

Creating a Flex-based widget to display WebSphere Business Monitor data in Business Space

Skill Level: Intermediate

[Scott Johnson \(scottjoh@us.ibm.com\)](mailto:scottjoh@us.ibm.com)
Software Engineer, WebSphere Business Monitor
IBM

[Caroline Daughtrey \(cdaughtr@us.ibm.com\)](mailto:cdaughtr@us.ibm.com)
Advisory Software Engineer
IBM

[Chris Ketchuck \(cmketchu@us.ibm.com\)](mailto:cmketchu@us.ibm.com)
Software Engineer
IBM

[Yingxin \(Nicole\) Xing \(nxing@us.ibm.com\)](mailto:nxing@us.ibm.com)
Software Engineer
IBM

29 Apr 2009

Updated 03 Jun 2009

Learn how to deploy an Adobe® Flex® widget into an IBM® WebSphere® Business Monitor business space using a sample. This article shows you how to register an iWidget so that you can drag it from the Business Space widget menu onto your own page, and how to access Monitor data from a Flex application.

Introduction

Business Space for WebSphere Business Monitor V6.2 (hereafter called Monitor) uses the iWidget component model (see [Resources](#) to read the specification).

iWidgets enable Business Space to support many types of browser components, including pure HTML, JavaScript, JavaScript using the Dojo framework, and Adobe Flash®-based components written using the Adobe Flex framework. The iWidget model enables you to develop and deploy your own widgets for Business Space, using the browser technologies of your choice.

In this article, you'll learn how to deploy a Flex widget into a business space. An enterprise application is provided for [download](#). This application contains a sample Flex-based iWidget that accesses data from the Monitor database. You'll learn how to register an iWidget so that you can drag it from the Business Space widget menu onto your own page, and how to access Monitor data from a Flex application. The knowledge you gain from this article will help you develop Flex widgets that provide functions unique to your business environment.

Assumptions

For the purposes of this article, we assume the following:

1. You have administrative access to an installation of Monitor V6.2.
2. You know how to install WebSphere applications. If not, refer to [Resources](#) for ways to install applications or modules.
3. You are familiar with managing spaces and pages in Business Space.
4. You are familiar with Adobe Flex, JavaScript, Dojo and XML.

Download and unzip the sample files

The following [sample files](#) are provided with this article:

- FlexKPIHistorySample.ear is an Enterprise Application Archive (EAR) file that you can install into Monitor V6.2. The Flex widget is contained in this EAR file.
- flexSampleFiles.zip contains the files that are discussed in this article.

Download both files and unzip flexSampleFiles.zip to a directory on your computer.

Deploy and run the Flex iWidget sample

The sample provided with this article displays Key Performance Indicator (KPI) historical data in a Flex widget deployed in Business Space. The KPI data is retrieved from the Better Lender Showcase, which is installed automatically with Monitor V6.2 by selecting **Showcase model** from the **First steps** menu.

Deploy the sample

To deploy the sample, do the following:

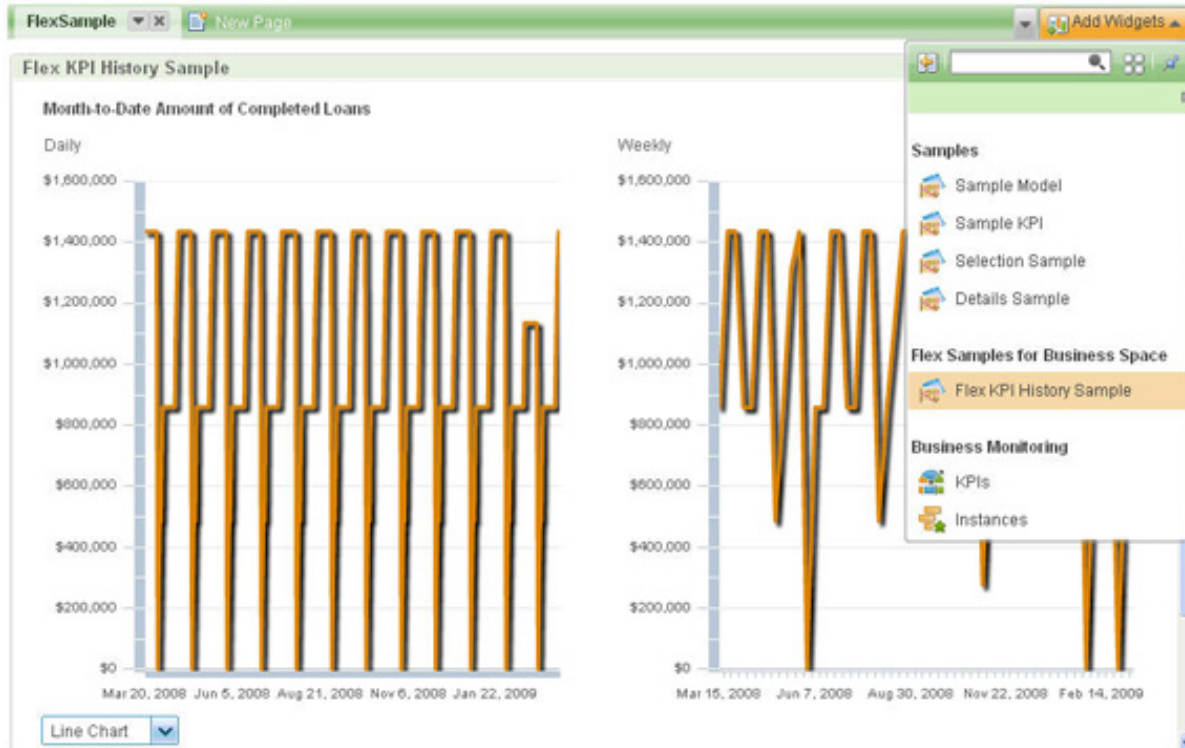
1. Install FlexKPIHistorySample.ear into Monitor V6.2 and start the application.
2. Find the directory registryData in the location where you unzipped flexSampleFiles.zip. From that directory, copy flexKPIHistorySampleWidget.xml and flexSampleEndpoints.xml into the following directory in your Monitor V6.2 installation:
<installdir>\profiles\<profilename>\BusinessSpace\registryData
For example, on a Windows® machine:
Q:\IBM\WebSphere\MonServer\profiles\WBMon01\BusinessSpace\registryData
Note that this directory is below the profiles directory; it is **not** the higher-level directory (on Windows):
Q:\IBM\WebSphere\MonServer\BusinessSpace\registryData

