

---

# Developer's Guide

Create GeoWeb Application with the Sample JavaScript Viewer

---

<b>Author</b>	Simon Biickert, Moxie Zhang
<b>Group</b>	Corporate Sales, ESRI, Inc
<b>File</b>	JSViewerDevelopersGuide.pdf
<b>Last Revised</b>	<b>January 13, 2009</b>
<b>Status</b>	Public Release

# Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>4</b>
1.1	Prerequisites .....	4
1.1.1	Skill and Software.....	4
1.1.2	Obtain Source Code and Libraries .....	5
<b>2</b>	<b>Sample JavaScript Viewer Architecture.....</b>	<b>5</b>
2.1	Overview .....	5
2.2	Sample JavaScript Viewer Lifecycle .....	6
2.3	Sample JavaScript Viewer Application.....	7
2.3.1	Understand Widget and Widget Programming Model .....	8
2.4	Widget Naming Convention.....	11
<b>3</b>	<b>JavaScript Viewer Setup for Development .....</b>	<b>12</b>
3.1	Unzip the source distribution .....	12
3.2	Edit config.xml .....	13
3.3	Configure Widgets.....	13
3.4	Configure Proxy Support.....	14
<b>4</b>	<b>Develop a Widget for Sample JavaScript Viewer .....</b>	<b>17</b>
4.1	Create the Dojo Module .....	17
4.2	Create the files for the Widget.....	19
4.3	Extend the _BaseWidget class.....	20
4.4	Define the Widget's template (user interface) .....	21
4.5	Make it configurable .....	23
4.6	Add i18n support .....	25
4.7	Access a Map.....	28
4.8	Display Widget Graphic Data on Map .....	31
4.9	Receive Data from Map (draw tool).....	32
4.10	Control Navigation from a Widget .....	34
4.11	Retrieve information via a proxy.....	35
<b>5</b>	<b>The Sample JavaScript Viewer Core Code .....</b>	<b>37</b>
5.1	Dojo Publish and Subscribe .....	37
5.1.1	JavaScript Viewer Subscription List .....	38
5.2	Dependency Injection.....	40

5.3     Logging and Error Handling .....42

    5.3.1    djConfig.isDebugEnabled .....42

    5.3.2    Console .....43

# 1 Introduction

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

## 1.1 Prerequisites

### 1.1.1 Skill and Software

The developers who develop Sample JavaScript Viewer based application should have sufficient knowledge and experience using HTML and JavaScript to develop Rich Internet Applications.

The base platform for the ArcGIS JavaScript API and by extension the JavaScript Viewer is Dojo (<http://www.dojotoolkit.org>). At the time of this writing, the current version of Dojo used by the ArcGIS JavaScript API is 1.1. The JavaScript Viewer has an object-oriented JavaScript programming model which takes advantage of the strengths of JavaScript and Dojo, and avoids poor practices. To get the best head start in Widget development, we recommend the following programming texts:

- *Dojo: The Definitive Guide*, 2008. Russell, O'Reilly Books. ISBN 978-0596516482.
- *JavaScript: The Good Parts*, 2008. Crockford, O'Reilly Books. ISBN 978-0596517748.

In addition, to develop a Widget with geo spatial features, a level of familiarity with using the ArcGIS JavaScript API is recommended, or at least familiarity with the samples provided by ESRI.

The only required software is a programmer's text editor, a standards-compliant web browser and an Internet connection. However, the JavaScript Viewer development team recommends:

- ☐ Aptana Studio IDE (Free version), Visual Studio 2008 or TextMate
- ☐ Firefox 3 with the Firebug plugin
- ☐ Tortoise SVN (for source code control, optional)

### 1.1.2 Obtain Source Code and Libraries

The source code release package will be distributed in a zip file and posted in the Code Gallery of ArcGIS JavaScript API. The release package is named by the Code Gallery as *AS15905.zip*, for example.

The zip file contains a folder, **JSViewer**, where all the source files, images and documents will be included. The ArcGIS JavaScript API is dynamically downloaded from <http://serverapi.arcgisonline.com> at runtime.

As HTML and JavaScript is a dynamically interpreted environment, there is no “compiled” version of the JavaScript Viewer<sup>1</sup>. For a local installation of the ArcGIS JavaScript API, the URLs in index.html would need to be adjusted to point to the internal server which is hosting the API. URLs which include “<http://serverapi.arcgisonline.com/jsapi>” would need to be changed to point to the local server.

