Developer's Guide

Create GeoWeb Applications with the Sample Flex Viewer

Author	**Moxie Zhang**
Group	Corporate Sales, ESRI Inc.
File	FlexViewerDevelopersGuide.pdf
Last Revised	**November 21, 2008**
Status	Public

Table of Contents

1 Introduction

This document is for developers who intend to utilize the Sample Flex Viewer to develop ArcGIS API for Flex based applications.

1.1 Prerequisites

1.1.1 Skill and Software

The Sample Flex Viewer will be available as a compiled release package and a source code package. The compiled package can be deployed out-of-the-box. The source code package will be used by developers who want to develop GeoWeb applications.

Developers who develop applications using the Sample Flex Viewer should have sufficient knowledge and experience in Adobe Flex and in developing Flash based RIA web applications.

The programming language for Adobe Flex is ActionScript, a strong typed object-oriented standard JavaScript-based language. Developers experienced in programming with Java, C#, JavaScript or other modern object-oriented languages should find it relatively easy to adapt to ActionScript.

To develop an application with GIS capabilities, the knowledge and experience of using the ArcGIS API for Flex is required.

Required software:

- ☐ Adobe Flex 3 Builder Standard Edition (Professional Edition is recommended if advanced interactive data visualization, performance profiler, or the integration of third party test tools is required)
- ☐ Adobe SDK (optional if Adobe Flex Builder is not available)
- ☐ Subversion plug-in for Eclipse (only applicable to developers who need access to Subversion source control)
- ☐ ArcGIS API for Flex (required for developing GIS applications)

1.1.2 Obtaining Source Code and Libraries

The source code will be distributed in a zip file that is included in the release package. The source code zip file's name is *flexviewer-src-1.0-.zip*. The release package is a zip file itself that can be downloaded from Code Gallery of ArcGIS API for Flex. The release package zip file is named by the Code Gallery as *AS15905.zip*. The number in the file name will change when an updated package is uploaded.

The source code zip file contains a root folder, *FlexViewer*, where contains all the source files, libraries, documents and Flex Builder project files. The library sub folder, *libs*, contains the ArcGIS API for Flex swc file that is used to build this release.

The Widget Programming Model Library is also included in the source code zip file. It is for developing a widget without requiring, accessing and compiling the full Sample Flex Viewer source code. The library file is *flexviewer-1.0.swc* and is under the root folder *FlexViewer*. The approach-two in the next chapter will cover how to use this library to create a Sample Flex Viewer widget.

1.2 File Formats

This section lists the file formats and their major files included in the release package.

Type	Extension	Files included	Description
MXML	.mxml	Source code	Flex's XML based markup language for defining UI.
ActionScript	.as	Source code	The language for developing the Flex application.
XML	.xml	config.xml and individual widget configuration files.	Used for configuration of application and widgets.
Style Sheet	.css	style.css	Define the look and feel of the Sample Flex Viewer.
Flash	.swf	Index.swf	The main Sample Flex Viewer compiled application.

		The compiled built-in widgets Example: AboutWidget.swf	Individual compiled widget
		style.swf	The compiled style sheet (aka the theme file)
Flex Library	.swc	agslib-1.0-2008-10-22.swc	ArcGIS API for Flex library
		flexviewer-1.0.swc	Sample Flex Viewer widget programming model library

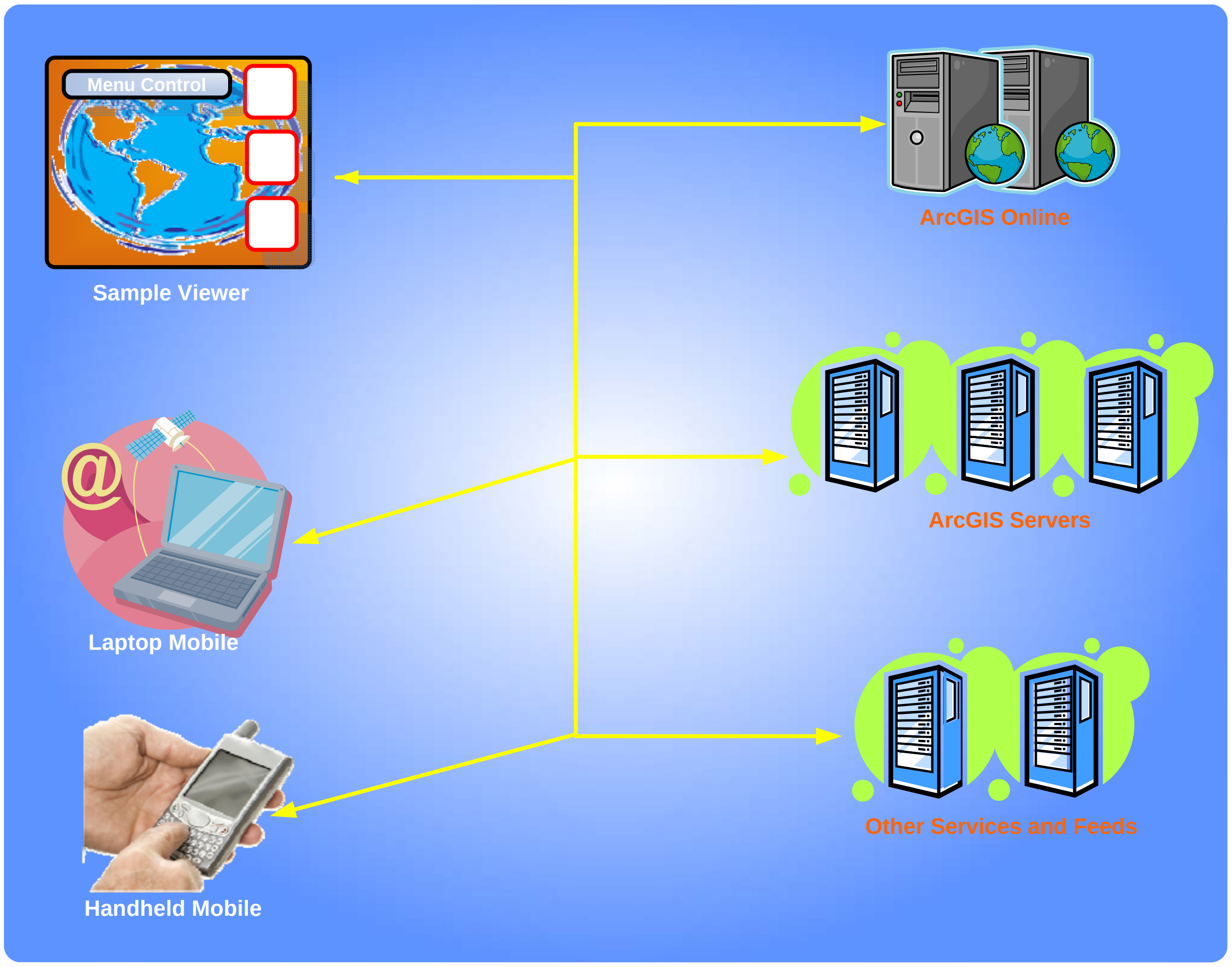
2 Sample Flex Viewer Architecture

2.1 Overview

The Sample Flex Viewer is architected to help develop and deploy GeoWeb focused applications that can fully leverage the power of the server side spatial services. The server side services could be provided by the ArcGIS Server and ArcGIS Online.

A Sample Flex Viewer application provides users a simple way to access geospatial services. As illustrated below, a geospatial service could come from the hosted SaaS (Software as a Service) type of providers such as ArcGIS Online, ArcGIS Servers or web data sources such as GeoRSS feeds, KML files, JSON/REST data, etc. The data consumed within the Sample Flex Viewer application could be the data set at server side or could be operational dynamic data generated from mobile devices such as field engineers' laptops or smart phones.

The Sample Flex Viewer is designed to be able to participate in the full spectrum of the geospatial service implementation architecture focusing on simplicity and flexibility.

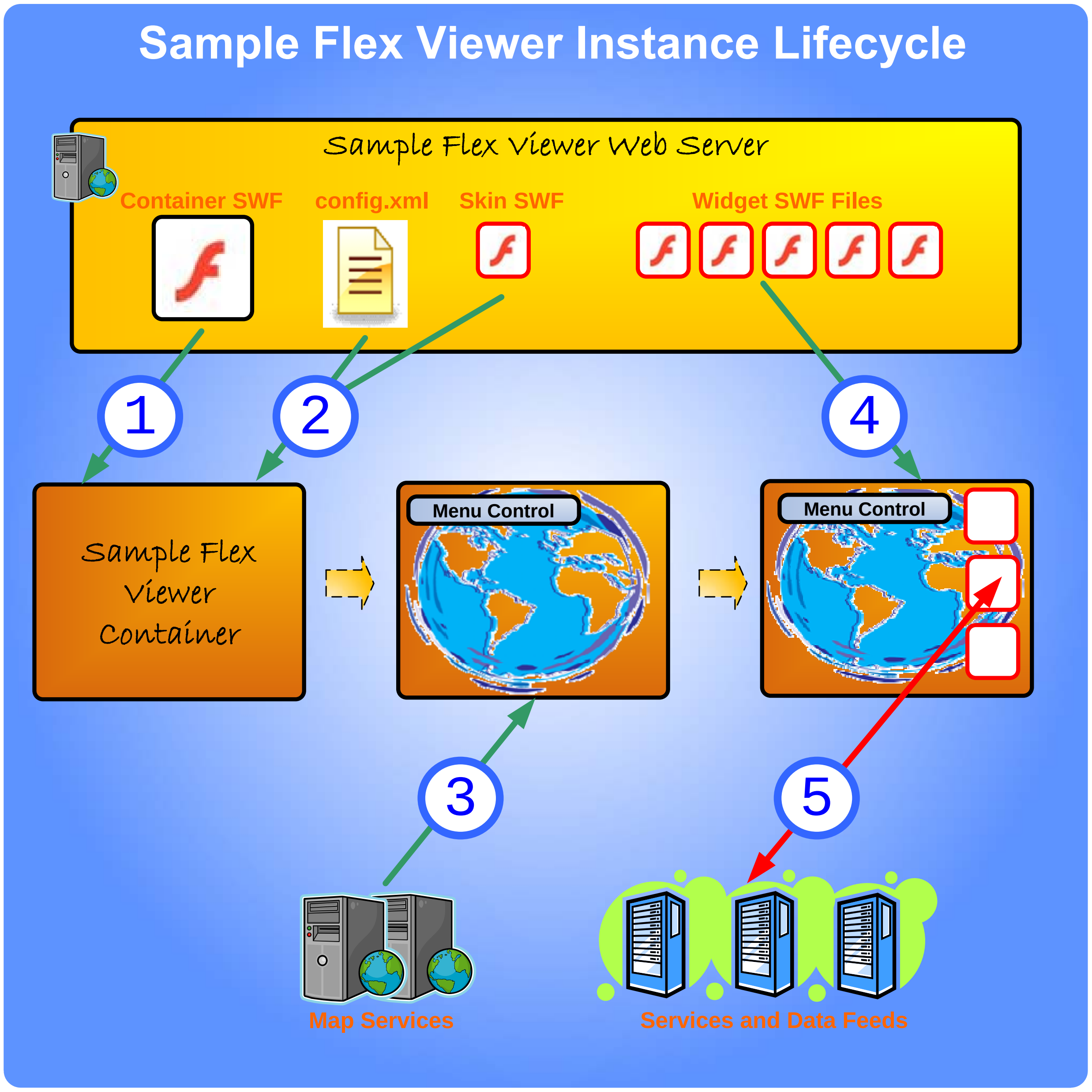


2.2 Sample Flex Viewer Instance Lifecycle

As illustrated below, a Sample Flex Viewer instance goes through a simple lifecycle from starting the application to the user's interaction with widgets. The five major lifecycle events are:

1. Flash Player starts the Sample Flex Viewer application inside a browser by loading and running the container flash file.
2. The Flex Viewer container loads the configuration XML file and skin Flash file from the web server and applies them to the entire viewer application.
3. Based on the configuration file, the container loads the map services from map servers such as ArcGIS Online or ArcGIS 9.3 servers. The container also constructs and displays the menu on the controller bar and the branding information from the configuration file.

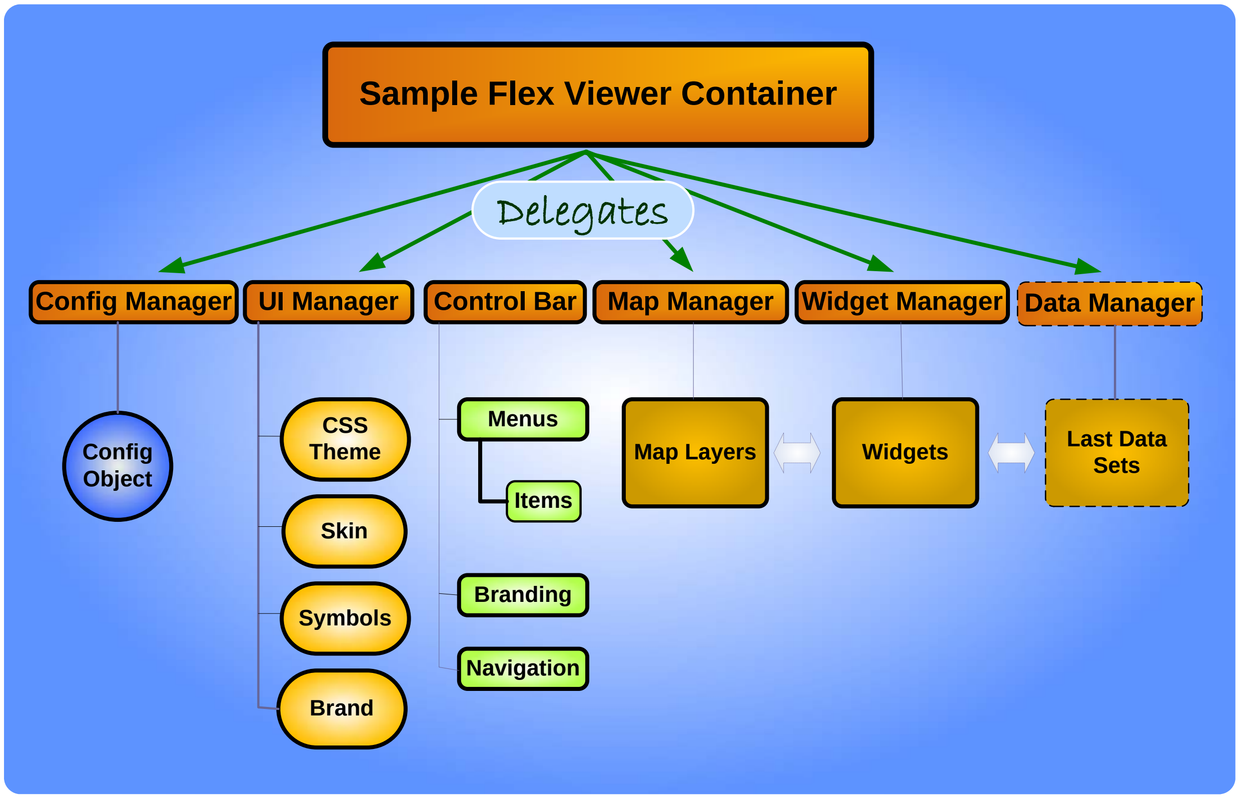
4. The container widget manager loads the widget flash files from the URLs specified in the configuration file.
5. Users interact with widgets to run business logic.



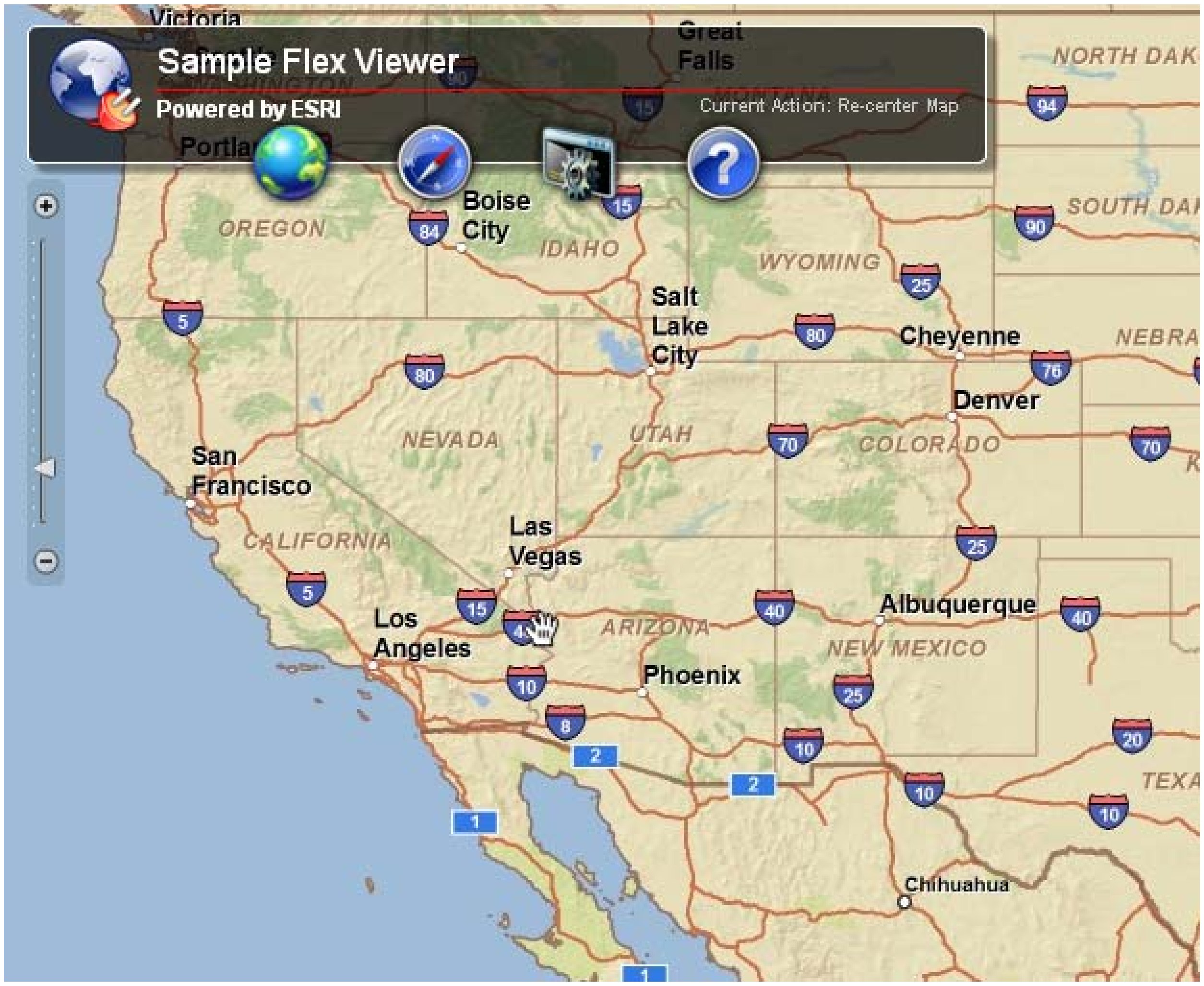
2.3 Sample Flex Viewer Container

The Sample Flex Viewer container liberates programmers from having to deal with the programming complexity of managing maps, map navigation, application configuration, inter-component communication, data management, etc. It allows Web developers, especially those working with ESRI ArcGIS technology, to focus their time and effort on implementing core business functions in their custom applications. Additionally, the results of this focused rapid development can be quickly deployed into existing Sample Flex Viewer applications in the form of widgets by simply adding a configuration entry into the Sample Flex Viewer application's configuration file.