Run the sample

To run the sample, do the following:

1. Log into Business Space and navigate to a page where you want to put the sample widget.
2. Open the **Add Widgets** menu and look for the category called **Flex Samples for Business Space**. You'll see the widget **Flex KPI History Sample** in this category.
3. Drag this widget onto your page. This widget shows historical data for the KPI called **Month-to-Date Amount of Completed Loans**, for both daily and weekly intervals. A drop-down list at the bottom-left of the widget allows you to toggle between Line, Area, and Plot chart representations of the data.

Figure 1. Adding a Flex widget to Business Space



iWidget definition and implementation files

Every widget that is deployed in Business Space has iWidget definition and implementation files. The sample files are:

- flexKPIHistorySample_iWidget.xml, the definition file
- flexKPIHistorySample.js, the implementation file

These files are found in subdirectory iWidgetFiles, in the directory where you unzipped flexSampleFiles.zip.

Open these files in a text editor and take a look.

The iWidget definition file

The definition file, flexKPIHistorySample_iWidget.xml, describes properties of the widget. The following properties are significant for the sample widget.

The <iw:iwidget> tag:

- id="flexKPIHistorySample"
The id is referenced in the deployment file

flexKPIHistorySampleWidget.xml.

- `id="flexKPIHistorySample"`
The id is referenced in the deployment file `flexKPIHistorySampleWidget.xml`.
- `iScope="com.ibm.bspace.widgets.samples.FlexKPIHistorySampleWidget"`
The `iScope` value is the fully qualified name of the sample widget. This value is also found in the `iWidget` implementation file `flexKPIHistorySample.js`.
- `supportedModes="view"`
This widget supports view mode, but not edit mode.

The `<iw:resource>` tags:

- `uri="<relativepath>/BSpaceCommonUtilityLoader.js"`
This file provides Business Space utilities.
- `uri="flexKPIHistorySample.js"`
This is the `iWidget` implementation file, so it is, of course, required!

The `<iw:content>` tag

- `mode="view"`
The content is used in view mode (which is this widget's only mode).
- `span id="_IWID_KPIHistorySampleSampleWidget"`
This is where the Flash content will be placed.

The `iWidget` implementation file

The implementation file, `flexKPIHistorySample.js`, displays the widget and handles some events like refresh and unload. The following sections of the implementation file are significant for the sample widget.

`dojo.provide` and `dojo.declare`:

- `"com.ibm.bspace.widgets.samples.FlexKPIHistorySampleWidget"`
This is the fully qualified name of the sample widget. This value is also

seen in the iWidget definition file flexKPIHistorySample_iWidget.xml.

```
onview: function(){}  
  
    • var endpoint = this.getServiceURLRoot();  
      The widget will need to access Monitor data. The endpoint will contain  
      the necessary URL for making requests to the Monitor database.  
    • var url =  
      this.iContext.io.widgetBaseUri+"FlexKPIHistorySample.swf";  
      This locates the Flash content; url is used when placing the Flash  
      content on the page.  
    • selectionTextNode.innerHTML = "<object id='"+id etc.  
      This places the Flash content on the page.
```


Business Space registry files

Every widget that is deployed in Business Space must be registered with Business Space. Our sample has two files that register the widget. The sample files are:

- flexKPIHistorySampleWidget.xml, the widget's registry file
- flexSampleEndpoints.xml, the widget's endpoints file

These files are found in subdirectory registryData, in the directory where you unzipped flexSampleFiles.zip.

Open them now and take a look.

The widget registry file

The registry file, flexKPIHistorySampleWidget.xml, tells Business Space where the widget is located, what category it belongs to in the widget menu, and the widget's display name, among other details. The following sections of the registry file are significant for the sample widget.

The <tns:Category> tag:

- <tns:id>{com.ibm.bspace}bpaceflexsamplewidgets</tns:id>
This is the ID for our new widget category.

- `<tns:name>` , `<tns:description>`, etc.
Self-explanatory attributes.

The `<tns:Widget>` tag:

- `<tns:id>{com.ibm.bspace}flexKPIHistorySample</tns:id>`
The ID `flexKPIHistorySample` is also found in the `iWidget` definition file; the ID associates the registry entry with the `iWidget` definition file.
- `<tns:type>{com.ibm.bspace}iWidget</tns:type>`
Identifies the widget as a Business Space `iWidget`.
- `<tns:categoryId>{com.ibm.bspace}bpaceflexsamplewidgets</tns:cate`
Identifies the category to which this widget belongs.
- `<tns:widgetEndpointId>{com.ibm.bspace}flexKPIHistorySampleId</tns`
Provides the context root for the sample application which hosts our widget. The context root is actually defined in the widget endpoint file which is described below.
- `<tns:url>widgets/samples/flexKPIHistorySample/flexKPIHistorySampl`
Identifies the location of the `iWidget` definition file within our sample application. The `iWidget` definition file, as described earlier, identifies the JavaScript implementation file that actually places the widget on the page.
- `<tns:serviceEndpointRef...`
`<tns:name>serviceUrlRoot</tns:name>`
`<tns:refId>{com.ibm.wbimonitor}monitorServiceRootId</tns:refId>`
Identifies the service endpoint that provides the URL for Monitor data access. Note that `{com.ibm.wbimonitor}monitorServiceRootId` is declared in another file in the same directory as our widget registry files. That file is `monitorEndpoints.xml`.

The widget endpoint file

The endpoint file, `flexSampleEndpoints.xml`, tells Business Space the context root for the application that hosts the widget.

The `<tns: Endpoint >` tag:

- `<tns:url>flexKPIHistorySample/</tns:url>`

This is the context root of our sample application.

