name="port" value="8080" />
  <property name="username" value="designer" />
  <property name="password" value="password" />

  <target name="remove">
    <mosaic-application action="delete"
      name="Basic"
      property="response"
      server="${server}"
      port="${port}"
      username="${username}"
      password="${password}" />
    <echo message="${response}" />

    <mosaic-catalog action="delete"
      name="sample_catalog"
      file="sample_catalog.zip"
      property="response"
      server="${server}"
      port="${port}"
      username="${username}"
      password="${password}" />
    <echo message="${response}" />

    <mosaic-policy action="delete"
      name="MyPolicy"
      toFile="myPolicy.xml"
      property="response"
      server="${server}"
      port="${port}"
      username="${username}"
      password="${password}" />
    <echo message="${response}" />
  </target>
</project>
```

You can save the build.xml file to any location. However, it is important that the classpath attribute of the taskdef element contains the appropriate absolute or relative path to the mosaicTasks.jar file.

2 Open a new Command Prompt dialog.
3 Switch to the folder containing the build.xml you updated in Step 1.
4 Type `ant`.

The build script attempts to remove the items specified in the build.xml file from the Mosaic server.

**More Help topics**

“Overview” on page 1
“Creating Flex tiles” on page 3
“Creating HTML Tiles” on page 12
“Creating Catalogs” on page 21
“Creating Applications” on page 30

Last updated 3/31/2011
Deploying assets to the server

“Creating and Assigning Policies” on page 52
“Creating Services” on page 59
“Debugging” on page 73

LiveCycle Mosaic 9.5 Server Task Reference
Chapter 9: Debugging

To assist with debugging your tiles and applications, Mosaic includes a debugging version of the Mosaic server that provides additional logging information at runtime. You can use this verbose logging to assist you in isolating and correcting issues. You can also take advantage of the debugging capabilities of Flash Builder when debugging Flex tiles.

Deploying the debug version of Mosaic server

By default, the standalone Tomcat application server instance included with Mosaic is preconfigured with the standard release version of Mosaic server. Similarly, the installation and configuration instructions in LiveCycle Mosaic 9.5 Getting Started involve deploying the standard version of the Mosaic server.

Mosaic also includes a debug version of the Mosaic server to assist with troubleshooting when developing tiles and applications. The debug version of Mosaic server provides additional information through the application server log files to assist you with troubleshooting issues. After deploying the debug version of Mosaic, access Mosaic assets, such as tiles, catalogs, and applications normally, and consult the application server log files. By default, the log files for the standalone Tomcat application server included with Mosaic are located in the [Mosaic installation directory]/standalone/logs folder.

Note: It is recommended that you deploy the debug version of Mosaic server with a development environment, rather than a production environment. For information on installing and configuring a development environment, see “Installing and Configuring the Development Environment” in LiveCycle Mosaic 9.5 Getting Started. LiveCycle Mosaic 9.5 Getting Started includes information on using the standalone Tomcat application server provided with Mosaic.

Deploy the debug version of Mosaic server to standalone Tomcat:

1. Copy the debug version of the mosaic.war file, located in the [Mosaic installation directory]/deploy/war/jdklogging/debug or [Mosaic installation directory]/deploy/war/log4j-logging/debug folder.
2. Paste the file in the standalone/webapps folder of the Tomcat application server instance.
3. Restart Tomcat.

Debugging Flex tiles

Using debug version of tiles is useful for testing purposes. However, do not deploy debug tiles or tiles that link to automation libraries to a production server.

Debugging Flex tiles in the browser client

Before you begin, ensure that:

- You are using a debug build of the browser client.
- The latest debug version of your tile is deployed to the Mosaic server.
- An application which uses the tile has also been deployed.

1. In the source view within Flash Builder, right-click and then select Debug As > Debug Configurations.
2. Select a debug configuration, or create a configuration by double-clicking Web Application.
3 In the URL Or Path To Launch section, deselect Use Default and type the URL to your Mosaic application. For example: http://localhost:8080/mosaic/#/applications/Brokerage.

4 Click debug.

Your Mosaic application launches in the browser client and you can debug your tiles.

More Help topics
“Overview” on page 1
“Creating Flex tiles” on page 3
“Creating HTML Tiles” on page 12
“Creating Catalogs” on page 21
“Creating Applications” on page 30
“Creating and Assigning Policies” on page 52
“Creating Services” on page 59
“Deploying assets to the server” on page 67