The container consists of a set of high cohesive and low coupling components as illustrated below. The container delegates the focused tasks to the responsible components. This design approach allows simpler code maintenance and customization as well as minimizes the friction among programming modules.



The following screenshot demonstrates a typical Sample Flex Viewer application container.



2.4 Understanding the Widget Programming Model

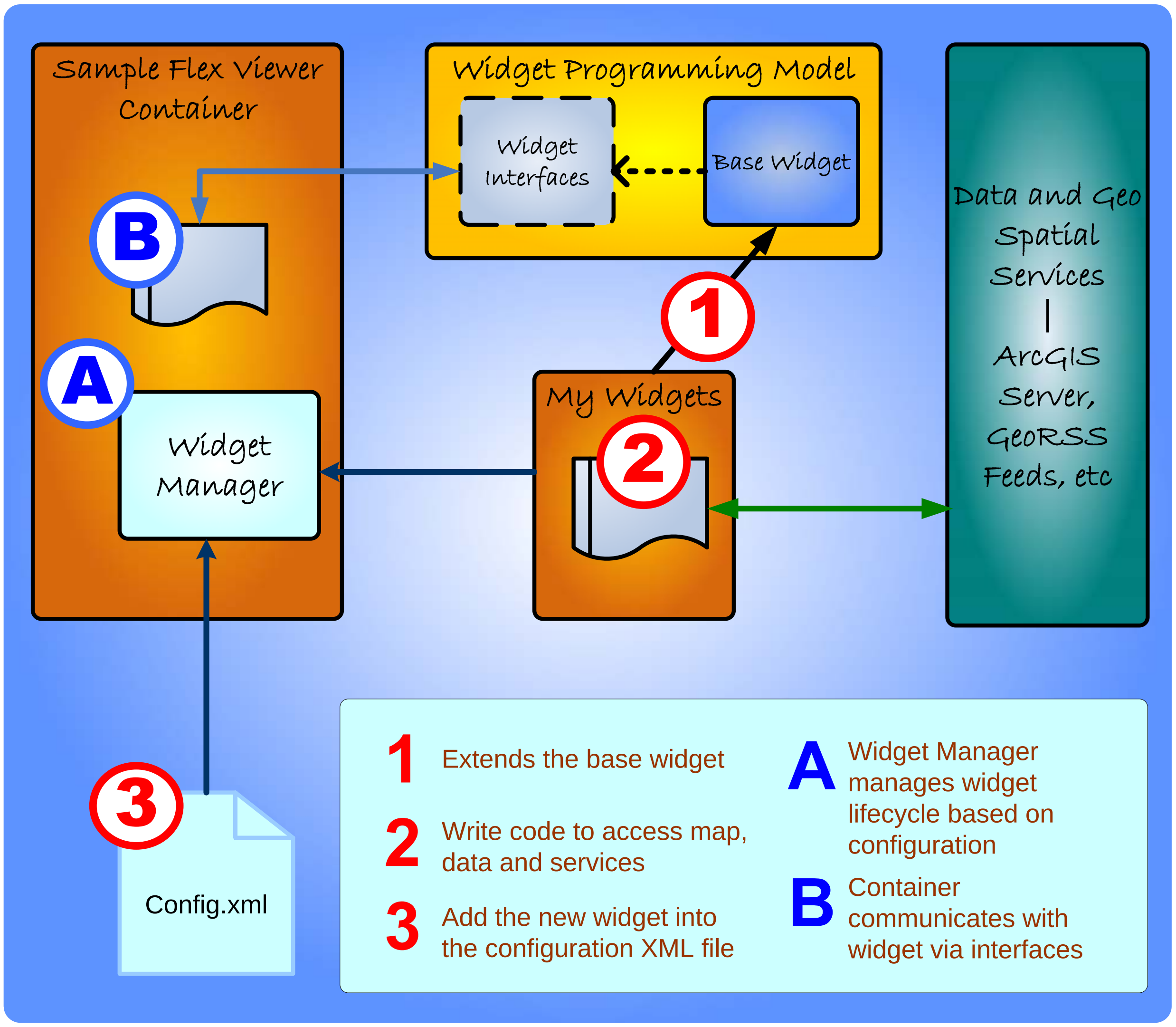
A compiled widget of the Sample Flex Viewer is a standalone SWF (Flash) file that can be shared, moved and deployed in the Sample Flex Viewer application.

Typically, a widget encapsulates a set of isolated and focused business logic that allows users perform a task. In a service oriented environment, a widget represents a service or an orchestration of services with which users can clearly execute a business function. A widget not only provides a visual interface for the user, but could also connect to server side resources such as Map Services from ArcGIS server or ArcGIS online.

A set of correlated widgets plus a clearly defined business workflow applicable to these widgets forms a business solution. The solution, furthermore, can be deployed to participate in an enterprise wide business process.

The lightweight Widget Programming Model that comes with the Sample Flex Viewer allows developers to easily develop custom widgets without having to deal with the low level coding required to integrate their widgets into the Sample Flex Viewer application.

The following diagram shows the steps to develop a widget for the Sample Flex Viewer.


Sample Flex Viewer Container
B
A
Widget Manager
Widget Programming Model
Widget Interfaces
Base Widget
1
My Widgets
2
Data and Geo Spatial Services
ArcGIS Server, GeoRSS Feeds, etc
3
Config.xml
1 Extends the base widget
2 Write code to access map, data and services
3 Add the new widget into the configuration XML file
A Widget Manager manages widget lifecycle based on configuration
B Container communicates with widget via interfaces

The Widget Programming Model contains two ActionScript classes (of which one is MXML) and two ActionScript interfaces. A later chapter will cover details on how to use these classes and interfaces.

IBaseWidget Interface (IBaseWidget.as)
This interface defines the communication methods used by the *WidgetManager* to manage widgets. The base widget class *BaseWidget* implements this interface.

BaseWidget Class (BaseWidget.as)
This is the base widget class that all widgets should be derived from. By extending the *BaseWidget* class, either a new MXML or ActionScript class will be recognized by the Sample Flex Viewer's *WidgetManager* as a deployable widget. In addition, extending the *BaseWidget* enables the new widget to be compiled into a stand alone SWF file.

A Sample Flex Viewer widget **must** extend the *BaseWidget* class.

IWidgetTemplate (IWidgetTemplate.as)
This interface defines the common operations that a widget template requires to be recognized by the *BaseWidget*. Having a widget template or using the built-in widget template is optional. The built-in *WidgetTemplate* class implements *IWidgetTemplate*.

WidgetTemplate (WidgetTeamplate.mxml)
This is the built-in widget template. A widget template is a UI component that provides basic UI layout and behaviors for a widget. These include a styled window panel, title bar with custom image buttons, etc. By using the widget template, the widget developers can spend more of their development time on their core business requirements. The built-in widget template is also a good example for widget developers to create their own widget templates, which should implement the *IWidgetTemplate* interface.

2.5 Widget Naming Convention

Widget class: Recommend that a widget class has suffix “Widget”, for example, *SomethingWidget.mxml.*

Widget Configuration File: For best practice, name the configuration file with the same base name, but use the *.xml* extension, for example, *SomethingWidget.xml.*

3 Setting up a Flex Builder Project for Widget Development

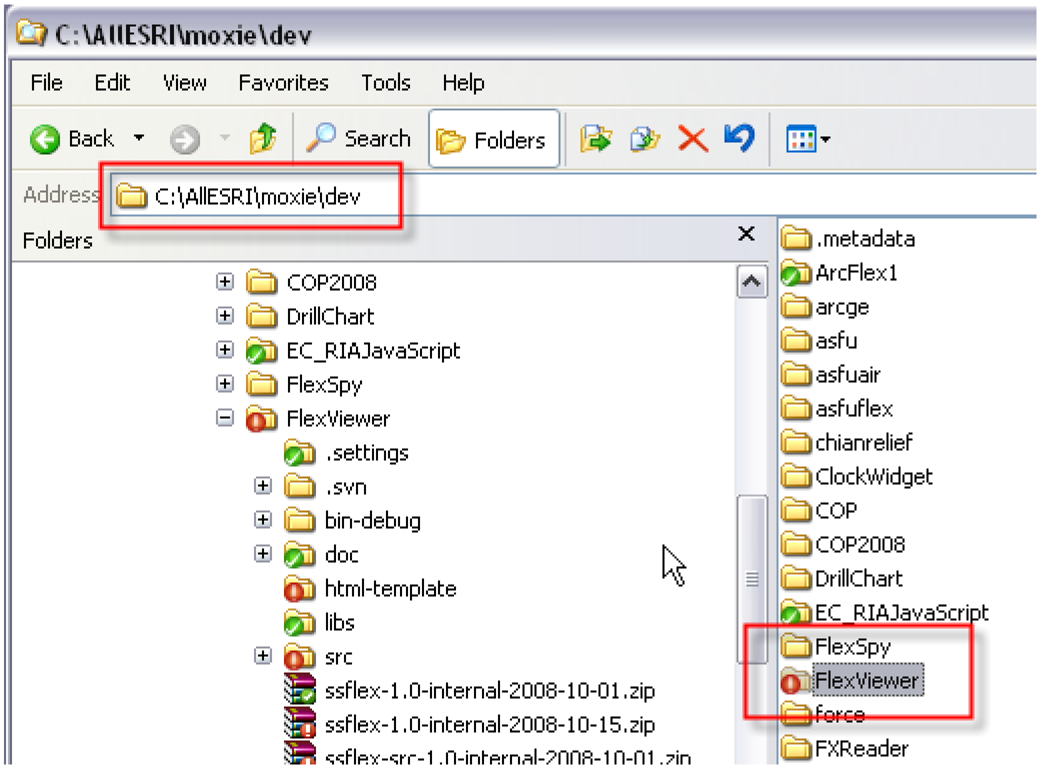
Even though you can deploy and run the compiled Sample Flex Viewer out-of-the-box, there are many use cases where you might want to access the full Sample Flex Viewer source code. For instance, you may want to modify the Sample File Viewer core functionality or you may want to extend the Sample Flex Viewer by developing custom widgets. This chapter shows you how to create a Flex project with the Sample Flex Viewer source code and compile the Sample Flex Viewer.

With the source files for the Sample Flex Viewer come Flex Builder project files to make it easy to get started with this code in Flex Builder. Below are detailed steps to unzip the source code and import it into Flex Builder.

Step 1: Unzip the source distribution package.

The zip file, *flexviewer-src-1.0.zip* (included in the release package *flexviewer-1.0.zip*), contains the root folder *FlexViewer*. Therefore, you **don't** need to manually create the root folder for the project unless you want to use a different folder name. Pick a folder on your hard drive to place the source file and unzip the source zip file to it. As shown on the screenshot as the example, the source was unzipped to

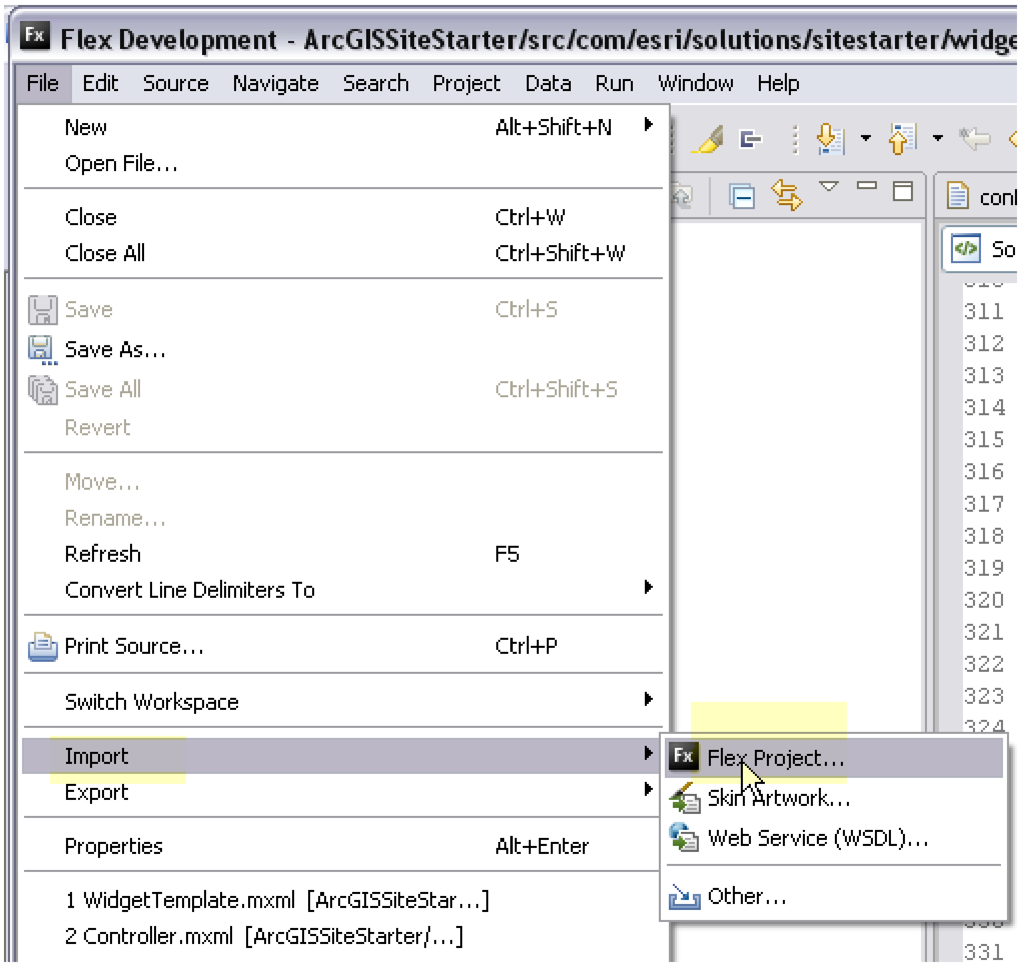
C:\ALLESRI\moxie\dev.



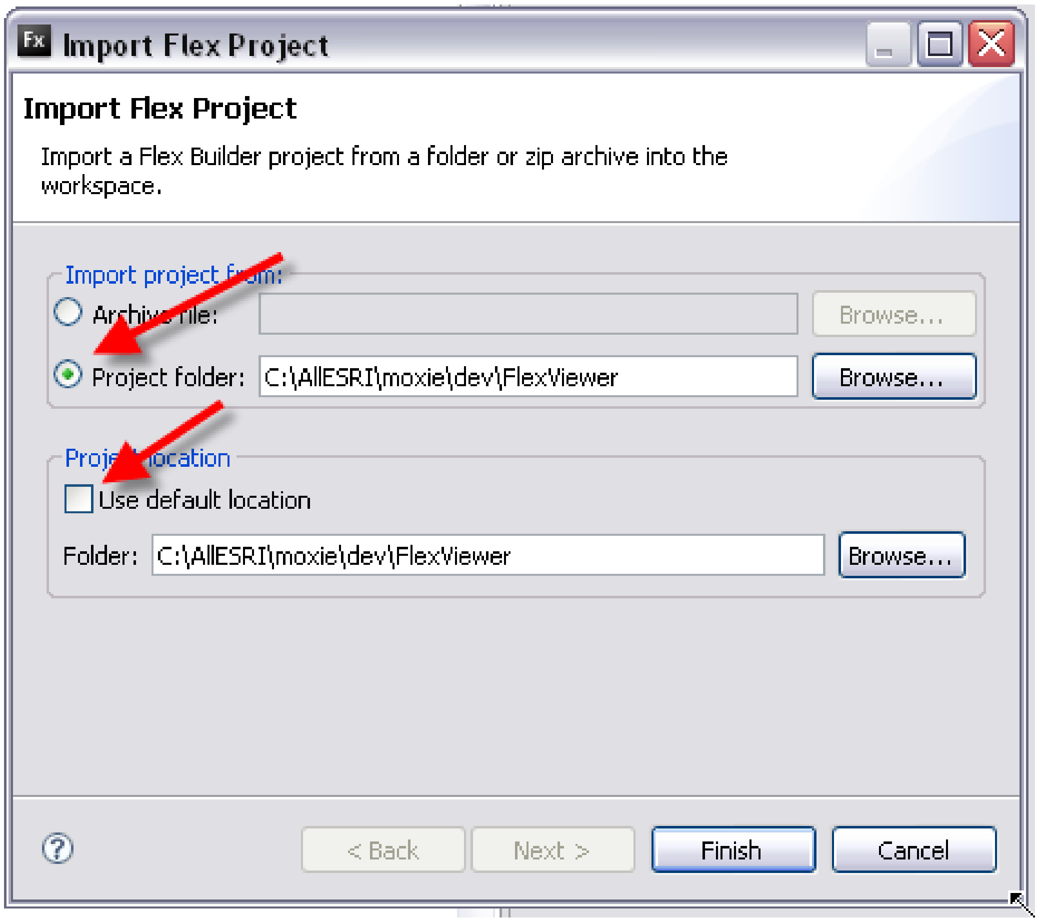
There are two approaches to using Flex Builder to create a widget. The later sections will cover these approaches in further detail.