Create a user interface with Flex

The first step in creating a Flex application is designing what will be on the page and where. For our sample, we want to see:

- KPI name in the upper right corner
- Two flex charts side by side, with the labels for their history granularity level positioned above the charts
- A drop-down menu that controls the chart data series type

For a quick start, you can take advantage of the Flex Builder's design view. Open Flex Builder and create a new Flex project. After creating a new Flex application, switch to the design view. This view allows you to quickly assemble the skeleton of the UI by dragging and dropping components onto the design panel, as shown in Figure 2.

Figure 2. Flex Builder Design view

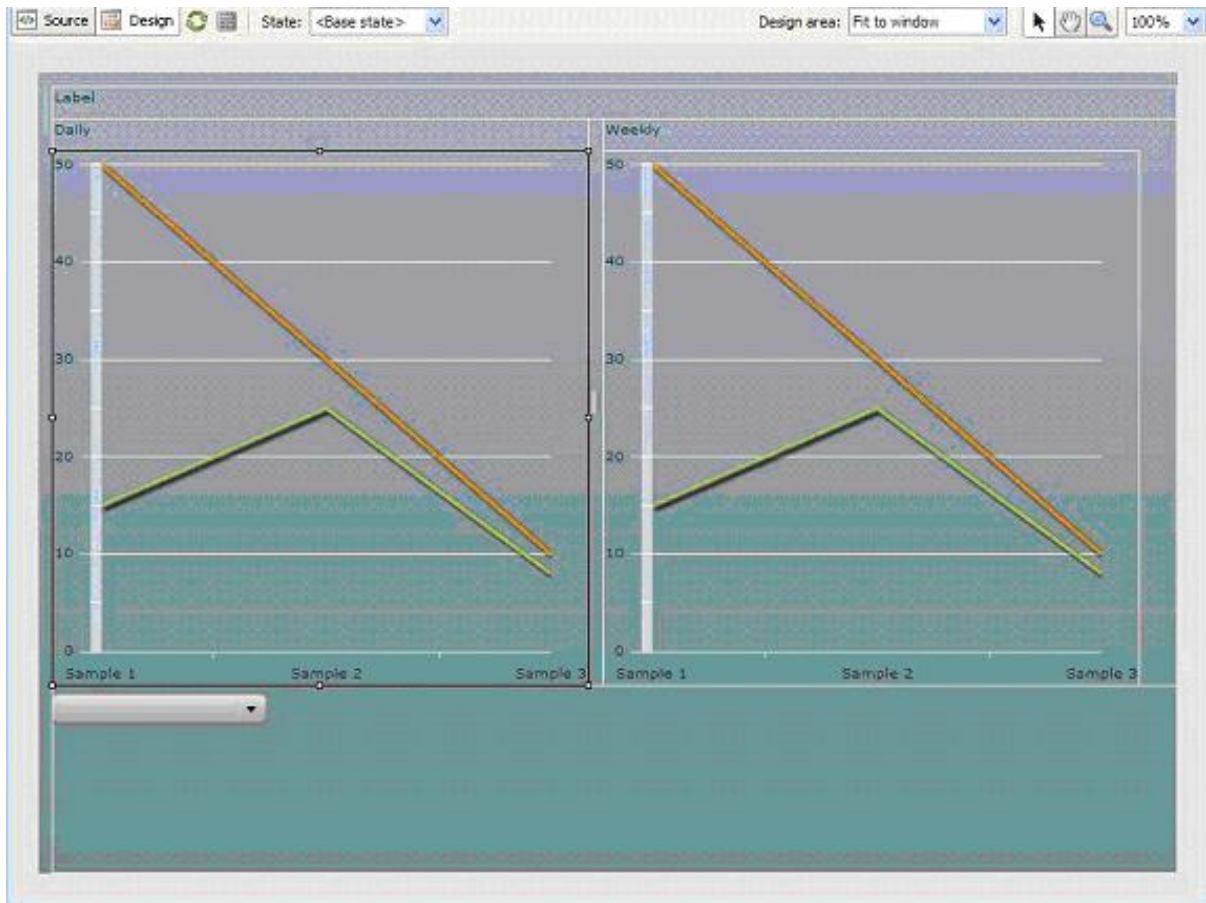
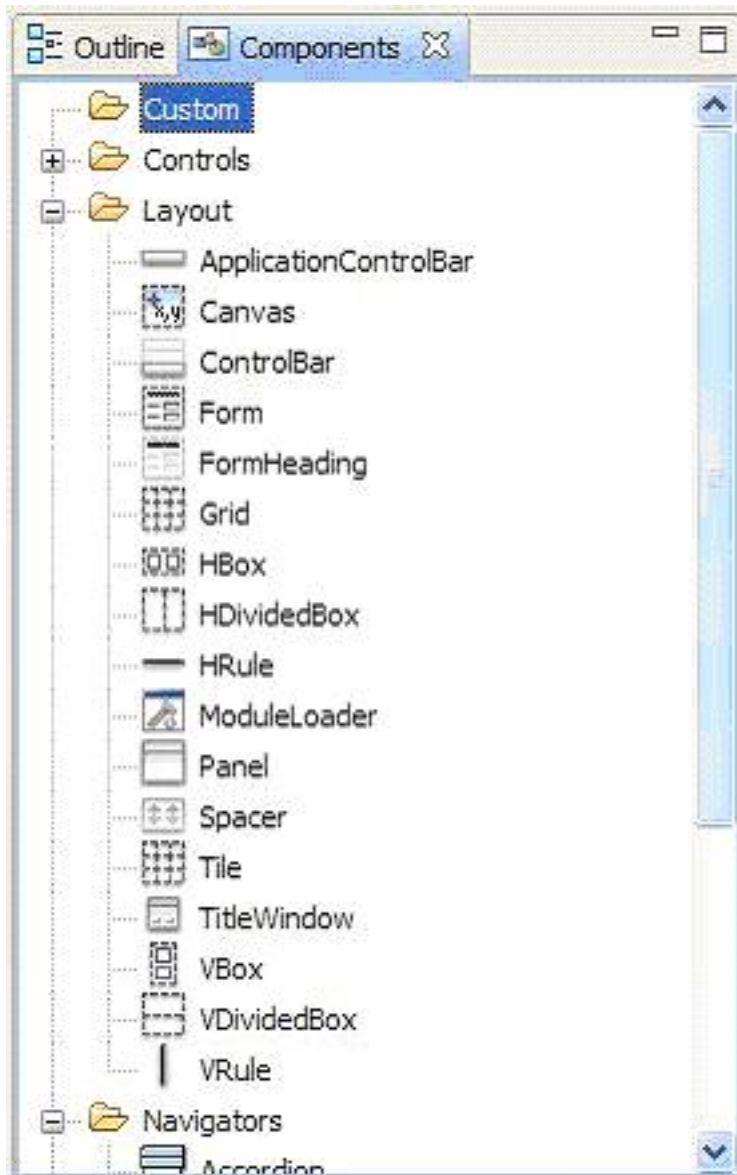