Step 2: Import the source project into Flex Builder project

From within Flex Builder, Click File and pick Import as shown below:



Then, on the import wizard dialog, find the source directory to import from and click Finish, as shown below:


Import Flex Project

Import Flex Project

Import a Flex Builder project from a folder or zip archive into the workspace.

Import project from:

○ Archive file:

◉ Project folder: C:\AllESRI\moxie\dev\FlexViewer

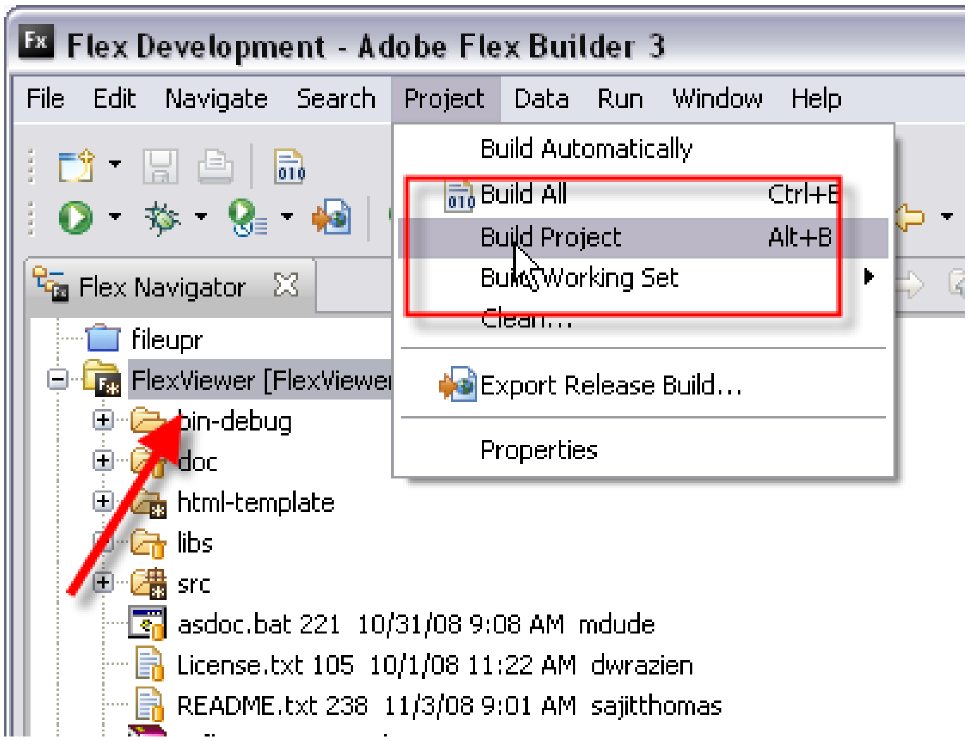
Project location

☐ Use default location

Folder: C:\AllESRI\moxie\dev\FlexViewer

Step 3: Build the Sample Flex Viewer

Now you just need to build the Sample Flex Viewer. Select the project in Flex Navigator so it is highlighted, then from the Flex Builder Project menu, select Build Project. Please note that the words “compile” and “build” have been used as interchangeable terms in this document.


Flex Development - Adobe Flex Builder 3
File Edit Navigate Search Project Data Run Window Help
Build Automatically
Build All Ctrl+B
Build Project Alt+B
Build Working Set
Clean...
Export Release Build...
Properties
Flex Navigator
fileupr
FlexViewer [FlexViewer
bin-debug
doc
html-template
libs
src
asdoc.bat 221 10/31/08 9:08 AM mdude
License.txt 105 10/1/08 11:22 AM dwrazien
README.txt 238 11/3/08 9:01 AM sajitthomas

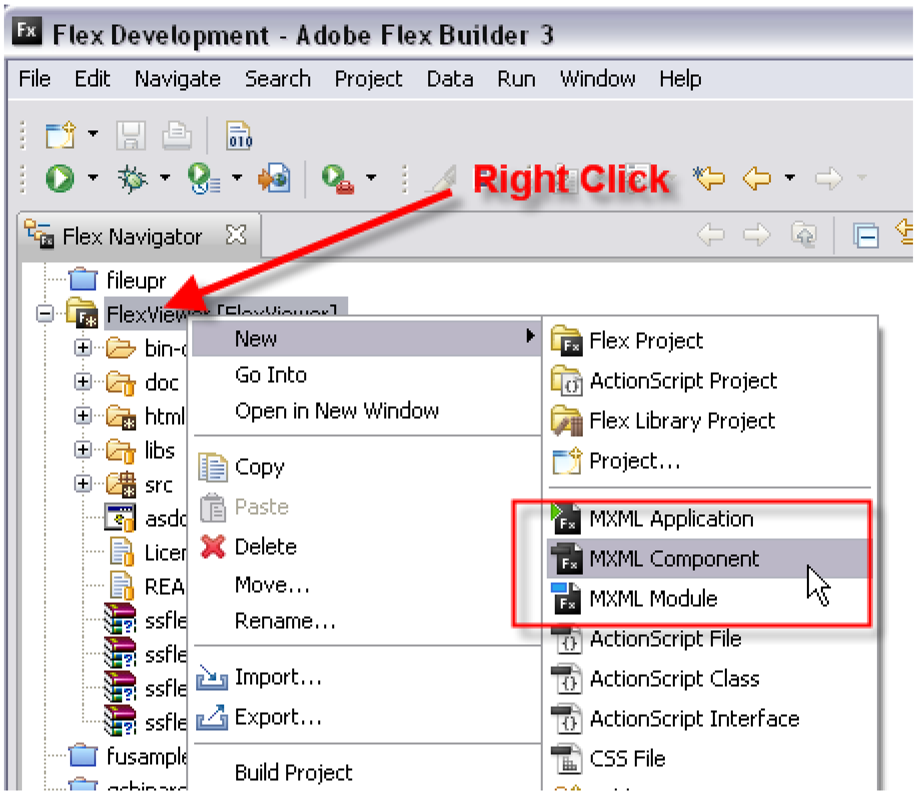
Once the project is built successfully, you are ready for the widget development.

3.1 Developing a Widget within Sample Flex Viewer Flex Builder Project

To create a widget, a new widget class can be created within the Sample Flex Viewer project *FlexViewer*. Here are the steps.

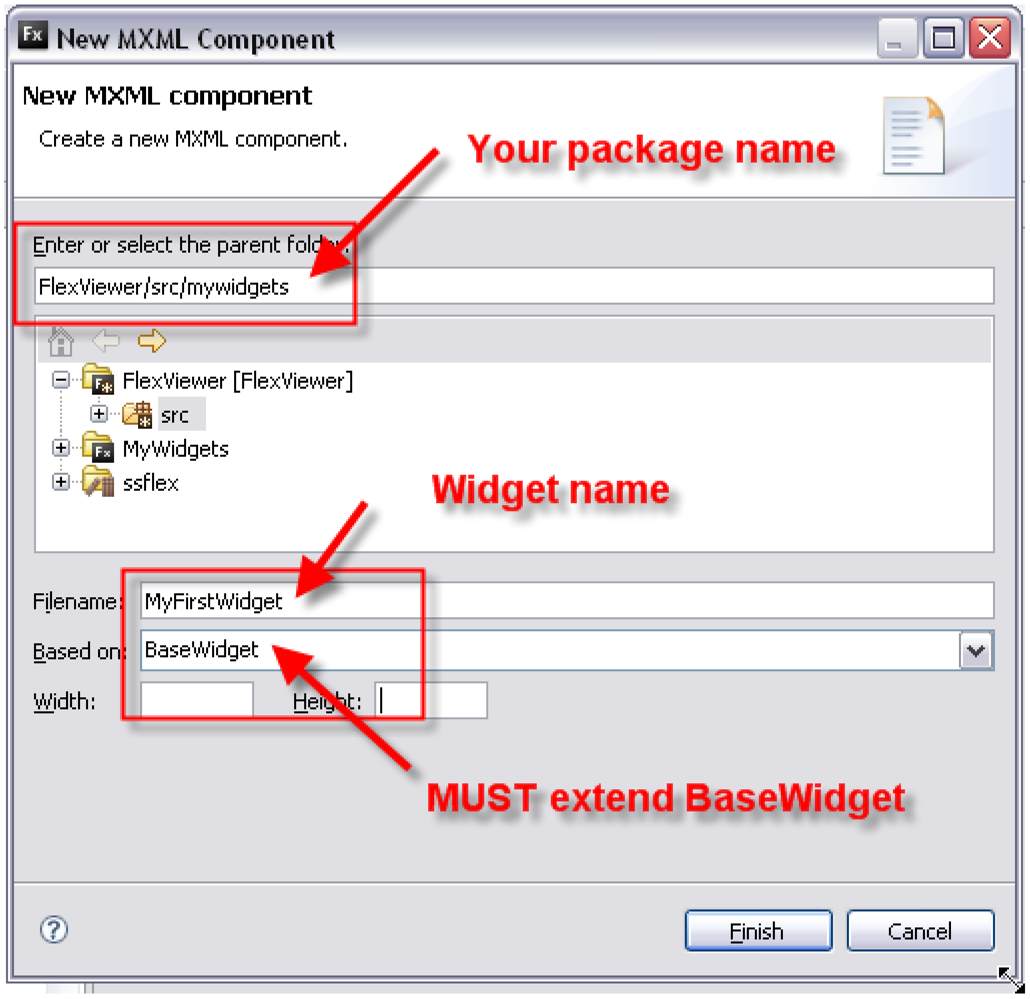
Step 1: Create a new MXML class, MyFirstWidget.mxml

Right click the *FlexViewer* project (or your own project name) in Flex Navigator and select New MXML Component.



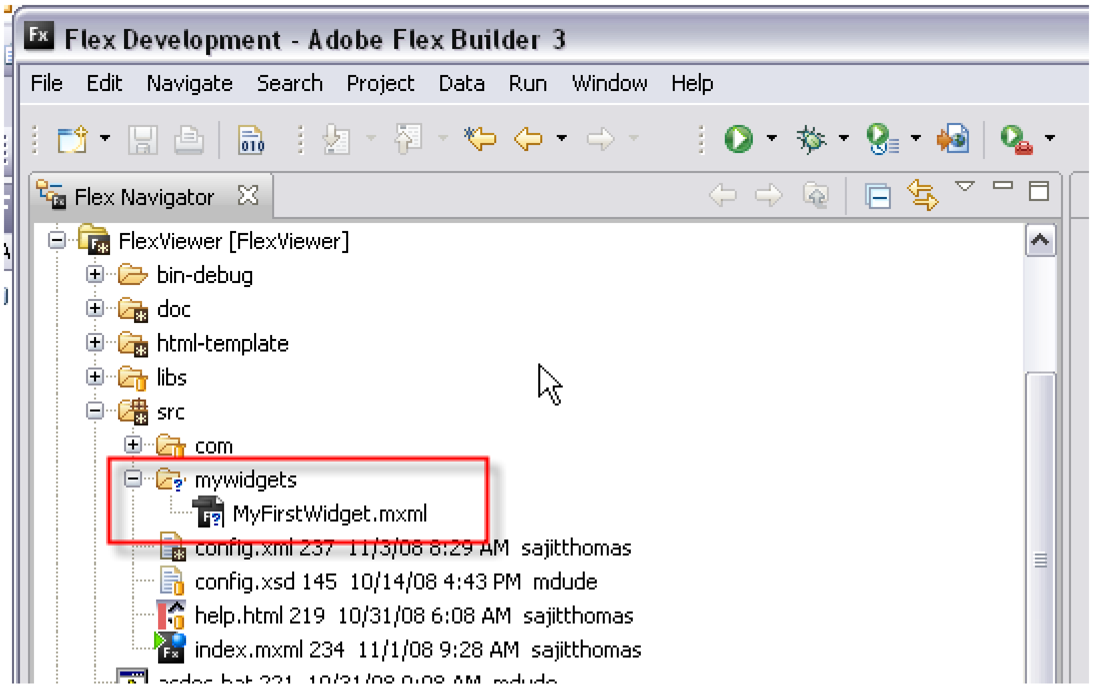
Step 2: Fill up the MXML component creation dialog

It's important that you use *BaseWidget* as your widget's base class.



Please note that you can put your new widget anywhere within the project. The best practice is to have a separate package namespace (folder) for your own widgets, as shown above.

Click Finish once you have finished providing the new widget information. A new folder is created accordingly and a new MXML file with the new widget name is created inside the folder, as show below.


Flex Development - Adobe Flex Builder 3
File Edit Navigate Search Project Data Run Window Help
Flex Navigator
FlexViewer [FlexViewer]
bin-debug
doc
html-template
libs
src
com
mywidgets
MyFirstWidget.mxml
config.xml 237 11/3/08 8:29 AM sajitthomas
config.xsd 145 10/14/08 4:43 PM mdude
help.html 219 10/31/08 6:08 AM sajitthomas
index.mxml 234 11/1/08 9:28 AM sajitthomas

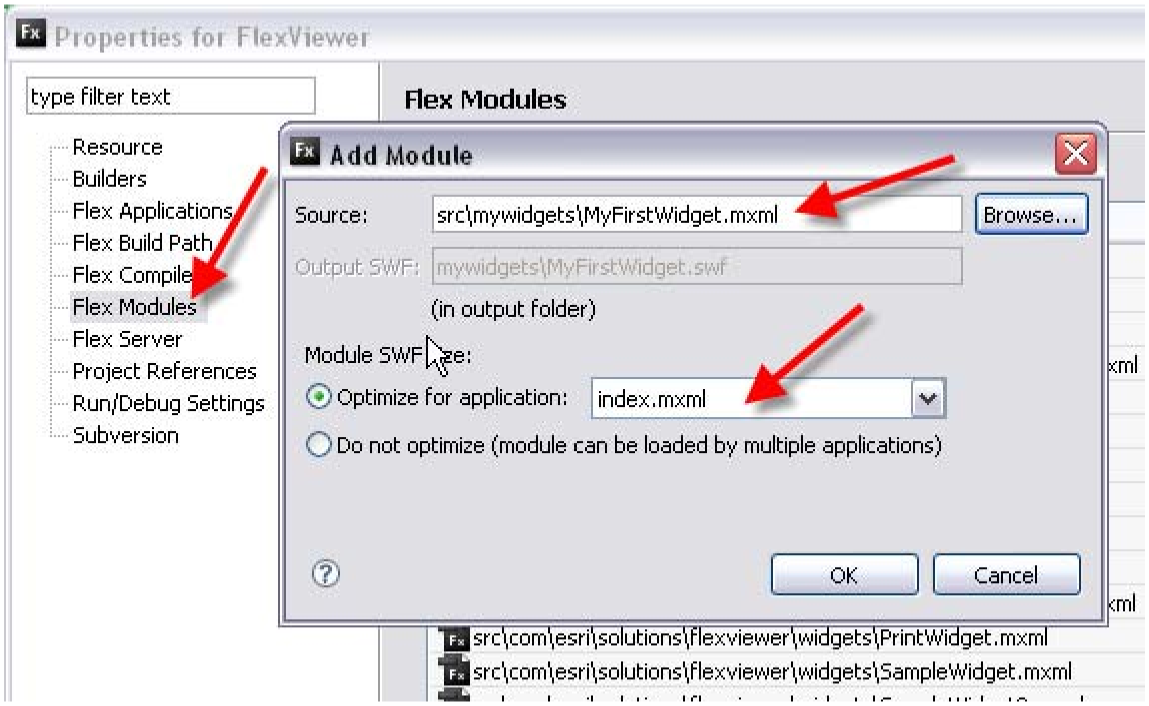
Step 3: Add *MyFirstWidget* to Flex Builder Project Module list

NOTE: This is a very critical step to allow Flex Builder to compile your widget into a stand alone SWF file.

Right click the project name and select Properties. On the project properties dialog:

- ☐ Select Flex Modules from the list on the left
- ☐ Click the Add button and use Browse button to find the new file, *MyFirstWidget.mxml*.
- ☐ Click OK.

The widget will be optimized for the Sample Flex Viewer application, represented by the *index.mxml* file.


Properties for FlexViewer

type filter text

- Resource
- Builders
- Flex Applications
- Flex Build Path
- Flex Compiler
- Flex Modules
- Flex Server
- Project References
- Run/Debug Settings
- Subversion

Flex Modules

Add Module

Source: src\mywidgets\MyFirstWidget.mxml

Output SWF: mywidgets\MyFirstWidget.swf (in output folder)

Module SWF size:

[✓ Optimize for application: index.mxml]

Do not optimize (module can be loaded by multiple applications)

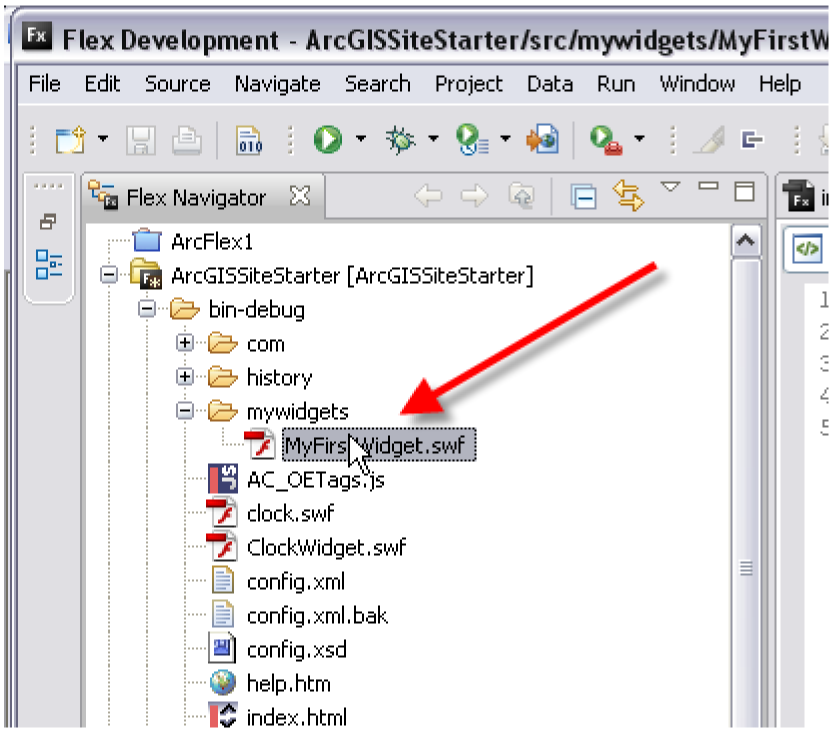
src\com\esri\solutions\flexviewer\widgets\PrintWidget.mxml

src\com\esri\solutions\flexviewer\widgets\SampleWidget.mxml

Step 4. Compile the Widget

Build this project again just like when you built the whole Sample Flex Viewer (now the new widget is part of the Sample Flex Viewer project.)

Once the build is complete, a new widget SWF file is created in the *bin-debug* folder (or its sub folder if you use a package for the widget) within the project. It is this widget SWF file that can be shared and deployed independently in the Sample Flex Viewer application. However, right now you just have an empty widget that has no UI elements or functional logic. A later chapter will show you how to further develop this widget.


Flex Development - ArcGISSiteStarter/src/mywidgets/MyFirstW
File Edit Source Navigate Search Project Data Run Window Help
Flex Navigator
ArcFlex1
ArcGISSiteStarter [ArcGISSiteStarter]
bin-debug
com
history
mywidgets
MyFirstWidget.swf
AC_OETags.js
clock.swf
ClockWidget.swf
config.xml
config.xml.bak
config.xsd
help.htm
index.html

Benefits of this approach:

- ☐ No need to create another Flex Builder project for the widget.
- ☐ When compiling, the widget will be automatically optimized for the Sample Flex Viewer application. This will result in a smaller widget file size. In turn, it will give the best performance when loading the widget at runtime.

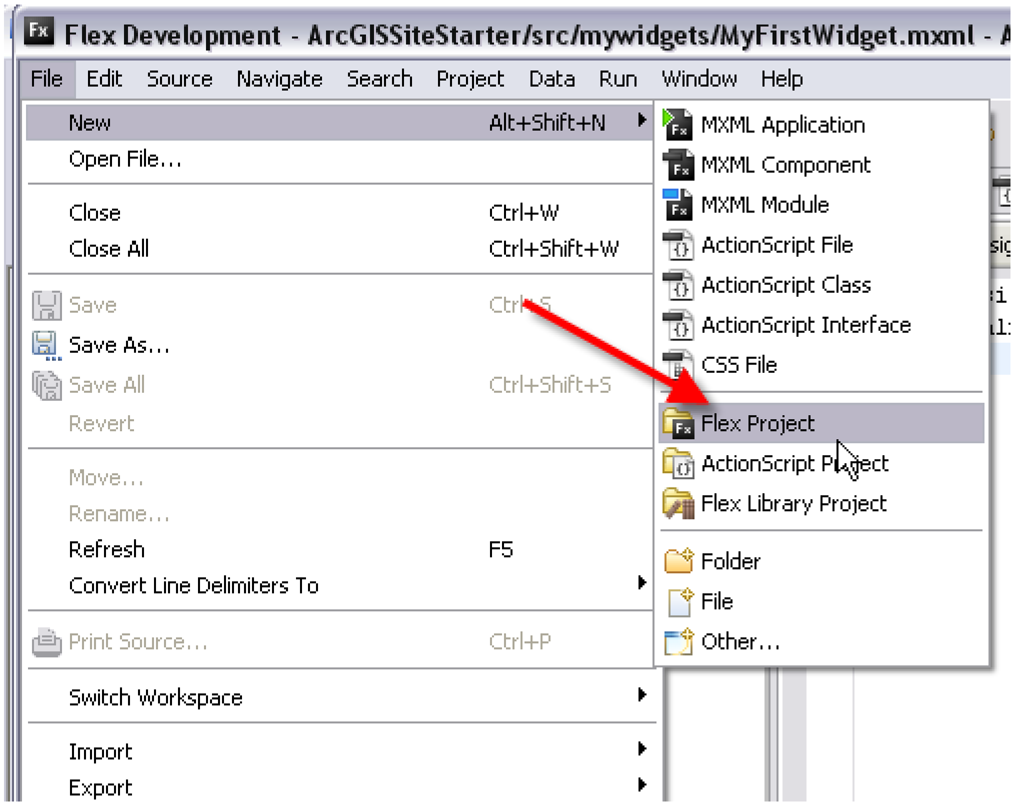
Caveat to this approach:

- ☐ You need to obtain the full Sample Flex Viewer source code and need to be able to compile it successfully.

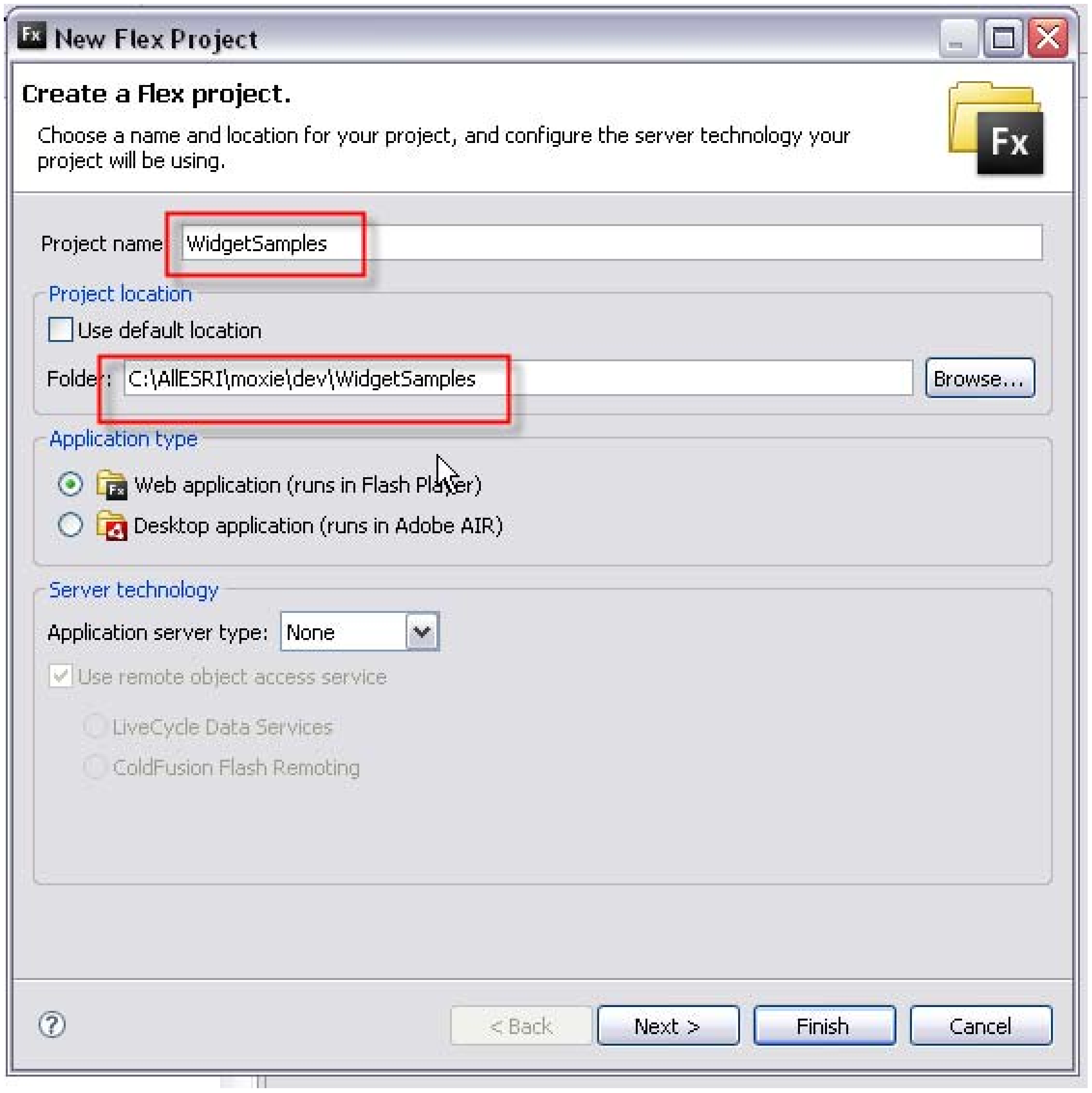
3.2 Developing a Widget outside Sample Flex Viewer Flex Builder Project

Alternatively, you can create a widget without the full Sample Flex Viewer code. You need the Widget Programming Model library, *flexviewer-1.0.swc*. You also need the Flex API library to develop an ArcGIS API for Flex application. This chapter shows you how to create a widget using the Widget Programming Model library without involving the full Sample Flex Viewer source code.

Step 1. Create a new Flex project



From the project creation wizard:


New Flex Project

Create a Flex project.

Choose a name and location for your project, and configure the server technology your project will be using.

Project name: WidgetSamples

Project location

☐ Use default location

Folder: C:\AllESRI\moxie\dev\WidgetSamples

Application type

[✓ Web application (runs in Flash Player)]

Desktop application (runs in Adobe AIR)

Server technology

Application server type: None

✓ Use remote object access service

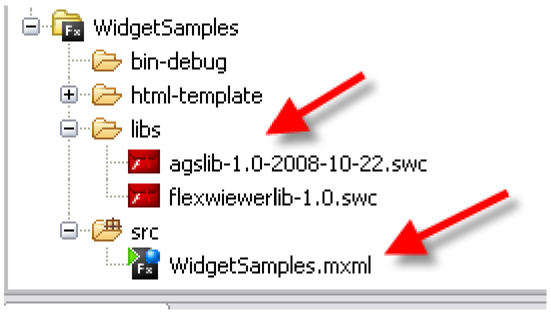
LiveCycle Data Services

ColdFusion Flash Remoting

Then, click *Next* to accept the default settings and click *Finish* to complete the wizard.

Step 2: Include the libraries for the project

By default, the Flex Builder will use all library files found in the *libs* folder within the project folder structure. Thus, all you need to do is to copy the needed library files into the *libs* folder. Alternatively, you can use library files in other locations by changing the library locations from the Project Properties wizard.



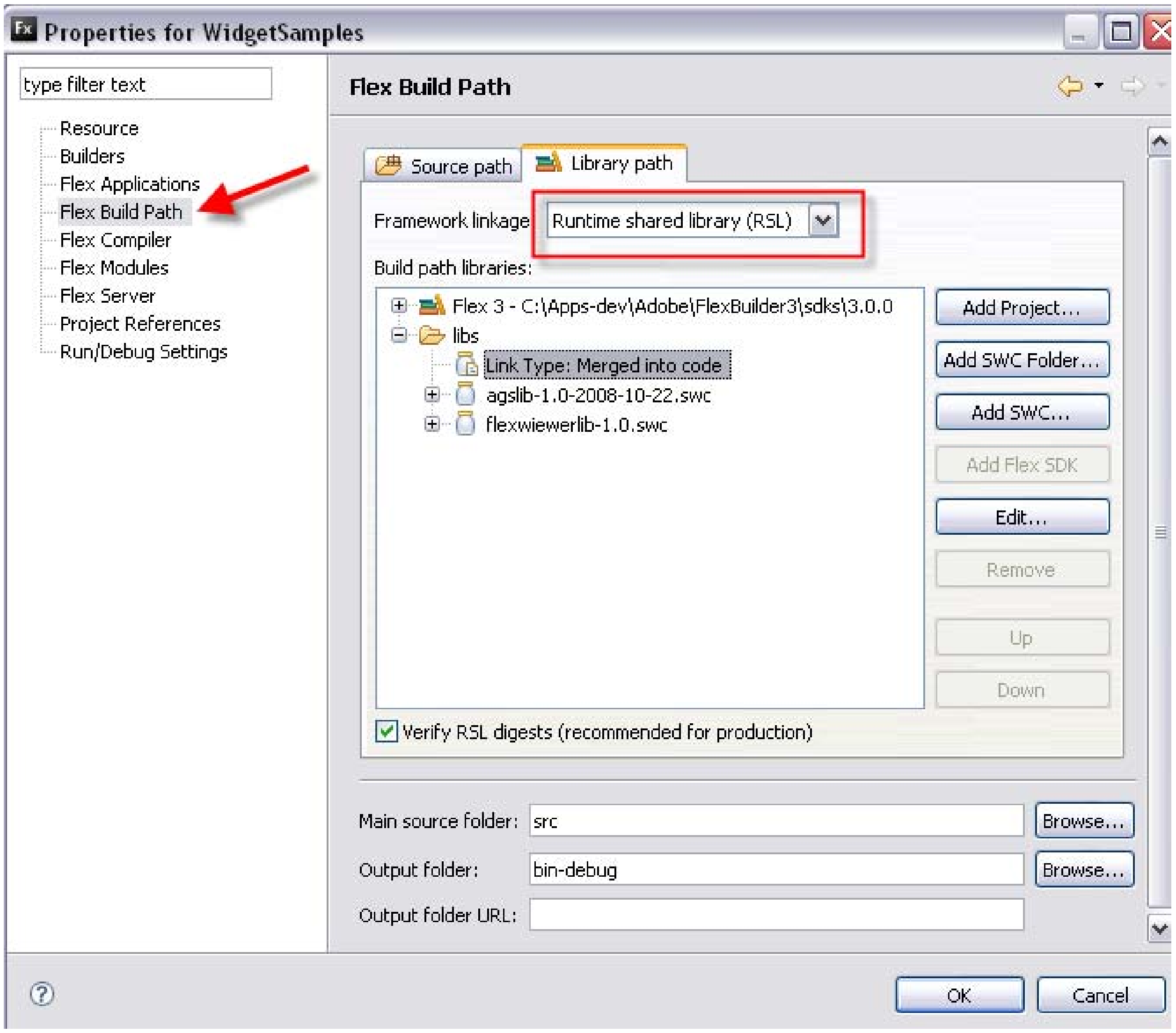
So far we copied two library files, one is the Widget Programming Model library and the other one is the ArcGIS Flex API library.

NOTE: There is an application file *WidgetSamples.mxml* created automatically by Flex Builder. Do not delete this file even though it is not used for widget development.

Step 2.5 Adjust the Compiler to Reduce Widget Size

This step tells the Flex Builder to not merge the Flex SDK library code into the compiled widget file to reduce the SWF file size.

Open the project properties:



Framework linkage: Runtime shared library (RSL)

Build path libraries:

- Flex 3 - C:\Apps-dev\Adobe\FlexBuilder3\sdks\3.0.0
- libs
  - Link Type: Merged into code
  - agslib-1.0-2008-10-22.swc
  - flexwiewerlib-1.0.swc

☑ Verify RSL digests (recommended for production)