Figure 3 shows the component list next to the design view panel.

Figure 3. Component list



Switching back to the Source view, you'll see the MXML code that has been generated by the builder, as shown in Listing 1.

Listing 1. Generated MXML code

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
  <mx:VBox x="10" y="10" width="100%" height="100%">
    <mx:Label text="Label"/>
    <mx:HDividedBox width="100%">
      <mx:VBox height="100%">
        <mx:Label text="Daily"/>
        <mx:LineChart id="linechart1">
          <mx:series>
```

```
        <mx:LineSeries displayName="Series 1" yField="" />
    </mx:series>
</mx:LineChart>
</mx:VBox>
<mx:VBox height="100%">
  <mx:Label text="Weekly" />
  <mx:LineChart id="linechart2">
    <mx:series>
      <mx:LineSeries displayName="Series 1" yField="" />
    </mx:series>
  </mx:LineChart>
</mx:VBox>
</mx:HDividedBox>
<mx:ComboBox></mx:ComboBox>
</mx:VBox>
</mx:Application>
```

Take a look at the code that has been generated by the builder. The root level is the MXML tag for `Application`, which is the starting point for any Flex application. Within `Application` you define the base layout and other visualization components. You can add components dynamically to the page using ActionScript by appending them as children of existing components.

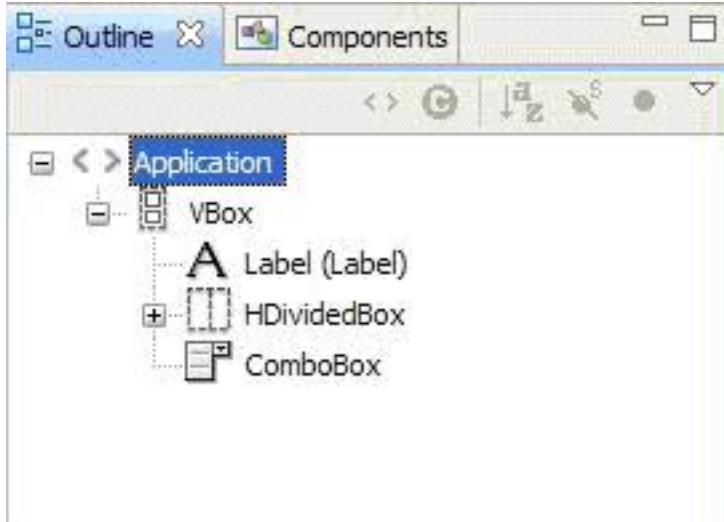
In our sample the level above `Application` is `VBox`. As the name suggests this is a layout container that automatically aligns all its child components vertically. Flex provides the following layout container components to facilitate the structuring of your Flex application:

- `Application ControlBar`
- `Box (HBox and VBox)`
- `Canvas`
- `ControlBar`
- `DividedBox (HDividedBox and VDividedBox)`
- `Form`
- `Grid`
- `Panel`
- `Tile`
- `TitleWindow`

For more detail on these components and their use, refer to [Resources](#).

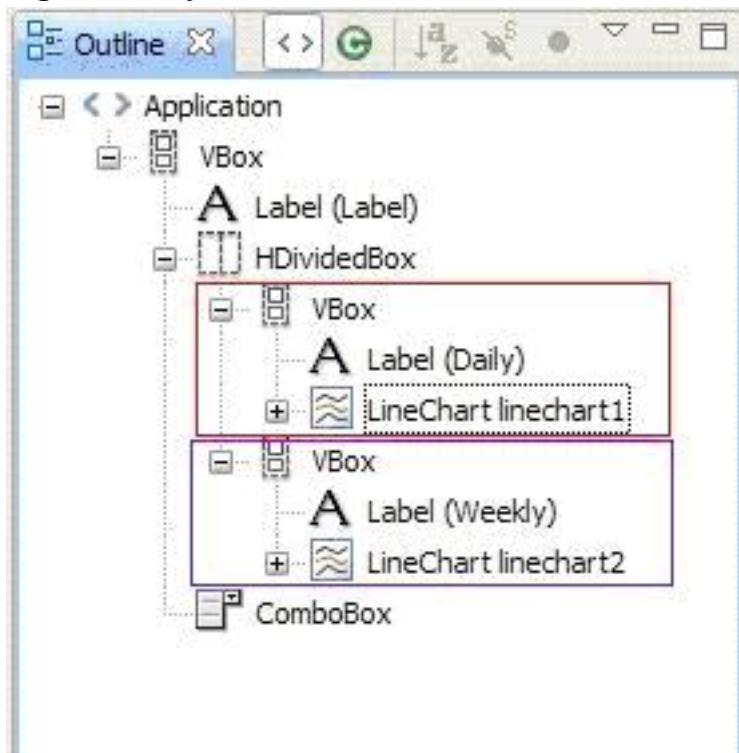
In our `VBox`, we'd like to see three child components aligned vertically, as shown in Figure 4:

Figure 4. Layout in `VBox`



The Label component is used to display the KPI name; HDividedBox is the place holder for charts and their labels; ComboBox is used as the controller to switch between different types of chart data series. We've chosen three types of series to demonstrate here: LineSeries, AreaSeries, and PlotSeries. We'll talk more about the charting component later in this article.

Figure 5. Layout in HDividedBox



Within the HDividedBox, we've nested another level of VBox. These two charts within VBoxes will be automatically aligned horizontally side by side within the HDividedBox. The purpose of the inner level VBox is to group the granularity label

with its corresponding chart.

The last visualization component on this page is the `ComboBox`. This `ComboBox` is intended to control the data series type of the two charts, but it cannot function without populating it with the right values and labels.

Let's go ahead and add the `Script` section to our `Application` and define a `CHART_TYPES` variable that contains the values for `ComboBox`, as shown in Listing 2.

The completed MXML source for this sample is found in `FlexKPIHistorySample.mxml` in the `FlexSource` directory where you unzipped `flexSampleFiles.zip`.

Listing 2. Define chart types (code snippet from `FlexKPIHistorySample.mxml`)

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
  <mx:Script>
    <![CDATA[
      [Bindable] private var CHART_TYPES:Array =
        [ {label:"Line Chart",data:"Line
Chart"},
          {label:"Area Chart",data:"Area
Chart"},
          {label:"Plot Chart",data:"Plot
Chart"} ];
    ]]>
  </mx:Script>
```