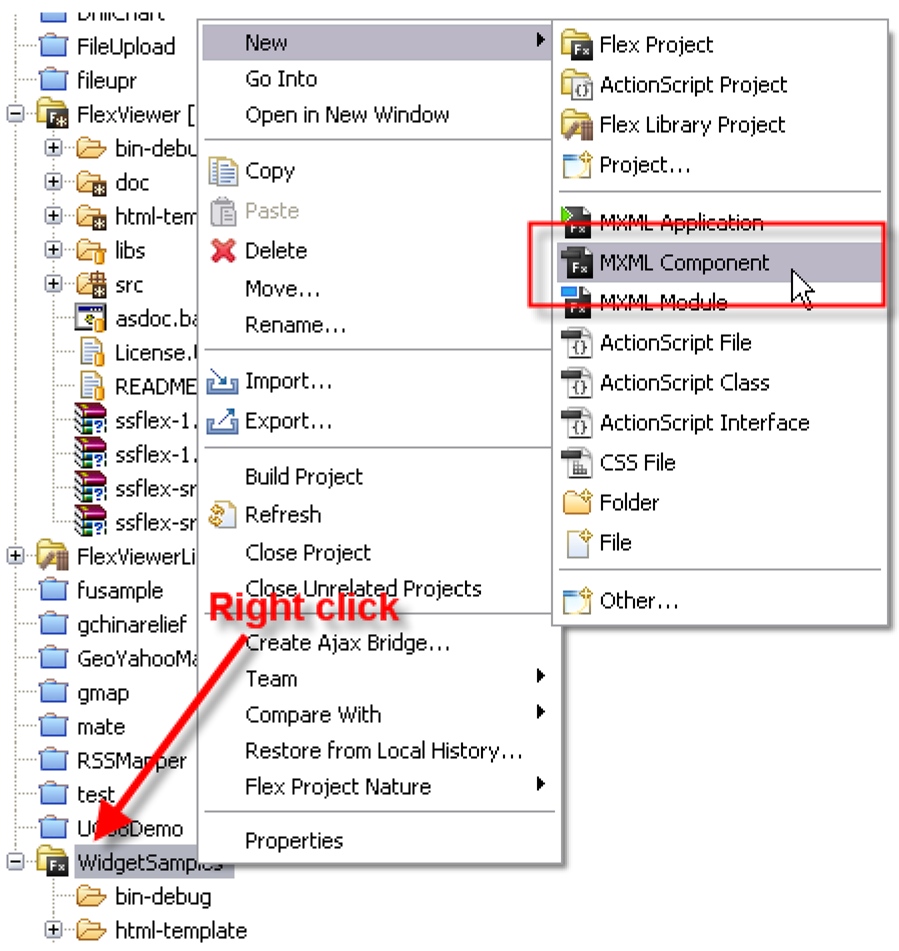
Main source folder: src

Output folder: bin-debug

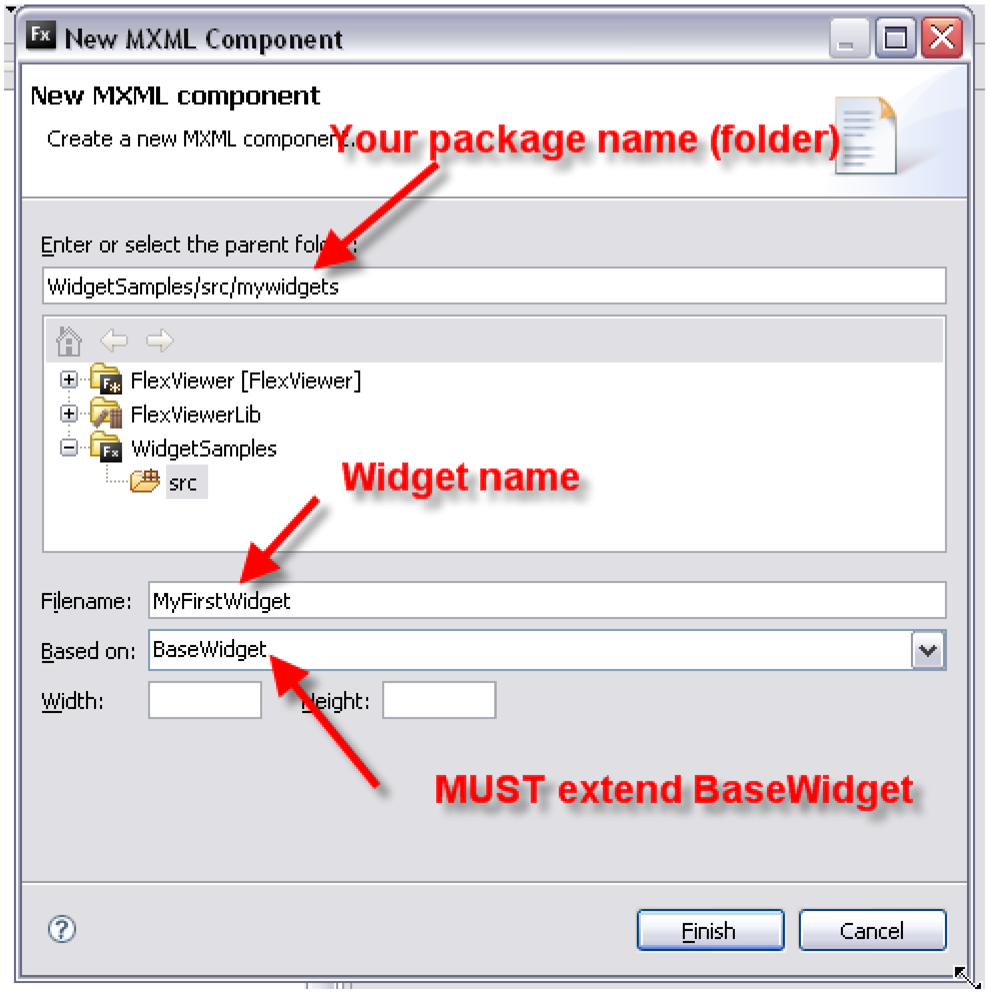
Output folder URL:

Change the framework linkage to **Runtime shared library (RSL)**.

Step 3: Create a new Widget class



On the component creation dialog:

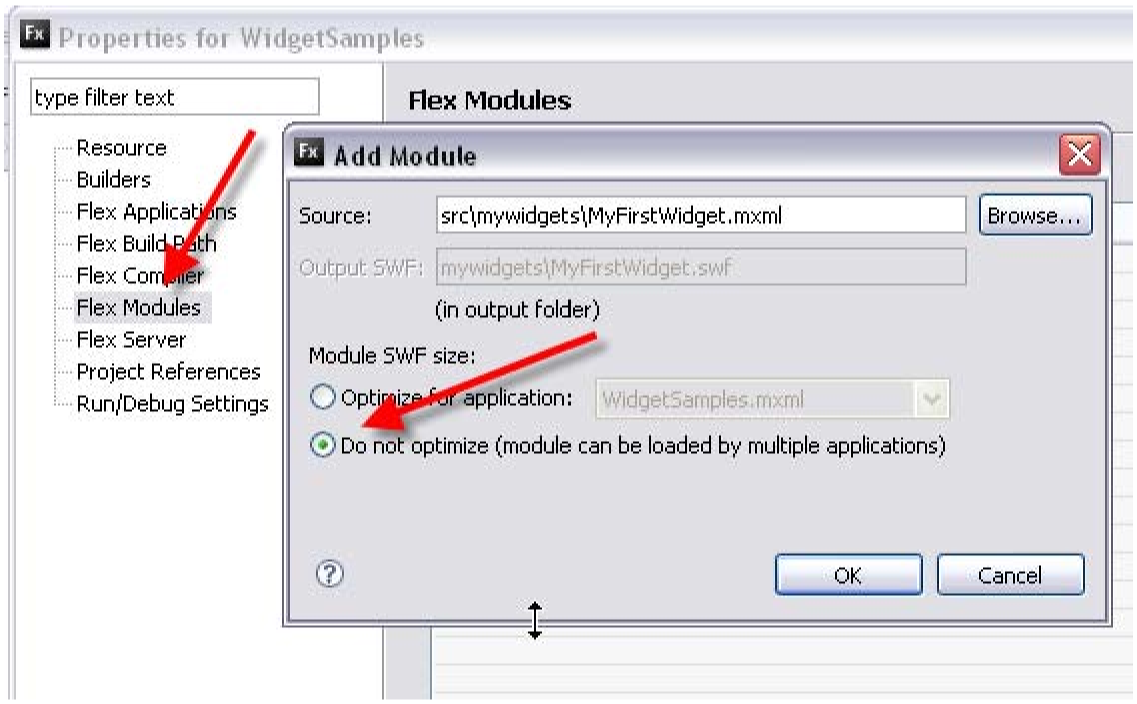


You widget **must** extend the *BaseWidget* class. Click Finish.

Step 4: Add the widget to the Flex Builder project module list

NOTE: This is a very critical step to allow Flex Builder to compile your widget into a stand alone SWF file.

Go to the project's properties window (right click project and select Properties)


Properties for WidgetSamples

Flex Modules

Add Module

Source: src\mywidgets\MyFirstWidget.mxml

Output SWF: mywidgets\MyFirstWidget.swf

(in output folder)

Module SWF size:

( ) Optimize for application: WidgetSamples.mxml

(●) Do not optimize (module can be loaded by multiple applications)

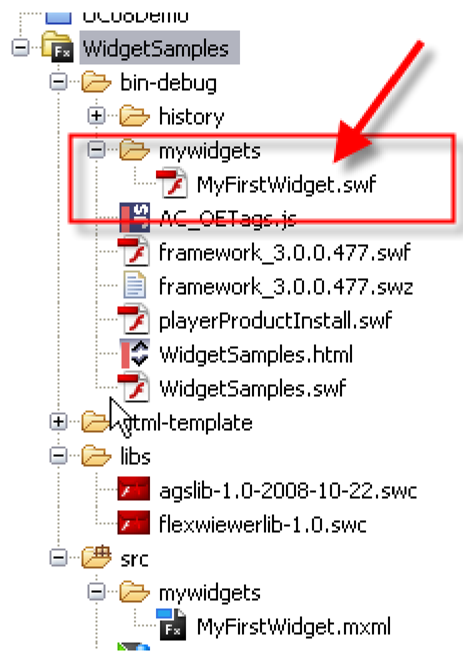
Add the newly created widget file, *MyFirstWidget.mxml* to the list. And click OK.

The code of the newly created widget looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<BaseWidget xmlns="com.esri.solutions.flexviewer.*"
            xmlns:mx="http://www.adobe.com/2006/mxml">

</BaseWidget>
```

Step 5: Build the project

Once you build this project you will find a new SWF file, *MyFirstWidget.swf*, created in the *bin-debug* folder. It's the widget SWF file that can be shared and deployed.



The current generated widget has no UI elements and no functional logic.

This shows the alternative approach to creating a widget without using the full Sample Flex Viewer souce code. A later chapter will go over the details of creating a more functional widget and deploying it in the Sample Flex Viewer applicaton.

3.3 Setting up a Test Server for Flex Builder

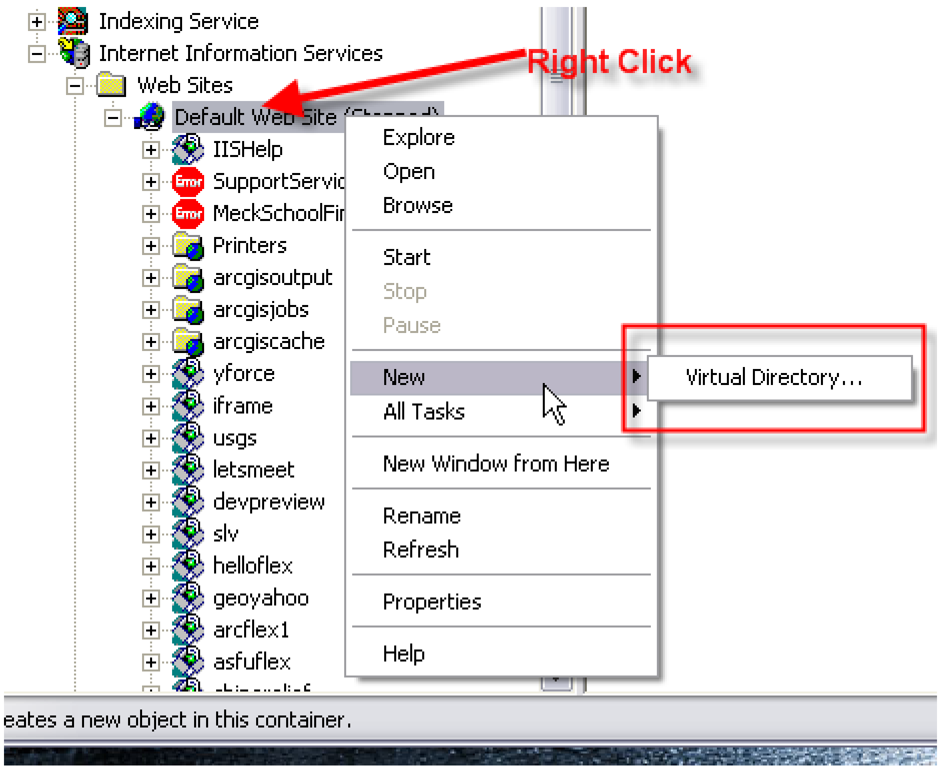
When testing a Flex application from Flex Builder, by default, it will use the default browser to bring up the HTML wrapper file directly from the file system (*bin-debug* directory). The URL will look like this: *file:///C:/AllESRI/moxie/dev/FlexViewer/bin-debug/index.html*.

However, even though the application will run, this is not the best way to test a Flex application. To simulate a deployment environment, where the Flex application will be run from a HTTP server, a local web server should be used for testing.

NOTE: The best practice is to always setup a local web server to test your Flex application.

Using the Windows development environment as an example:

Step 1: Create an IIS server virtual directory

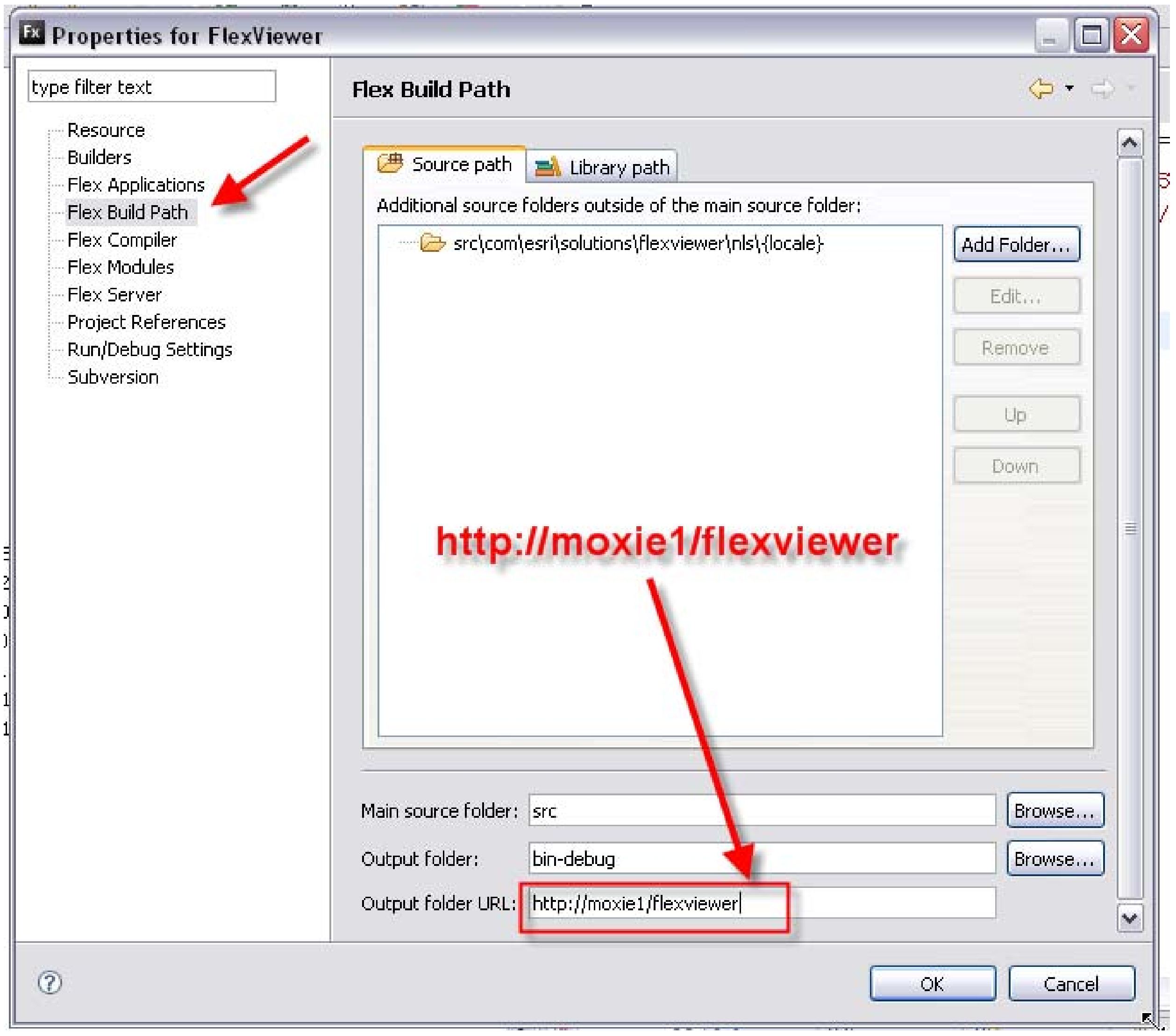


The newly created virtual directory should point to the *bin-debug* directory inside the Sample Flex Viewer Flex Builder project. That's where the generated test application is.

In the example, the virtual directory name is *flexviewer*. It means the root URL to access the virtual directory from the web server is *http://moxie1/flexviewer*. "*moxie1*" is the computer name.

Step 2: Configure the Flex Builder project to use the virtual directory

Bring up the project Properties of the Sample Flex Viewer application and at the Flex Build Path setting, as shown below, enter the new virtual directory URL to Output folder URL.



Output folder URL: http://moxie1/flexviewer

Once the output folder URL is set, the next time you run the application from the Flex Builder, it will run it from the web server path specified.

4 Developing a Widget

Even though there are two approaches to establishing a Widget development project as detailed in the previous chapter, the actual development of the core widget function is the same. In this chapter, we'll use the first approach as an example. An important assumption in this chapter is that the widget developers are familiar with Flex development.

4.1 Using WidgetTemplate

Following the steps in Section 2.1 creates a new widget file, *MyFirstWidget.mxml*. The code looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<BaseWidget xmlns="com.esri.solutions.flexviewer.*"
            xmlns:mx="http://www.adobe.com/2006/mxml">

</BaseWidget>
```

The *WidgetTemplate* that comes with the Widget Programming Model gives you a default widget window that provides a rich set of built-in features. To use *WidgetTemplate*, add two lines to *MyFirstWidget.mxml*:

```
<?xml version="1.0" encoding="utf-8"?>
<BaseWidget xmlns="com.esri.solutions.flexviewer.*"
            xmlns:mx="http://www.adobe.com/2006/mxml">

    <WidgetTemplate id="widgetTempalte">

    </WidgetTemplate>

</BaseWidget>
```

To see the results, follow the steps below to configure and deploy this widget:

Step 1: Add the widget into config.xml file

Open the config.xml file and add this line within the **<widgets>** section:

```

<widget label  = "My First Widget"
        icon   = " com/esri/solutions/flexviewer/assets/images/icons/i_globe.png"
        menu   = "menuWidgets"
        config = "";
        >
    mywidgets/MyFirstWidget.swf
</widget>

```

You can use your own icon for the widget. An icon image is a 40x40 PNG, GIF or JPG file.

Step 2: Save the files and compile the project

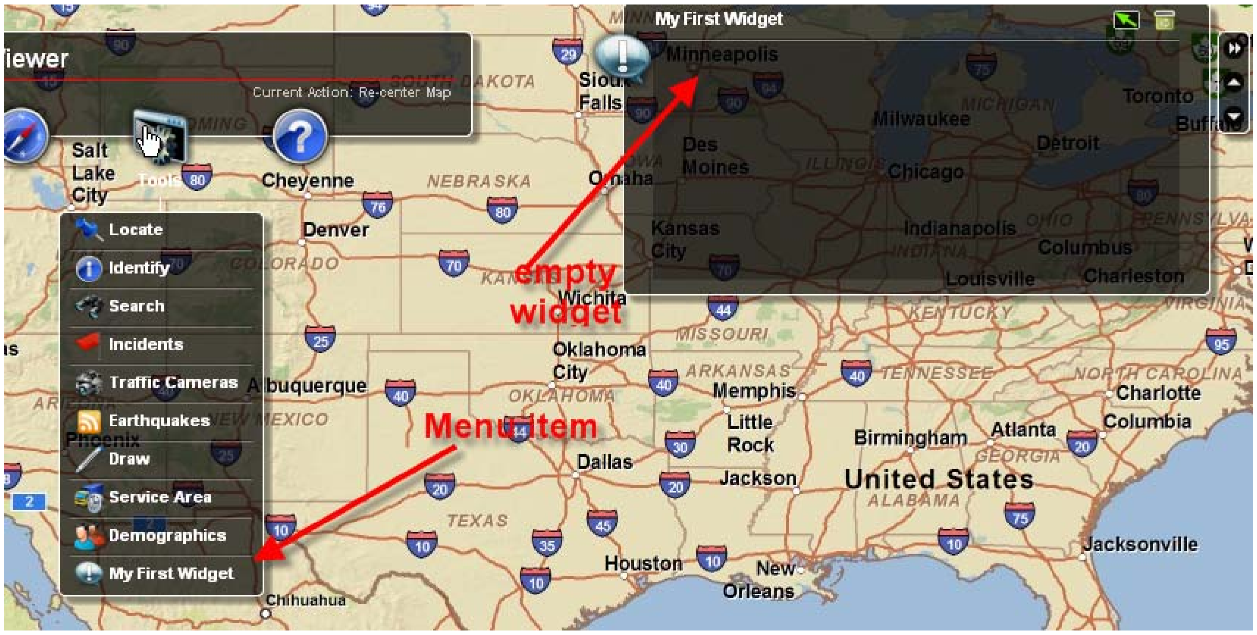
See earlier chapters on how to build the project.

Step 3: Run the Sample Flex Viewer application

Right click the *index.mxml* file in the project and click Run Application.

Step 4: From the Tool menu, click My First Widget

This will load and display your widget.



By adding two lines of code, this widget maintains a consistent look and feel and has some basic functionality including the ability to minimize, maximize and close the widget. Also, internally the widget can now communicate with the Sample Flex Viewer.

Next, you can add your own UI elements inside the WidgetTemplate and write your business logic.

4.2 Accessing a Map

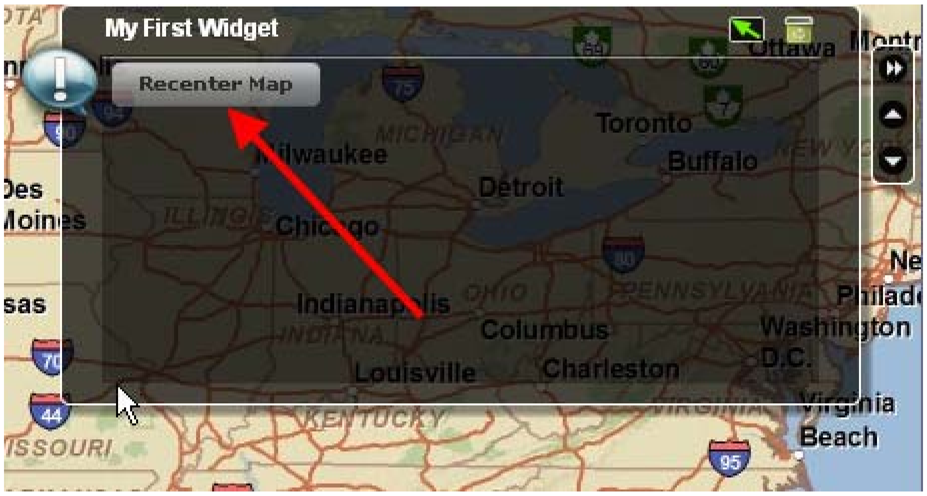
The Sample Flex Viewer is a map centric application. Once the Sample Flex Viewer application is initialized there are always one or more maps (from ArcGIS map services) that can be accessed by the widgets.

There is a public property, *map*, defined in the base widget class *BaseWidget*. After a widget is loaded, the widget manager will pass the current active map object reference to the widget. The map property of the Widget is of type *com.esri.ags.Map*, which is from the ArcGIS API for Flex. Thus, you can access all the map features provided by ArcGIS API for Flex. Widget code can be written just like writing a regular ArcGIS Flex application.

Here is an example where a button is added to *MyFirstWidget*. When the button is clicked, the map will be centered at the specified coordinates (Los Angeles).

```
<?xml version="1.0" encoding="utf-8"?>
<BaseWidget xmlns="com.esri.solutions.flexviewer.*"
            xmlns:mx="http://www.adobe.com/2006/mxml">

       <mx:Script>
              <![CDATA[
                     import com.esri.ags.geometry.MapPoint;

                     private function centerMap():void {

                            var point:MapPoint = new MapPoint(-118.24799,33.975004);
                            map.centerAt(point);
                     }
              ]]>
       </mx:Script>

       <WidgetTemplate id="widgetTemplate">
              <mx:Button click="centerMap()" label=”Recenter Map” />
       </WidgetTemplate>

</BaseWidget>
```

Run the Sample Flex Viewer application again after compiling and configuring the widget.



4.3 Displaying Widget Data on a Map

Add Graphics Layer

It’s recommended that you always create a graphics layer for each widget. The graphics layer can be created and added to the map, for example, in the *widgetConfigLoaded* event handler. The *widgetConfigLoaded* event will be dispatched after all the widget configuration information is loaded and ready to be used. Here is what the code looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<BaseWidget x="600" y="300" xmlns:esri="http://www.esri.com/2008/ags"
            xmlns="com.esri.solutions.flexviewer.*"
            xmlns:mx="http://www.adobe.com/2006/mxml"
            xmlns:mxeffects="com.adobe.ac.mxeffects.*"
            xmlns:widgets="com.esri.solutions.flexviewer.widgets.*"
            widgetConfigLoaded="init()" >


       <mx:Script>
              <![CDATA[
              …
              private function init():void
              {
                     graphicPointSym = new PictureMarkerSymbol(widgetIcon, 30, 30)
                     graphicsLayer = new GraphicsLayer();
                     graphicsLayer.symbol = graphicPointSym;
                     map.addLayer(graphicsLayer);

                     …
```

Once a graphics layer is added, you can follow the standard ArcGIS API for Flex methods of adding graphics to the layer.

Turn on/off the Widget Graphics Layer when Widget open/minimized

The *WidgetTemplate* exposes two window state events: *widgetOpened* and *widgetClosed*. Both are of type *flash.events.Event*. You can use these two events to synchronize the graphics layer's visibility.

In this example, you first add two handlers to the *WidgetTemplate*:

```
<WidgetTemplate id="wTemplate"
                   widgetClosed="widgetClosedHandler(event)"
                   widgetOpened="widgetOpenedHandler(event)">
…
```

Then define two handlers to simply toggle the visibility of the graphics layer:

```
private function widgetClosedHandler(event:Event):void
{
        graphicsLayer.visible = false;
}


private function widgetOpenedHandler(event:Event):void
{
        graphicsLayer.visible = true;
}
```

4.4 Receiving Data from Map (Click, Draw Line, etc.)

For GIS web applications, in addition to visualizing data, the map is a data source and allows users to collect data from it and process that data. There may be a need for a widget to receive point (click), polygon, polyline, etc., information generated by a user's interaction with the map. The Sample Flex Viewer and the Widget Programming Model allow a widget to request and receive such map data via the built-in method, *setMapAction*.

Here is a sample code segment that shows how to activate the draw tool for collecting a user's click point on the map:

```
private function activateTool():void
{
        setMapAction(Draw.POINT, "Click Point", drawEnd);

}

private function drawEnd(event:DrawEvent):void
{
        var geom:Geometry = event.geometry;
        var pt:MapPoint = geom as MapPoint
        Alert.show("Click location: " + pt.x + ", " + pt.y);
}
```

In the above example, the function *activateTool( )* can be called on the click operation of a button in the widget. When you are ready to obtain a map point (such as for an identify task, the user clicks the button in the widget. Then the user will click the map and the point information will be received by this widget. This is achieved by providing a call back function *drawEnd( )* as an argument to the function *setMapAction()*.

setMapAction

The public method setMapAction is defined in the *BaseWidget* class (*BaseWidget.as*) so that all widgets will have this feature by default. The method definition is as follows:

```
public function setMapAction(action:String, status:String, callback:Function):void
```

Where:

action: The string token indicates the ArcGIS Flex API's draw tool to be activated. The list of string tokens (defined in Flex API class *Draw*) which can be used to access the various draw tools provide by ArcGIS Flex API:

extent	(Draw.EXTENT)
point	(Draw.MAPPOINT)
line	(Draw.LINE)
polyline	(Draw.POLYLINE)
polygon	(Draw.POLYGON)
multipoint	(Draw.MULTIPOINT)

freehandpolyline	(Draw.FREEHAND_POLYLINE)
freehandpolygon	(Draw.FREEHAND_POLYGON)

status: Text displayed in the status area on the Controller.

callback: Call back function the Map Manager will call once the drawing is complete.

Callback function

There are two requirements for defining the callback function:

- The callback function has to be public
- The callback function has to accept the *DrawEvent* as a parameter.

The *drawEnd()* in the above example is an event handler of *DrawEvent* event. The *DrawEvent* is defined in the ArcGIS API for Flex. This event contains the geometry information collected from the map after the drawing is completed. In the above example, the *DrawEvent* provides the *drawEnd()* the point data from the user's click on the map.

4.5 Controlling Navigation from a Widget

When developing a widget, there are use-cases where you might want to control the map navigation mode in order to enable pan, zoom in, zoom out, etc.

For example, when a widget is closed, you want to make the widget's graphics layer invisible and enable the default pan mode of the map. The *BaseWidget* class provides a public method *setMapNavigation* to achieve this. For example:

```
<WidgetTemplate id="wTemplate" widgetClosed="widgetClosedHandler(event)"
…
       private function widgetClosedHandler(event:Event):void
       {
              graphicsLayer.visible = false;
              setMapNavigation(Navigation.PAN, "Pan Map");

       }
```

In the above code, you provide the *WidgetTeamplate* a handler to respond to the event when the widget is closed. Inside the event handler *widgetClosedHandler*, the public method *setMapNavigation* is called to reset the map's navigation mode to "pan".

setMapNavigation

The definition of *setMapNavigation* in *BaseWidget* is:

```
public function setMapNavigation(navMethod:String, status:String):void
```

Where:

navMethod:

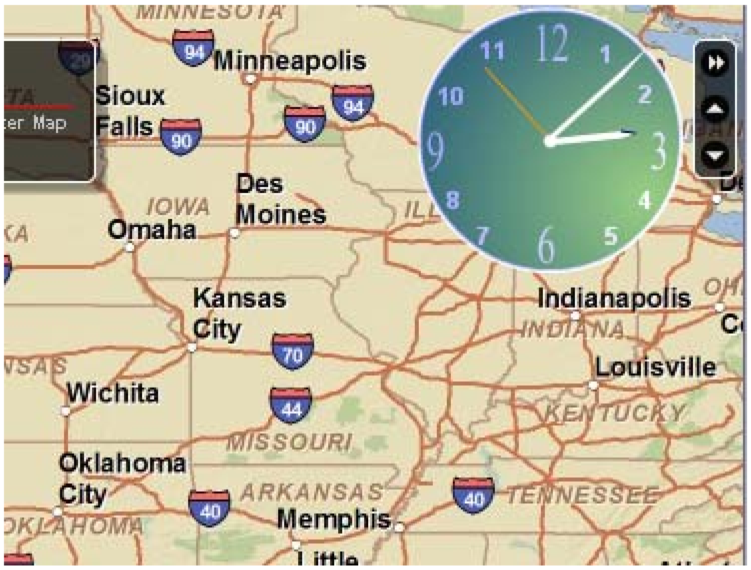
The list of map navigation strong tokens (defined in either Flex API class *Navigation* or Sample Flex Viewer class *SiteContainer*):

pan	(Navigation.PAN)
zoomin	(Navigation.ZOOM_IN)
zoomout	(Navigation.ZOOM_OUT)
zoomfull	(SiteContainer.NAVIGATION_ZOOM_FULL)
zoomprevious	(SiteContainer.NAVIGATION_ZOOM_PREVIOUS)
zoomnext	(SiteContainer.NAVIGATION_ZOOM_NEXT)

status: Text displayed in the status area on the Controller.

4.6 Developing a Widget without using Widget Template

You don't have to use the built-in *WidgetTemplate*. There may be some cases where a window based widget template is not necessary. For example, you may want to have a clock widget for your application that displays the current local time, like this:



There is no need for a window for this clock. Here is the code of this clock widget:

```
<BaseWidget xmlns="com.esri.solutions.flexviewer.*"
           xmlns:mx="http://www.adobe.com/2006/mxml"
           >


    <mx:SWFLoader source="mywidgets/clock.swf"/>

</BaseWidget>
```

In the above code, you use a SWF loader to load an external flash file that is the clock into the widget.

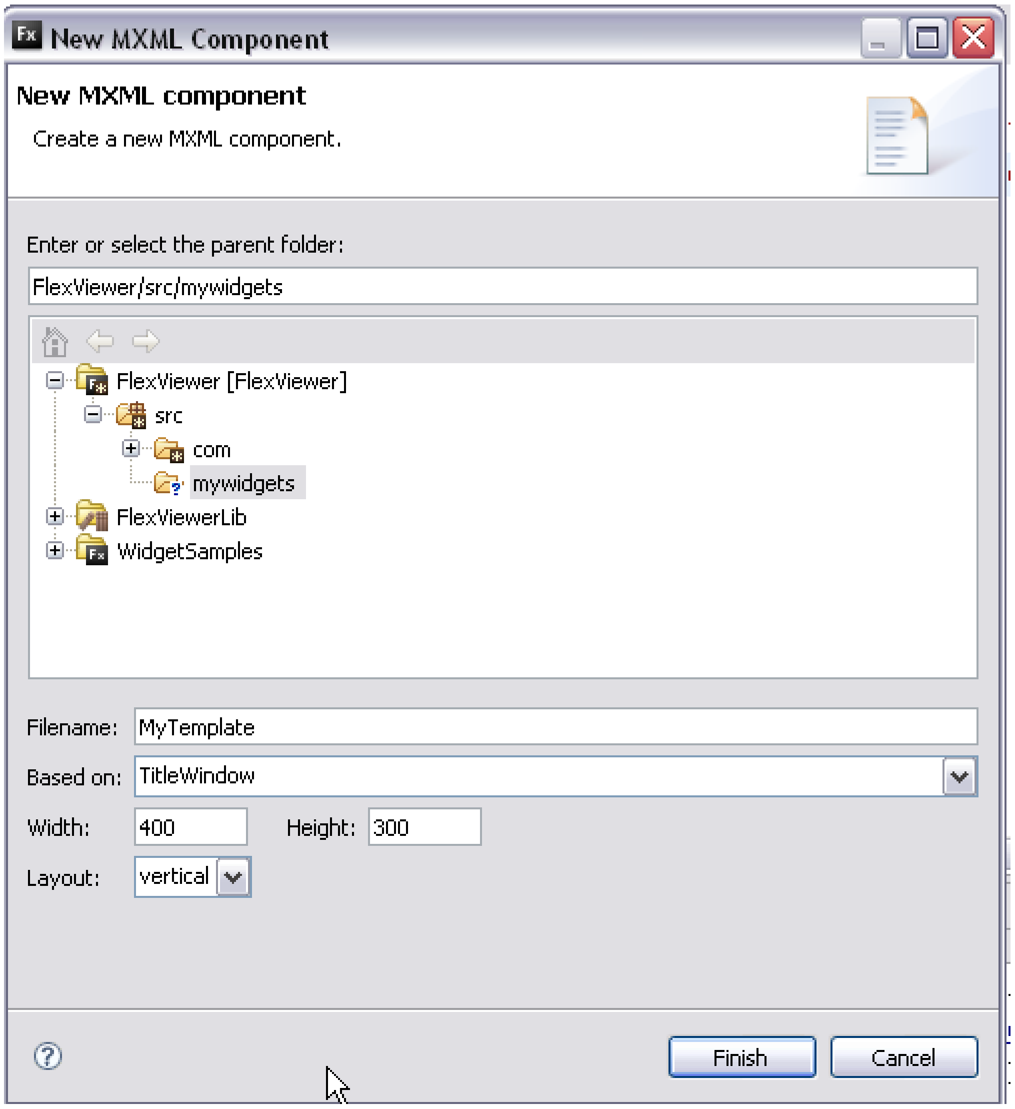
However, in this clock widget example you will notice that many of the nice features provided by the *WidgetTemplate* are missing. As a widget developer, you will need to write code to add functionality in order to make the widget really useable.

4.7 Developing a Custom Widget Template

The Widget Programming Model allows you to create a new widget template. To do that, the new widget template class needs to implement the interface *iWidgetTemplate*, which is used by the *BaseWidget* class to communicate with a widget template.

Implementing an interface means that all the methods defined in the interface must be implemented in the class. Therefore, the new widget template needs to implement three methods *setTitle*, *setIcon* and *setState*.

This example shows how to create a new widget template, *MyTemplate*, based on the Flex component *TitleWindow*:



Enter or select the parent folder: FlexViewer/src/mywidgets

Filename: MyTemplate

Based on: TitleWindow

Width: 400 Height: 300

Layout: vertical

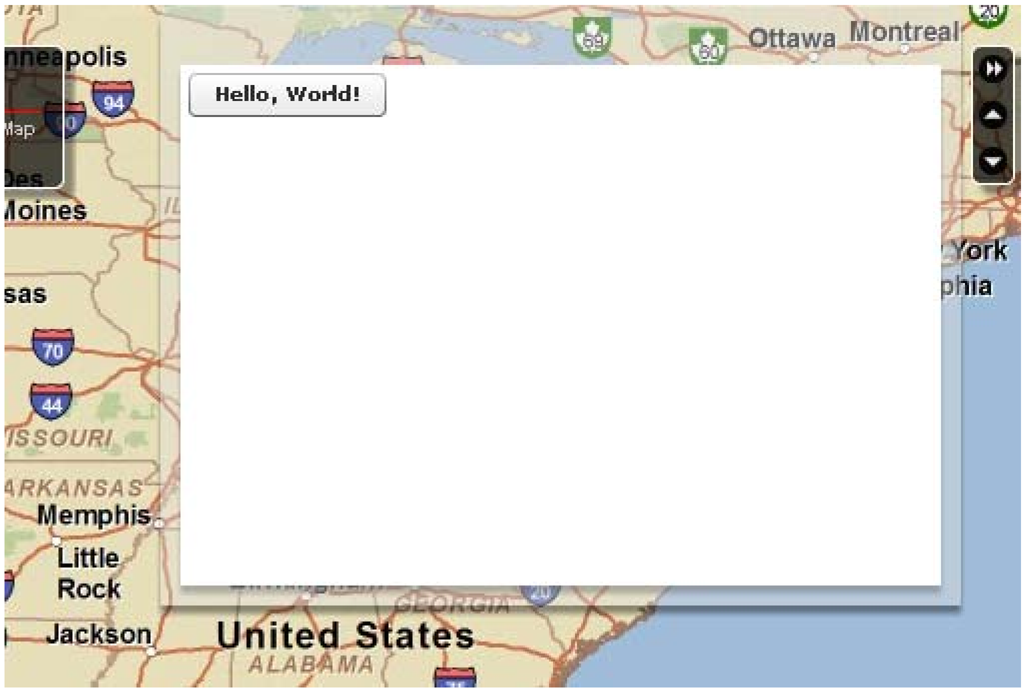
```
<?xml version="1.0" encoding="utf-8"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
                layout="vertical" width="400" height="300"
                implements="com.esri.solutions.flexviewer.IWidgetTemplate">