To populate the `ComboBox` with these `CHART_TYPES`, all you need to do now is specify the `CHART_TYPES` variable as its `dataProvider`, as shown here:

```
<mx:ComboBox dataProvider="{CHART_TYPES}"></mx:ComboBox>
```

To finish defining the `ComboBox`, add the following attributes:

```
<mx:ComboBox id="chartTypeDropDown" dataProvider="{CHART_TYPES}"
  editable="false" enabled="true"
  change="updateChart();"></mx:ComboBox>
```

The `change` attribute tells the application what to do when the selected value of `chartTypeDropDown` changes. In our sample, the `updateChart` function will be invoked when the selection changes. You need to define `updateChart` under the `Script` for `chartTypeDropDown` to work, as shown in Listing 3.

Listing 3. Define `updateChart` (code snippet from `FlexKPIHistorySample.mxml`)

```
<mx:Script>
  <![CDATA[
```

```
[Bindable] private var CHART_TYPES:Array =
    [{label:"Line Chart",data:"Line Chart"},
     {label:"Area Chart",data:"Area Chart"},
     {label:"Plot Chart",data:"Plot
Chart"}];
[Bindable] private var chartTypeSelection:String = "Line Chart";

private function updateChart():void{
    chartTypeSelection = new String(chartTypeDropDown.value);

    if(chartTypeSelection=="Line Chart")
        addLineSeries();
    else if(chartTypeSelection=="Area Chart")
        addAreaSeries();
    else
        addPlotSeries();
}
]]>
</mx:Script>
```

First, `updateChart` saves the `chartTypeDropDown` selection value in the global variable `chartTypeSelection`. Later, the corresponding function to add data series for the two charts is invoked based on the selection value of `chartTypeDropDown`.

See [Resources](#) to learn more about `ComboBox` use.

Create the KPI History data service and build the URL

The KPI history stored on the server is accessible using a Monitor Representational State Transfer (REST) service. In our sample, we'll use the **Month-to-Date Amount of Completed Loans** KPI.

The KPI History service requires the model ID, model version, and KPI ID to build the URL. In this sample, we'll use the Better Lender Showcase Model. The model ID is `MortgageLendingBAMShowcase`, the model version is `20080918060000`, and the KPI ID is `Monthly_Total_of_Completed_Loan_Dollars`.

The History service supports a wide range of time ranges when retrieving the history values. History can be retrieved using a sliding, fixed, or current period, or last completed period filter. For this sample, we'll retrieve the history using a sliding interval of one year. On the server the sliding interval is referred to using as `rollingPeriod`. The following parameters are used to construct the filter:

- `timerangemethod = rollingPeriod`
- `rollingperiodduration = years`
- `rollingperiodquantity = 1`

Additional parameters can be passed on the URL using query parameters. For

example, the History service supports localization by passing in a locale and a time zone. We'll hard-code these values to `en` and `America/New_York` respectively. You can find a complete list of supported parameters in the WebSphere Business Monitor Information Center (see [Resources](#)).

Access the REST service with Flex

Once the URL has been constructed, Flex provides a variety of ways to make the request. For our sample, we'll use the `URLLoader` class. The `URLLoader` works off of a number of events. We'll use the `Event.COMPLETE` and `IOErrorEvent.IO_ERROR` events to determine the success or failure of the request. Listing 4 shows how to build the REST URL and create the request using the `URLLoader`. You can make a call to `getKpiHistoryByIdRollingTimePeriod` to request the daily granularity as shown in Listing 5.

The completed Action Script source for the sample's `KpiHistoryService` is found in `KpiHistoryService.as` in the `FlexSource` directory where you unzipped `flexSampleFiles.zip`.

Listing 4. Build the REST URL (code snippet from `KpiHistoryService.as`)

```
private const HISTORY_URL: String =
    "/bpm/monitor/models/#/versions/##/kpis/history/###";

public function getKpiHistoryByIdRollingTimePeriod(kpiId:String,
    granularity:String,
    rollingPeriodDuration:String, rollingPeriodQuantity:int):void{
    getKpiHistoryById(kpiId, KpiHistoryDataModel.rollingTimeRangeMethod,
        granularity, rollingPeriodDuration, rollingPeriodQuantity);
}

public function getKpiHistoryById(kpiId:String,
    timerangemethod:String = null,
    granularity:String = null, rollingPeriodDuration:String = null,
    rollingPeriodQuantity:int = -1):void{

    // Setup events
    loader.addEventListener(Event.COMPLETE, historyLoadComplete);
    loader.addEventListener(IOErrorEvent.IO_ERROR,
        historyLoadFailure);

    // Build the url
    var modelURL:String = HISTORY_URL.replace('#', _model);
    var modelVersionURL:String = modelURL.replace('##', _version);
    var modelVersionKpiURL:String = modelVersionURL.replace('###',
        kpiId);

    // Add query params
    var queryString:String = "?locale=" + _locale + "&timezone=" +
        _timezone;

    // Setup the rolling params for rest, will vary with different
    time range methods
    if(timerangemethod != null && timerangemethod ==
        KpiHistoryDataModel.rollingTimeRangeMethod){
```

```

        queryString = queryString + "&timerangemethod=" +
timerangemethod +
        "&granularity=" + granularity + "&rollingperiodduration=" +
        rollingPeriodDuration + "&rollingperiodquantity=" +
rollingPeriodQuantity;
    }

    var request:URLRequest = new URLRequest(_restHost +
        modelVersionKpiURL + queryString);
    loader.load(request);
}

```

Listing 5. Request daily granularity (code snippet from FlexKPIHistorySample.mxml)

```

_kpiHistoryService.getKpiHistoryByIdRollingTimePeriod(
    "Monthly_Total_of_Completed_Loan_Dollars",
    KpiHistoryDataModel.granularityDaily,
    KpiHistoryDataModel.rollingPeriodDurationYearly, 1);

```

The URL used in the history request will look similar to the following:

```

<hostname>/rest/bpm/monitor/models/MortgageLendingBAMShowcase/versions/20080918060000
/kpis/history/Monthly_Total_of_Completed_Loan_Dollars?locale=en&timezone=America/New_York
=rollingPeriod&granularity=daily&rollingperiodduration=years&rollingperiodquantity=1

```

Define error handling

In the event of a network error, the error handler `historyLoadFailure` is called so the error can be handled appropriately. A network error could result from an unauthorized URL due to model access or an incorrect URL. In the event of a successful REST request, `historyLoadComplete` is called. One more check has to be performed in `historyLoadComplete`, because the request may have returned but might contain an error from the REST service. Errors can happen for a variety of reasons. The most common reason in a development environment is that a parameter is missing or unrecognized. In a production environment, it's more common that a KPI has been deleted and the history does not exist. If no errors are detected, we set the returned data to the data model using `setKpiHistory` and dispatch the complete event. The caller should set up a listener to handle a successful data fetch before invoking `getKpiHistoryByIdRollingTimePeriod` by setting up a listener for the complete event on the `kpiHistoryService`. The KPI history requests are asynchronous; an event can be dispatched after you have verified the data and finished the error handling. Listing 6 shows how to specify the error check of the returned data and dispatch the event.

Listing 6. Check for errors and dispatch event (code snippet from KpiHistoryService.as)

```

private function historyLoadComplete(e:Event): void {

```

```

    // Reached this point with no network errors, could still be REST
errors
removeEventListeners()
var jsonObject:Object = decodeJSONObject(e.target.data);
var error:Object = getError(jsonObject);
if(error != null){
    // Error received from the REST server
    trace("Rest error detected: " + error["formattedMessage"]);
} else {
    // No errors from the server and no networks errors
    trace("No rest errors detected");
KpiHistoryDataModel.getInstance().setKpiHistory(jsonObject["KPI
ID"], jsonObject);
    dispatchEvent(new Event(Event.COMPLETE));
}
}
}

```

Store the data in the KPI data model

After a successful REST request, you need to store the data for easy access. The data comes back from REST as serialized JavaScript Object Notification (JSON). This format can be turned into a JSON object using the utilities in the `ascorelib` library. For this sample, we'll use an object to store the latest request for each KPI. The assumption is that a configuration exists for each model and version. In this case, each KPI has a unique identifier, which can be used as a key to access the history data.

The KPI data model contains useful constants that are used in accessing the KPI service. For example, a valid `timerangemethod` can be found in the data model. The data model also has a setter and getter to access the data. Listing 7 shows some of the constants and a setter method.

The completed Action Script source for the sample's `KpiHistoryDataModel` is found in `KpiHistoryDataModel.as` in the `FlexSource` directory where you unzipped `flexSampleFiles.zip`.

Listing 7. Data model constants and setter method (code snippet from `KpiHistoryDataModel.as`)

```

private var _history:Object = new Object();

public static const
    rollingTimeRangeMethod:String="rollingPeriod";

public function setKpiHistory(kpiId:String,
    historyDetails:Object):void{
    history[kpiId] = historyDetails;
}

```

Load the chart data

To load the chart data, use the `_kpiHistoryService` and `KpiHistoryDataModel` as shown in Listing 8. The service is created in the `init()` function. First, add an event listener to invoke the `loadChart()` function when the next service event is complete. Then make the first history service request for daily granularity with a duration of yearly. When this request is complete, the `loadChart()` function is called and the current event listener is removed. Next we'll get the history data from the model, setting the variables defined in the mxml for the Label text `kpiName` and the LineChart dataProvider `kpiValue`. Then we'll start the entire process over again to create the second chart with weekly granularity.

Listing 8. Load chart data (code snippet from FlexKPIHistorySample.mxml)

```
public function init():void {
    _kpiHistoryService = new KpiHistoryService(host, modelId,
    version, locale, timezone)

    // Make request for daily granularity
    _kpiHistoryService.addEventListener(Event.COMPLETE, loadChart);
    _kpiHistoryService.getKpiHistoryByIdRollingTimePeriod(
        "Monthly_Total_of_Completed_Loan_Dollars",
        KpiHistoryDataModel.granularityDaily,
        KpiHistoryDataModel.rollingPeriodDurationYearly, 1);
}

private function loadChart(e:Event):void{
    _kpiHistoryService.removeEventListener(Event.COMPLETE,
    loadChart);

    historyData =
    KpiHistoryDataModel.getInstance().getKpiHistoryById(
        "Monthly_Total_of_Completed_Loan_Dollars");
    kpiName = historyData["KPI Display Name"];
    kpiValue = historyData["KPI Value Array"];

    // Make request for weekly granularity
    _kpiHistoryService.addEventListener(Event.COMPLETE, loadChart2);
    _kpiHistoryService.getKpiHistoryByIdRollingTimePeriod(
        "Monthly_Total_of_Completed_Loan_Dollars",
        KpiHistoryDataModel.granularityWeekly,
        KpiHistoryDataModel.rollingPeriodDurationYearly, 1);
}

private function loadChart2(e:Event):void{
    _kpiHistoryService.removeEventListener(Event.COMPLETE,
    loadChart2);
    historyData2 =
    KpiHistoryDataModel.getInstance().getKpiHistoryById(
        "Monthly_Total_of_Completed_Loan_Dollars");
    kpiValue2 = historyData2["KPI Value Array"];
}
```