<mx:Script>
       <![CDATA[
        public function setTitle(value:String):void
        {
            this.title = value;
        }

        public function setIcon(value:String):void{

        }

        public function setState(value:String):void{

        }

      ]]>
</mx:Script>

</mx:TitleWindow>
```

Then add some code into the new *MyTemplate* component. The above code is the simplest widget template. It receives the title and icon and sets them in the *TitleWindow*.

In the next step we will create a new widget in *mywidgets* called *MyTempWidget*, using this new template and just place a button in it:

```
<?xml version="1.0" encoding="utf-8"?>
<BaseWidget xmlns="com.esri.solutions.flexviewer.*"
        xmlns:mx="http://www.adobe.com/2006/mxml"
        xmlns:mywidgets="mywidgets.*">

    <mywidgets:MyTemplate>
       <mx:Button label="Hello, World!"/>
    </mywidgets:MyTemplate>

</BaseWidget>
```

In the above code, we use the new widget template, *MyTemplate*. And this is what the widget will look like once it is compiled and deployed:



4.8 Modifying or Creating a New Theme

The look-and-feel of the Sample Flex Viewer application can be fully customized or changed. The theme (or skin) of the application is independent of the programming code. The built-in theme is called *Dark Angel*.

The Dark Angel theme is constructed using web standard CSS file, *com/esri/solutions/flexviewer/themes/darkangel/style.css*. The CSS file is compiled into a stand alone theme file, a SWF file, which is loaded into the application at run time from the location specified in the configuration file *config.xml*.

Inside the *config.xml*, the tag <*stylesheet*> is used to provide the URL pointer to the theme SWF file.

NOTE: The theme change from the Sample Flex Viewer application will automatically be applied to the widgets at run time.

The following steps show how to create a custom theme.

Step 1: Modify Dark Angel CSS file or create a new CSS file

Just follow the CSS standards and refer to the Flex documentation on how to create styles in a CSS file.

Step 2: Change or replace icon images

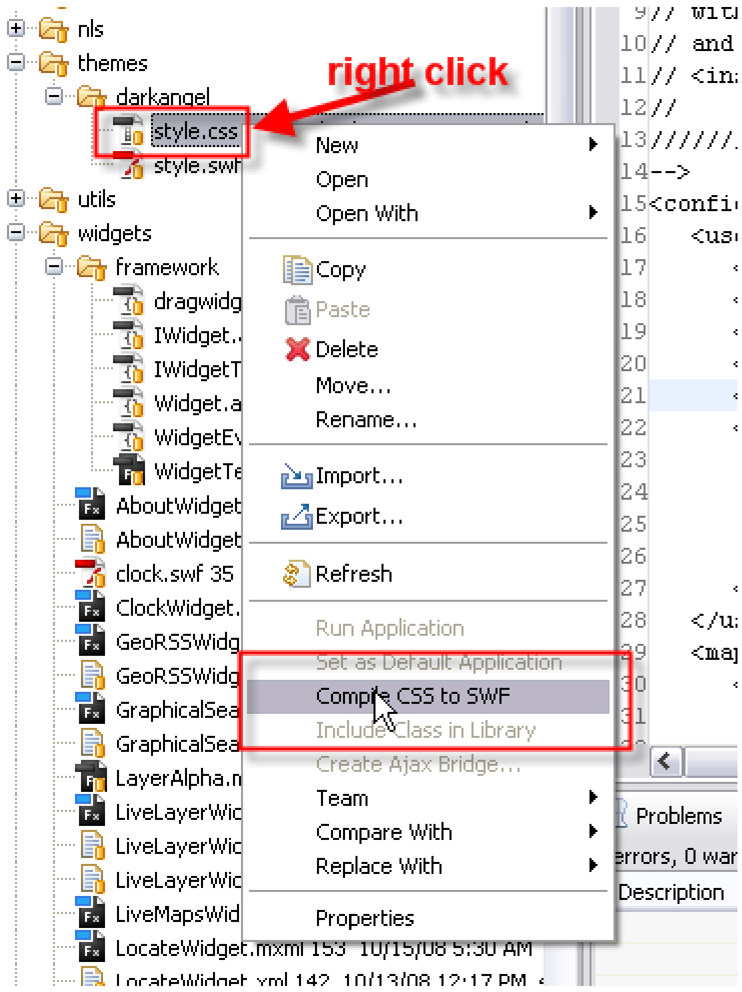
Inside the CSS file, there are some elements that use built-in icon image files. You can also change the icons used within the CSS file.

Step 3: Compile the CSS file into SWF file

This is an important step which allows the Sample Flex Viewer application to load the new theme at run time.

To enable Flex Builder to compile the CSS file, right click the CSS file and choose the option *Compile CSS to SWF* from the drop-down menu, as shown in the image below.

Once the *Compile CSS to SWF* option is selected, the next time you build the Sample Flex Viewer the CSS will automatically be compiled into a stand alone SWF file. In order to regenerate the theme SWF, you can just recompile the Sample Flex Viewer application.



Step 4: Add the URL of the new theme to Config.xml

To use the new theme, you can just modify the <*stylesheet*> tag inside the *config.xml* file to point to the new theme SWF file. The URL can be a relative path if the SWF file is deployed on the same server.

4.9 Widget Configuration

By design, each widget can have its own configuration file. The location of the configuration file is specified in the *widget* attribute inside the Sample Flex Viewer configuration file *config.xml*.

Here is an example:

```
<widget label="A New Widget" icon="urlpath/myicon.png" menu="menuWidgets"
config="youconfig.xml">relative/urlpath/MyNewWidget.swf</widget>
```

By default, the Widget Programming Model only supports the XML configuration format. The *WidgetManager* will pass the widget's configuration file URL to the *BaseWidget*. The *BaseWidget* loads and parses the configuration XML file and makes the configuration parameters available as the *configXML* (in XML ActionScript data type) as a public property.

You can access the configXML in your widget code once widget creation is complete.

Even though the out-of-the-box Viewer only supports the XML format, you can easily use your own configuration file format in your widget code. The configuration file is loaded using Flex's HTTPService connection. Therefore, the same approach can be used to load and parse other configuration data formats.

5 Working with the Sample Flex Viewer Core Code

The Sample Flex Viewer source code is freely available to the public. There are no restrictions on customizing and extending the viewer's core functionalities. Please refer to the license file included in the release package for licensing details.

The Sample Flex Viewer is designed to make it easy for developers to customize and extend. There are two design approaches that enable the high cohesion and low coupling composite application architecture, Container Event Bus and the utilization of Dependence Injection (DI). To customize the Sample Flex Viewer core, it's important to understand these two approaches.

5.1 Container Event Bus

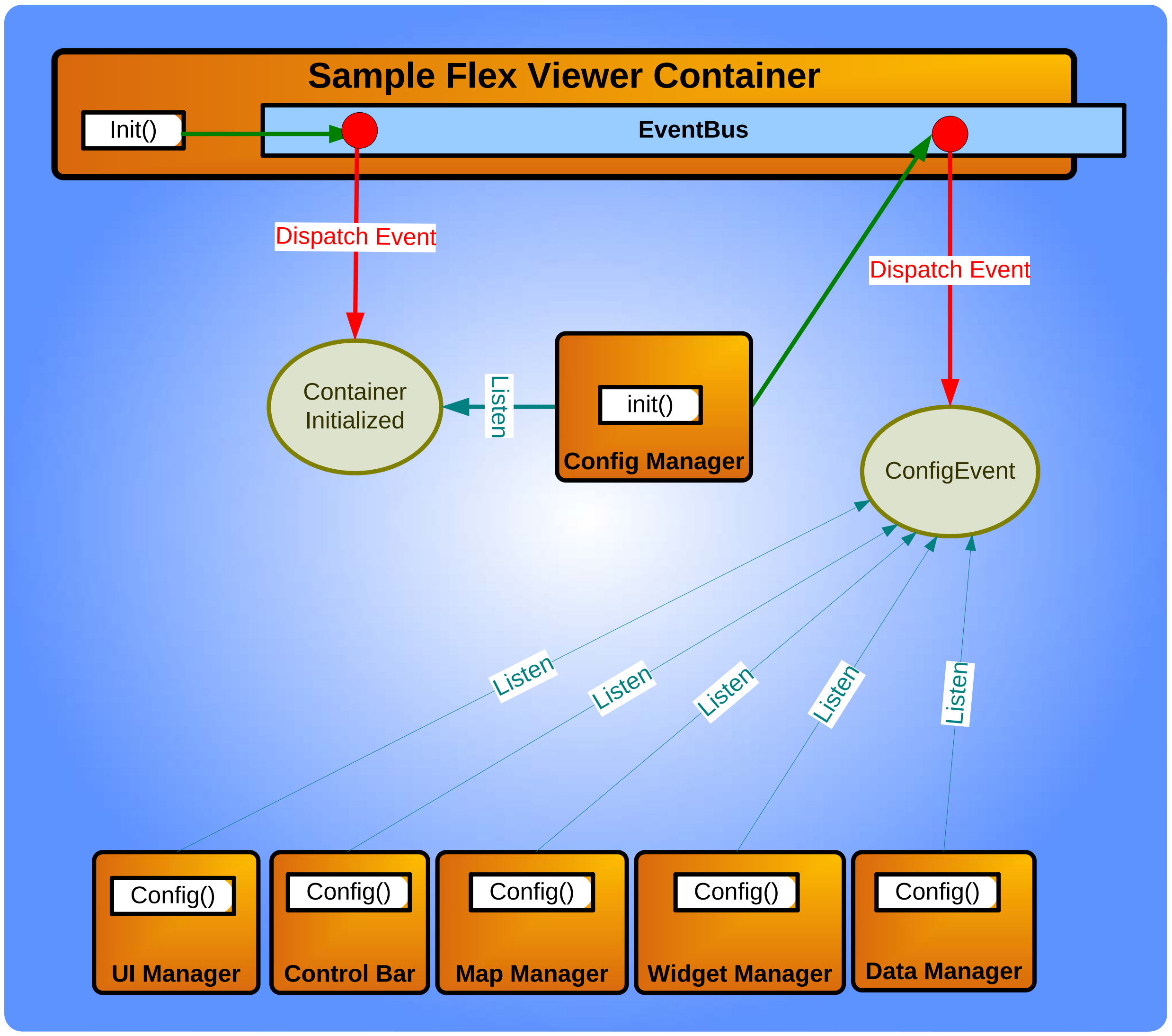
The container event bus is a global event dispatcher that is used to facilitate the communication (messaging) between different components. The event bus is defined in the *EventBus* (EventBus.as) class. A set of static proxy methods are defined inside the *SiteContainer* (SiteContainer.mxml) class to enable access to the event bus.

The following diagram shows how configuration data is loaded by the *ConfigManager* and applied to other components via the event bus.

When the *ConfigManager* is instantiated, it starts listening to an event as shown below:

```
public function ConfigManager()
{
      super();
      //make sure the container is properly initialized and then
      //proceed with configuration initialization.

SiteContainer.addEventListener(SiteContainer.CONTAINER_INITIALIZED, init);
}
```

In the above code, the *init* function is the event handler that will start loading configuration data once the event is received. The *addEventListener* is the proxy method used to access the event bus.



The event *CONTAINER_INITIALIZED* is dispatched by the SiteContainer when the container is fully initialized, as shown in the following code:

```
public function init():void
{
	_container = this;
	_lock = true; //make sure only one container is created.

	initLogging();

	//make sure the event bus is ready.
	_containerEventDispatcher = EventBus.getInstance();

	//tell the modules it's on business.

	SiteContainer.dispatch(SiteContainer.CONTAINER_INITIALIZED);
}
```

In the SiteContainer class, as shown above, once it is initialized, the event *CONTAINER_INITIALIZED* is dispatched. By using a publish/subscribe messaging approach the components can interact with each other without directly accessing the methods or data of the other components. As a result, the components are decoupled by the event based messaging. Thus, any modification done to one component will have no direct impact to the others as long as they maintain the same messages.

In the above diagram, once the *ConfigManager* finishes parsing the configuration file, it dispatches an event with the configuration data as:

```
SiteContainer.dispatchEvent(new AppEvent(AppEvent.CONFIG_LOADED,
false, false, configData));
```

All other components that need the configuration data, such as the *WidgetManager* and the *MapManager,* can simply consume the data by subscribing to this event:

```
private function init():void
{

      SiteContainer.addEventListener(AppEvent.CONFIG_LOADED, config);
                      …
}
```

This publish/subscribe coding method via the event bus can be seen through out the Sample Flex Viewer core. You can use the same approach to add components or change the way components communicate with each other.

5.2 Dependence Injection (DI)

Dependency Injection (DI) in computer programming refers to the process of supplying an external dependency to a software component. It is a specific form of inversion of control (IoC) where the concern being inverted is the process of obtaining the needed dependency.

Conventionally, if an object needs to gain access to a particular service, the object takes responsibility to get hold of that service: either it holds a direct reference to the location of that service, or it goes to a known 'service locator' and requests that it be passed back a reference to an implementation of a specified type of service. In contrast, by using dependency injection, the object simply provides a property that can hold a reference to that type of service; and when the object is created a reference to an implementation of that type of service will automatically be injected into that property - by an external mechanism.

The dependency injection approach offers more flexibility because it becomes easier to create alternative implementations of a given service type, and then to specify which implementation is to be used via a configuration file, without any change to the objects that use the service. Within the Sample Flex Viewer, the alternative implementations are the widgets. DI is also used to create custom widget templates.

The following code shows how DI is used by *WidgetManager* to load widgets into the containers without knowing the specific implementations of the widgets.

```
private function widgetReadyHandler(event:ModuleEvent):void
{
      var info:IModuleInfo = event.module;
      moduleTable.add(info.url, info);
      var id:Number = info.data.id;
      var label:String = configData.configWidgets[id].label;
      var icon:String = configData.configWidgets[id].icon;
      var config:String = configData.configWidgets[id].config;

      var widget:IBaseWidget = info.factory.create() as IBaseWidget;