Add the series type to the chart

When the `updateChart()` function is called, one of the three add series functions is called as shown in Listing 9. As the sample demonstrates, Flex allows you to add different types of series to a chart. The sample allows the user to change the series

type between `LineSeries`, `AreaSeries` and `PlotSeries`. You create a series as shown in the `addLinesSeries()` function in Listing 9, by creating a new `Series` object, setting the `ID`, `xField`, `yField` and `displayName` properties on the object and then adding the `Series` to the chart's current series. The `ID` corresponds to the `ID` in the `mxml`'s `mx:LineChart`. The `xField` and `yField` correspond to a key in the `kpiValue` `dataProvider` array.

Listing 9. AddSeries function (code snippet from FlexKPIHistorySample.mxml)

```
private function updateChart():void{
    chartTypeSelection = new String(chartTypeDropDown.value);

    if(chartTypeSelection=="Line Chart")
        addLineSeries();
    else if(chartTypeSelection=="Area Chart")
        addAreaSeries();
    else
        addPlotSeries();
}

private function addLineSeries():void {
    var currentSeries_d:Array = dailyGranularityChart.series;
    if(currentSeries_d.length>0)
        currentSeries_d.pop();
    var series_d:LineSeries = new LineSeries;
    series_d.id = "dailyValueLine";
    series_d.xField = "KPI Period Timestamp";
    series_d.yField = "KPI Value";
    series_d.displayName = "Daily Kpi History Value";
    currentSeries_d.push(series_d);

    dailyGranularityChart.series = currentSeries_d;

    var currentSeries_w:Array = weeklyGranularityChart.series;
    if(currentSeries_w.length>0)
        currentSeries_w.pop();
    var series_w:LineSeries = new LineSeries;
    series_w.id = "weeklyValueLine";
    series_w.xField = "KPI Period Timestamp";
    series_w.yField = "KPI Value";
    series_w.displayName = "Weekly Kpi History Value";
    currentSeries_w.push(series_w);

    weeklyGranularityChart.series = currentSeries_w;
}
```

Load the theme at runtime

You can easily apply the unique look and feel of the business space to your application. The `BSpace_Skin` theme overrides most of the default Flex styles. Flex has many ways of applying styles. For this sample, we've packaged the theme into an SWF file by compiling our external style sheet, `BSpace_Skin.css`. To apply this theme to the sample, use the `StyleManager` class to load `BSpace_Skin.swf` at runtime as shown in Listing 10. The `loadStyleDeclarations()` method requires the path to the SWF file in the first parameter. For this sample, we've hardcoded the context root of the EAR file. Alternately, you could pass this into Flex using the

flashVars parameter in the onView() function of flexKPIHistorySample.js. Then in the FlexKPIHistorySample.mxml init() function, you could read the variable in by using Application.application.parameters. Because the styles should be updated immediately, you'll set the second parameter of the loadStyleDeclarations to true.

Listing 10. Apply the BSpace_skin theme (code snippet from FlexKPIHistorySample.mxml)

```
<mx:Script><![CDATA[
    private static const contextRoot:String="flexKPIHistorySample";

    public function init():void {
        StyleManager.loadStyleDeclarations("/"+contextRoot+
            "/widgets/samples/flexKPIHistorySample/BSpace_Skin.swf",
            true);
    }
}]></mx:Script>
```

Tips on developing Flex widgets

This section contains some tips and practical advice on developing Flex widgets.

Calling a function in a Dojo widget from Flex

You may find you need to call a function in your Dojo widget from your Flex code. This is easy! In your Flex code, use the ExternalInterface.call function as shown in Listing 11. Copy and paste the function window.invokeWidget (Listing 12) into a JavaScript file that you are certain will be loaded on the page where your Flex application is loaded.

In your Flex code use the ExternalInterface.call function:

Listing 11. The ExternalInterface.call function

```
// params is optional
ExternalInterface.call("invokeWidget", this.widgetId,
    "yourWidgetFunctionName", params);
```

- "invokeWidget" is the name of a function provided for you in Listing 12.
- this.widgetId is your Dojo widget's unique ID. This ID can be passed into your Flex application by using flashVars properties when the Flex application is created.
- "yourWidgetFunctionName" is the name of the function in your Dojo widget that you want to call.

- **params.** If you pass parameters (not required), they need to be in an Object, such as:

```
var params:Object=new Object();
params["param1"]=param1Value;
params["param2"]=param2Value;
```

The function in Listing 12 just needs to be available on the page where your Flex application is loaded. It is just a function in a .js file. Be sure to define it at the window level with "window." or else it won't work on Internet Explorer. If you rename the function from 'invokeWidget' to something else, be sure to use that new name in your ExternalInterface.call function — the first argument is the name of that function.

Listing 12. The invokeWidget function

```
window.invokeWidget=function(widgetId, fnName, params){
    // summary:
    //   Invoke a function on a widget
    //
    // description:
    //   This is a generic function that is being defined at the
window
functions // level to allow Flex and other plug-ins a way to invoke
// defined on our widgets.
var result = undefined;
var widget = dijit.byId(widgetId);
if (widget) {
    var fn = widget[fnName];
    if (fn && dojo.isFunction(fn)) {
        result = fn.call(widget, params);
    }
}
return result;
}
```

Calling Flex from a Dojo widget function

You may find you need to call your Flex code from your Dojo widget. In order to do this you have to tell Flex to register the function as callable, by using ExternalInterface.addCallback method:

In Flex:

Listing 13. Registering a function as callable in Flex

```
// to keep it simple, both args are the same string
ExternalInterface.addCallback("flexFunctionName", flexFunctionName);
```

In Dojo:

Listing 14. Calling Flex code from a Dojo widget

```
myDojoWidgetFunction: function(){
    // Note that swfId is the id attribute of the OBJECT tag and
    // the name attribute
    // of the EMBED tag that loaded your Flex application.
    var swfId=yourFlexAppId;
    myObj=new Object();
    myObj["var1"]="myVar1";
    myObj["var2"]="myVar2"
    // Call your Flex app here! Note that passing arguments is
    // optional, but if you
    // do pass args the signature of the function in Flex must
    // match what you call.
    if (dojo.isIE) {
        console.log(" is IE");
        window[swfId].flexFunctionName(myObj);
    } else {
        console.log(" is NOT IE");
        document[swfId].flexFunctionName(myObj);
    }
}
```

Important: After you have added your Flash application to the page, be sure not to move the div to which the Flash object tag is attached. If you move the div after the Flash object tag has been set -- for example, by appending it to another node -- then Flex's `ExternalInterface` will probably fail on Internet Explorer.