      widget.setId(id);
      widget.setTitle(label);
      widget.setIcon(icon);
      widget.setConfig(config);
      widget.setConfigData(configData);
      widget.setMap(map);
      widgetTable.add(id, widget);
      var widgetDO:DisplayObject = widget as DisplayObject;
      addChild(widgetDO);
      this.cursorManager.removeBusyCursor();
}
```

In the above code, once the widget module is loaded, it will be cast into the *IBaseWidget* interface type. The *BaseWidget* extends *IBaseWidget* just for this DI usage. Once the widget is cast into *IBaseWidget* type, the *WidgetManager* doesn’t need to know what kind of widget it is or what it does. The *WidgetManager* only needs to call the methods defined in *IBaseWidget* to communicate with the loaded widget. Therefore, the implementation of a widget, regardless of how complicated it might be, is totally independent of the *WidgetManager* or the whole Flex Viewer container. The developer can freely develop a widget without being constrained by the Flex Viewer.

It's the recommended best practice to maintain such flexibility through DI when you customize the Sample Flex Viewer core. This allows you to continue to build and increase the functionality horizontally, by just adding widgets instead of adding features to the core. This approach keeps your custom application simple and lean and avoids increased complexity from added features.

5.3 Internationalization

There are differences between internationalization (aka i18n) and localization (aka l10n). Internationalization intends to support multiple languages as a choice based on user's preference. Localization usually focuses on one local language. Based on the differences, the design approaches are different.

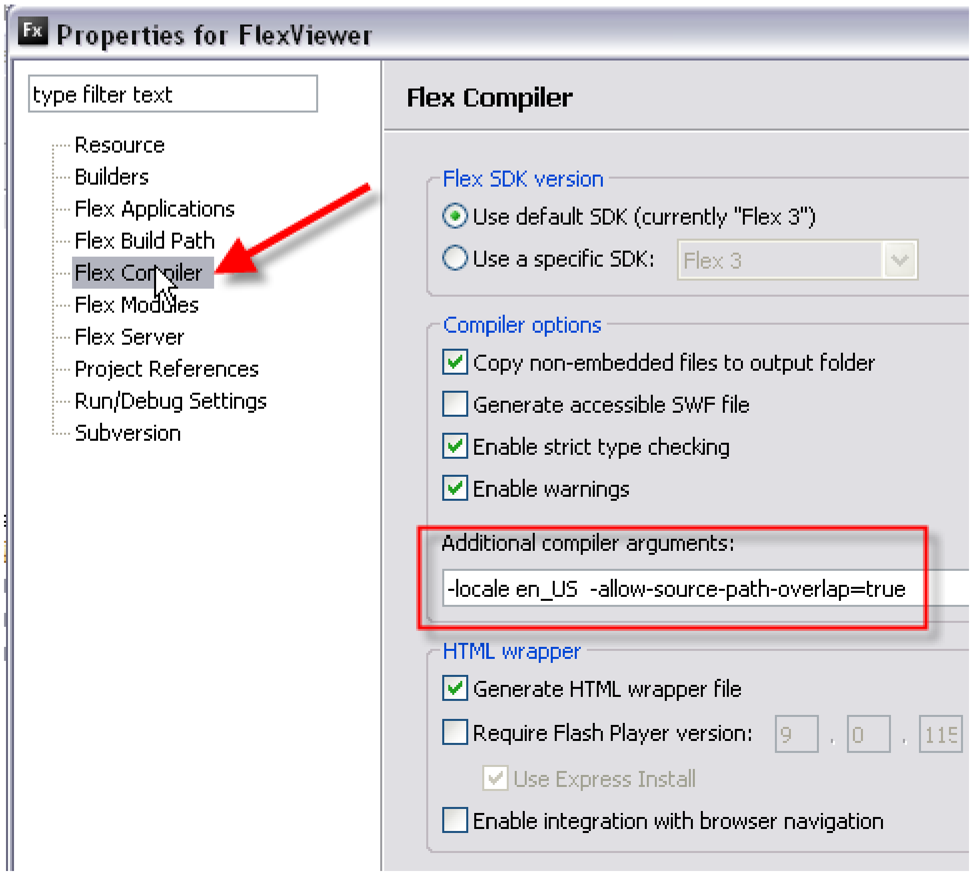
Since the Sample Flex Viewer is not designed and developed as a complete product, both internationalization and localization are provided as samples inside the code, rather than a complete implementation.

5.3.1 Use Flex Internationalization Feature

For details of how Flex supports internationalization, please refer to Flex SDK documentation. In the Sample Flex Viewer project, a sub folder, *nls*, is created to host the internationalization text strings. As an example, there are two languages provided, en_US (US English) and fr_FR (French).

Here are steps to configure the Flex Builder to support internationalization:

Step 1: Add compiler options from project Properties as follow



Flex Compiler

Flex SDK version

[✓ Use default SDK (currently "Flex 3")]

Use a specific SDK: Flex 3

Compiler options

- [x] Copy non-embedded files to output folder
- [ ] Generate accessible SWF file
- [x] Enable strict type checking
- [x] Enable warnings

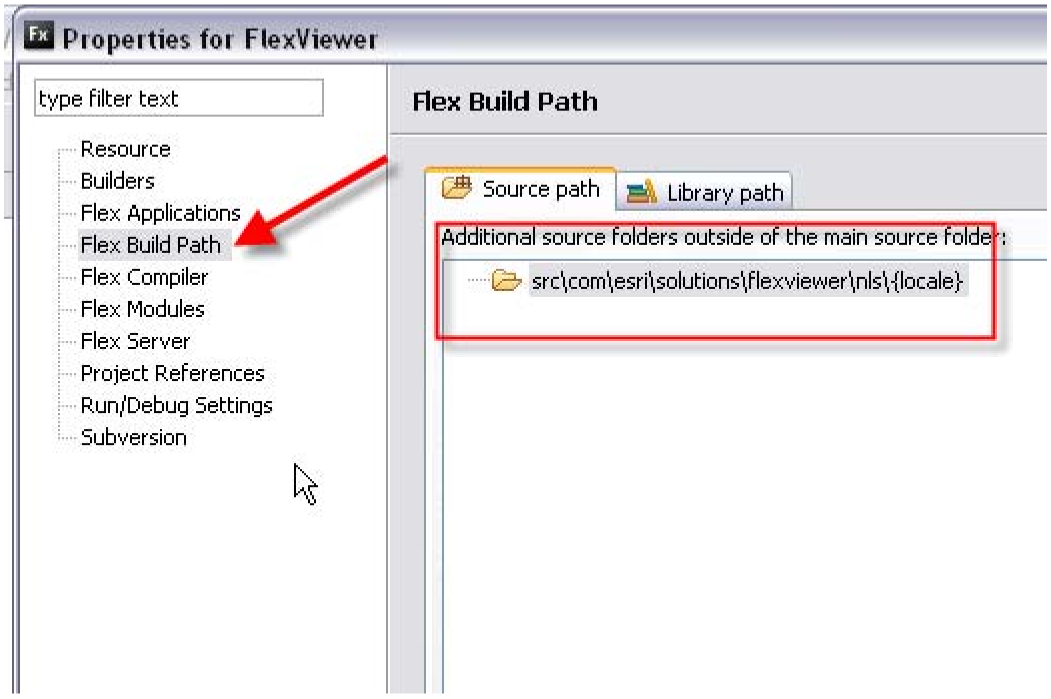
Additional compiler arguments:

```
-locale en_US  -allow-source-path-overlap=true
```

HTML wrapper

- [x] Generate HTML wrapper file
- [ ] Require Flash Player version: 9 . 0 . 115
- [x] Use Express Install
- [ ] Enable integration with browser navigation

Step 2: Add the internationalization directory to the Build Source



Flex Build Path

Additional source folders outside of the main source folder:

```
src\com\esri\solutions\flexviewer\nls\{locale}
```

The *{locale}* in the path allows the compiler to find the right language based on the *–locale* compiler option specified in step 1.

Step 3: Create resource files for the languages

For example, we create the resource file *WidgetTemplateStrings.properties* under both *en_US* and *fr_FR* folder under the *nls* folder. The Flex compiler will recognize the file extension "*properties*" by default. The string definitions inside the resource file are very simple:

```
close=Fermez
minimize=Minimisez
```

The file contains name, value pairs. The above example is for French.

Step 4: Use the resource inside the code.

To use the resource, for example in the *WidgetTemplate.mxml*, first define the resource bundle that ties to the defined resource file:

```
    <mx:Metadata>
        [ResourceBundle("WidgetTemplateStrings")]
    </mx:Metadata>
```

Then a function named *nlsString* is created to retrieve the language specific string:

```
   private function nlsString(nlsToken:String):String
   {
      return resourceManager.getString('WidgetFrameStrings', nlsToken);
   }
```

To get the string for "close", *nlsString("close")* will do.

The Flex documentation provides more details and many advanced techniques for supporting internationalization through such resource bundles.

5.3.2 Use Custom Configuration for Localization

As an alternative, using configuration file to localize the label strings is a simple approach. The Sample Flex Viewer built-in widget *LiveMapsWidget* is a good example.

With *LiveMapsWidget*, the label strings are defined in the widget configuration file *LiveMapsWidget.xml*.

```
<configuration>
      <labels>
            <visibilitylabel>Layer Visibility</visibilitylabel>
            <transparencylabel>Layer Transparency</transparencylabel>
      </labels>
</configuration>
```

In this case, if you want to override the default label, you can just edit the configuration file.

5.4 Logging and Error Handling

The combination of logging and error handling could help develop stable and good quality software.

Logging

There are many options and tools available for ActionScript/Flex application logging. It's a best practice to use managed logging for a production application. As an example, we set up logging inside the SiteContainer to log fatal error occurred during RPC and network based messaging. HTTP connection is of type RPC connection. The logging code is as follow:

```
/**
 * Initialize the logging. As an example, the logging is setup to only
 * log the fatal event during the RPC related network communication,
 * such as HTTP call to obtain configuration file.
 */
private function initLogging():void {
    // Create a target.
    var logTarget:TraceTarget = new TraceTarget();

    // Log only messages for the classes in the mx.rpc.* and
    // mx.messaging packages.
    logTarget.filters=["mx.rpc.*","mx.messaging.*"];

    // Log on fatal levels.
    logTarget.level = LogEventLevel.FATAL;

    // Add date, time, category, and log level to the output.
    logTarget.includeDate       = true;
    logTarget.includeTime       = true;
    logTarget.includeCategory   = true;
    logTarget.includeLevel      = true;

    // Begin logging.
    Log.addTarget(logTarget);
}
```

What is used here is based on Flex's logging framework. Please refer to Flex documentation for details.

Error Handling

ActionScript 3 offers enhanced error handling. However, the compiler doesn't enforce handling possible exceptions like Java would do. It's the Flex/ActionScript developers' responsibility to use either error event or *try-catch-finally* code block to catch and handle the errors.

The best practice we suggest is to:

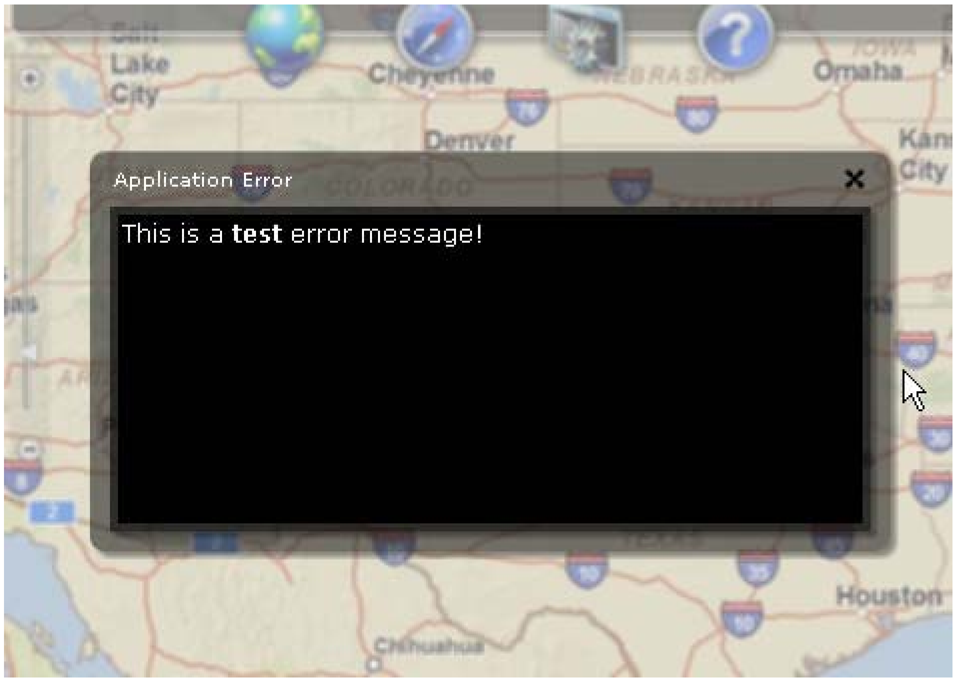
- Implement all possible error events
- Try use *try-catch-finally* code block as much as possible

Once the error is captured, it's also the best practice to gracefully notify the users with plain language and explain the error situation. The Sample Flex Viewer provides a simple error display designed to allow send and display error message easily.

From a widget, a public function defined in the *BaseWidget*, *showError()*, can be used to display a error message. For example:

```
showError("This is a <b>test</b> error message!");
```

This code will result the Sample Flex Viewer application displays:



This is a **test** error message!

Noticed are that the error message window is modal and the text can be styled using basic HTML tags.

Behind the scene, the error message is sent via the Event Bus discussed in earlier section. In other word, the error message can be sent from any where inside the Flex Viewer or a widget as such:

```
SiteContainer.dispatchEvent(new AppEvent(AppEvent.APP_ERROR,
                            false,
                            false,
                            "the error message string"));
```

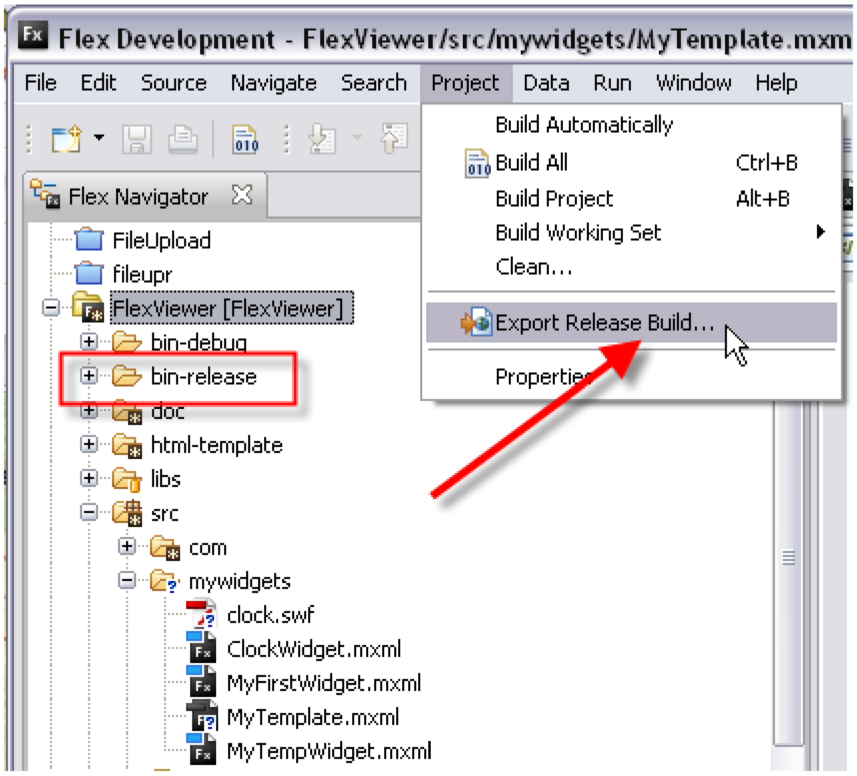
6 Sample Flex Viewer and Widget Deployment

6.1 Deploying the Sample Flex Viewer Application

If you have obtained the binary release of the Sample Flex Viewer, the deployment is simply to unzip the release package into a web server accessible file system location. It might need to configure the web server to have simple URL for this file system location, such as creating an IIS virtual directory, as detailed in the section 2.3 of this document. This aspect of deployment has no different than any other HTML web application deployment.

Deploy Your Own Compiled Sample Flex Viewer Application

If you want to deploy the Sample Flex Viewer application using your own compiled binary, the extra step recommended is to export a release build from the Flex Builder project. A binary (SWF files) from the released build will have reduced size due to the removal of debugging information. As the result, the application will perform better than the debugging build.



From Flex Builder, as shown above, you can choose from the Project menu and do Export Release Build. Be default, a new folder *bin-release* will be created to have all the released files. Everything inside the *bin-release* will be part of your own release package. That means you can rename and zip or move this directory and deploy it as a regular web application.

6.2 Deploying a Widget to the Sample Flex Viewer

Assuming a Sample Flex Viewer application is deployed and a new widget is developed, here are the steps for the widget deployment:

Step 1: Copy the widget SWF file to the intent location

A widget is a stand alone SWF file. You can either copy the widget SWF to the directory that the Sample Flex Viewer application's web server can access, or to another web server.

Step 2: Modify the Sample Flex Viewer application's configuration file

The configuration file, *config.xml*, is at the root directory of the Sample Flex Viewer application. Inside the file, under the tag *widgets*, you need add a new *widget* entry.

The Sample Flex Viewer application supports relative path if the widget is deployed into the same web server where the Sample Flex Viewer is running. The new *widget* tag will be as this:

```
<widget label="A New Widget" icon="urlpath/myicon.png" menu="menuWidgets"
config=”youconfig.xml”>relative/urlpath/MyNewWidget.swf</widget>
```

If you deploy the widget to a separate web server, the fully qualified URL should be specified to the ***widget*** tag. It applies to the icon image files as well:

```
<widget label="A New Widget" icon="http://another-host/urlpath/myicon.png"
menu="menuWidgets" config=”yourconfig.xml”>http://another-
host/urlpath/MyNewWidget.swf</widget>
```

You can specify a configuration file to your widget. In your widget code, the global property *configXML* (in XML data type) defined in the *BaseWidget* will be available to you code. Current, out-of-box the Sample Flex Viewer support only XML configuration file and will automatically construct the *configXML* property. You can use any type of configuration format if you want to parse the configuration data in your won widget code.

Step 3: Refresh the Sample Flex Viewer application

Next time the Sample Flex Viewer application is refreshed in the browser, the new widget configuration will be recognized and a new widget entry will be available at the *Tool* menu (or menu you specified in the config.xml file) from the Sample Flex Viewer user interface.

6.3 Security Considerations

6.3.1 crossdomain.xml

A de facto standard of web browsers is to impose a security sandbox to the browser plug-ins. That means a browser plug-in can only access the remote resources provided by the same web server that servers the current page that runs the plug-in. Flash player, which runs the Flash based application, is a browser plug-in and enforces the same restriction to Flash based applications.

However, a Flash application can access the remote resource from separate web servers only if the severs have a *crossdomain.xml* file deployed, by default, to the root URL of the servers and can be accessed by the Flash application.

The simplest *crossdomain.xml* file is as such:

```
<cross-domain-policy
xsi:noNamespaceSchemaLocation="http://www.adobe.com/xml/schemas/PolicyFile.xsd">
     <allow-access-from domain="*"/>
</cross-domain-policy>
```

The above *crossdomain.xml* file gives access to the Flash application served by any server on internet.

Please refer to Adobe Flex documentation for more details on setting up a *crossdomain.xml* file.

NOTE: Whenever a security exception occurs when running the Sample Flex Viewer application, even internally at your organization, you need check if the resource (such as a ArcGIS Map Service) server has the *crossdomain.xml* file deployed.

6.3.2 Internet Resource Proxy

In many cases, you do need access the resources provided by other servers such as varies of GeoRSS feeds online. If those servers of the GeoRSS feeds have no *crossdomain.xml* file deployed, the only option is to setup a proxy on your own Flash deployed server and point your code to the proxy.

Here is a resource page that provides more information on proxy:
http://resources.esri.com/help/9.3/arcgisserver/apis/javascript/arcgis/help/jshelp_start.htm#jshelp/ags_proxy.htm

7 Appendix A: Configuration XML

Please refer to *README.txt* file comes with the source code distribution for the explanation of the usage of the *config.xml* file.

For developers, please refer to the configuration XML schema document in *doc/ConfigXMLSchema* directory inside the source distribution for the definition of the XML tags in *config.xml* file.