We don't use `ExternalInterface` in our sample, but we have encountered a problem on Internet Explorer where `ExternalInterface.addCallback()` fails if the Flash application is moved.

For example, look at the sample file `flexKPIHistorySample.js`. We create a div to hold the Flash content as shown here:

```
//Create a div to hold the Flash content
var selectionTextNode = document.createElement('div');
```

Next we append the div to the parent node:

```
// IMPORTANT: append the node to the parent BEFORE setting its
// innerHTML.
// This solves problems on Internet Explorer having to do with using
// ExternalInterface in the Flex app
widgetParentNode.appendChild(selectionTextNode);
```

Finally, we set the `innerHTML` to contain the object tag:

```
selectionTextNode.innerHTML = "<object id='"+id+.....
```

Encoding flashVars properties

According to Adobe, you must encode the values of flashVars properties as shown in the example in Listing 15.

Listing 15. Encoding flashVars properties

```
var flashVars = 'flashVar1=' + encodeURIComponent(yourArg1) +  
'&flashVar2=' + encodeURIComponent(yourArg2) + .....
```

Attaching your widget to a new node

Be sure to attach your widget to a new node; do not attach it directly to a `dojoattachpoint` in a Dojo HTML template or to a `div` element in an `iWidget` XML file. When a Dojo widget is destroyed, its root node is destroyed as well. By appending a new node to your `dojoattachpoint` or your `iWidget`'s `div` element, and using that new node as the root node of the widget, you will ensure that your `dojoattachpoint` or `div` is not destroyed and your widget can be recreated using that same `dojoattachpoint` or `div`. In the example in Listing 16, `viewAttach` is the id of a `div` element in an `iWidget` XML file.

Listing 16. The viewAttach div element

```
<!-- View mode markup -->  
<iw:content mode="view">  
  <![CDATA[  
    <div id="viewAttach" class="yourThemeClassNames"></div>  
  ]]>  
</iw:content>
```

In the code that attaches your widget to the page, you should first create a `div` element, append it to the `div` element in your template or `iWidget` file, and pass the new `div` as the root node of the widget.

Listing 17. Attaching the widget to the page

```
var viewAttach = this.getElementById('viewAttach');  
var div = document.createElement('div');  
div.id = this.widgetId;  
viewAttach.appendChild(div);  
  
this.viewWidget = new my.widget.package.MyViewWidget(params,  
div);
```

Summary

With what you have learned in the article, you will be able to develop, deploy, and register your own custom Flex-based widgets for Business Space. Your business customers will be able to easily drag and drop your new widgets from the Business Space widget menu onto the page they use every day for monitoring business processes.

Downloads

| Description | Name | Size | Download method |
|------------------------------------|--------------------------|-------|----------------------|
| EAR containing Flex widget | FlexKPIHistorySample.ear | 654KB | HTTP |
| Flex source and XML registry files | flexSampleFiles.zip | 77KB | HTTP |

[Information about download methods](#)

Resources

- [iWidget specification](#): For details on the iWidget component model.
- [WebSphere Application Server InfoCenter](#): For instructions on how to install applications.
- [Adobe Help Resource Center](#): For details on using layout containers.
- [Adobe Help Resource Center](#): For details on ComboBox controls.
- [KPI History reference](#): For details on KPI history parameters.
- [Using ActionScript](#): Learn more about ActionScript.
- [Programming ActionScript 3.0](#): For in-depth programming information on ActionScript.
- [Flex Adobe 3.2 Language Reference](#): For details on ActionScript classes.
- [Flex Charting Component](#): For more information about charting.
- [Using Styles and Themes](#): For more information about styles.
- [Communicating with the wrapper](#): For more information about passing variables into your Flex application.
- [WebSphere business process management zone](#): Get the latest technical resources for WebSphere BPM solutions, including articles, tutorials, events, downloads, and more.
- [WebSphere Business Monitor education](#): A course list of WebSphere Business Monitor classes.
- [WebSphere Business Monitor V6.2 Information Center](#): Get complete product documentation.
- [developerWorks articles related to Monitor](#): Check out other developerWorks articles on WebSphere Business Monitor for a wealth of how-to information and samples.

About the authors

Scott Johnson

Scott Johnson has been a software developer for 25 years. Before joining IBM he worked in the areas of banking, cash management, survey research, and healthcare scheduling and staffing. He joined IBM in 2000 as the JavaServer Pages component lead for WebSphere Application Server. He currently uses Javascript, Dojo and Adobe Flex to develop widgets for WebSphere Business Monitor.

Caroline Daughtrey

Caroline Daughtrey is an Advisory Software Engineer with the WebSphere Business Monitor team in Research Triangle Park, NC. In the past 10 years with IBM, she has worked on products such as WebSphere Business Components and WebSphere Business Integration. Prior to joining IBM, Caroline began her career as a medical technologist and then worked as a developer and manager of clinical laboratory information systems.

Chris Ketchuck

Chris Ketchuck has been a software developer on the WebSphere Business Monitor team for two years. While in this position, he has developed Web-based dashboards using a variety of technologies including HTML, JavaScript, and Flex.

Yingxin (Nicole) Xing

Yingxin Xing currently works on the development team for WebSphere Business Monitor dashboards. She has five years of industry experience with WebSphere and Tivoli.