## 2 Sample JavaScript Viewer Architecture

### 2.1 Overview

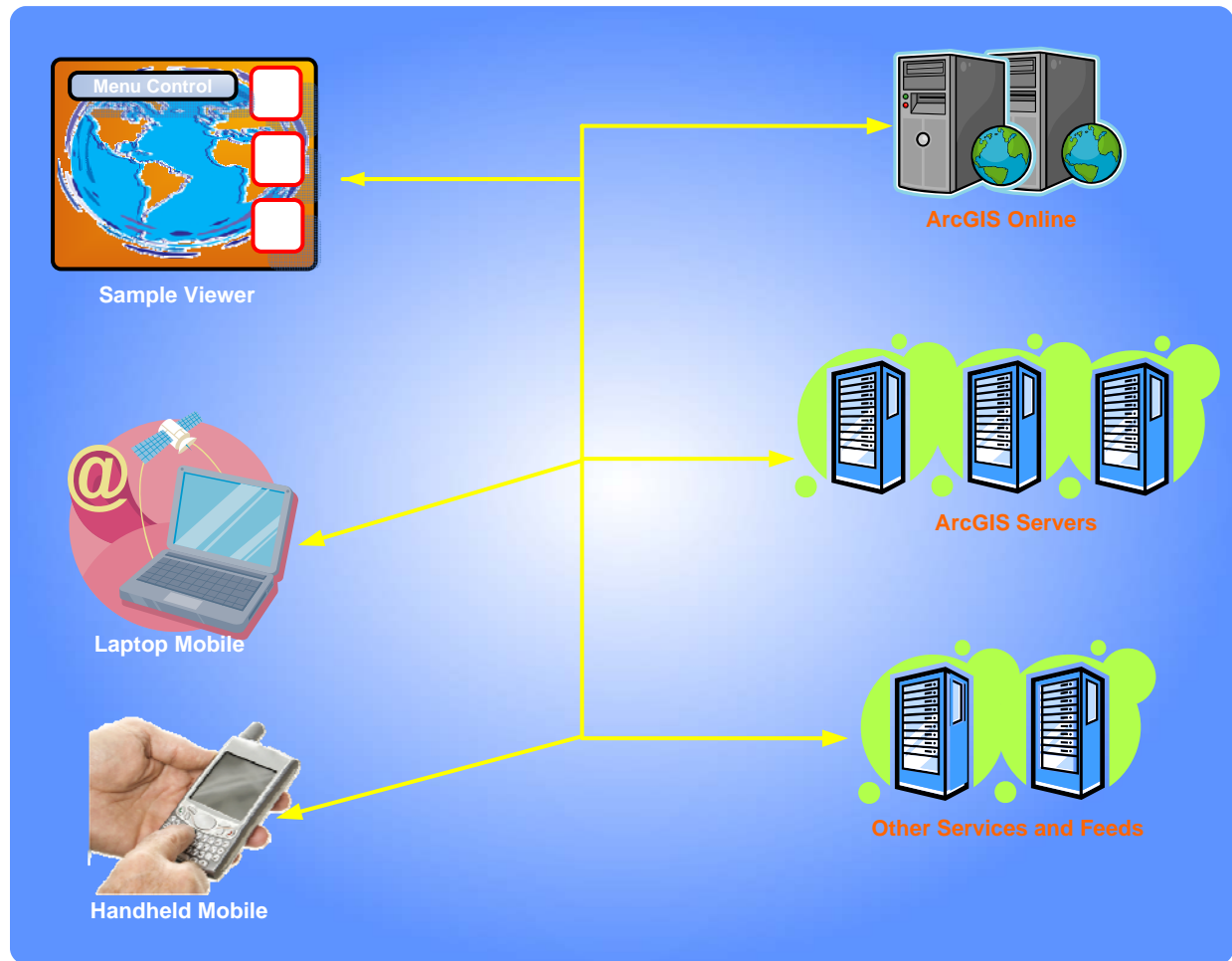
The Sample JavaScript Viewer is architected to help develop and deploy GeoWeb focused applications that can fully leverage the power of the server side spatial services such as those provided by ArcGIS Server and ArcGIS Online.

A Sample JavaScript Viewer application provides users a simple way to access geospatial services. As illustrated below, a geospatial service could come from the hosted SaaS type of provider such as ArcGIS Online, ArcGIS Server or web data sources such as GeoRSS feeds, KML files, JSON/REST data, etc. The data consumed within the Sample JavaScript Viewer application could be the data set at servers or could be generated from mobile devices such as field engineers' laptops or smart phones.

---

<sup>1</sup> The Dojo Toolkit has a build system which allows the application developer to create a compact, efficient version of the application. These tools are available at <http://dojotoolkit.org> and are recommended when putting applications into production.

The Sample JavaScript Viewer is designed to be able to participate into the full spectrum of the geospatial service implementation architecture with the focus on simplicity and flexibility.

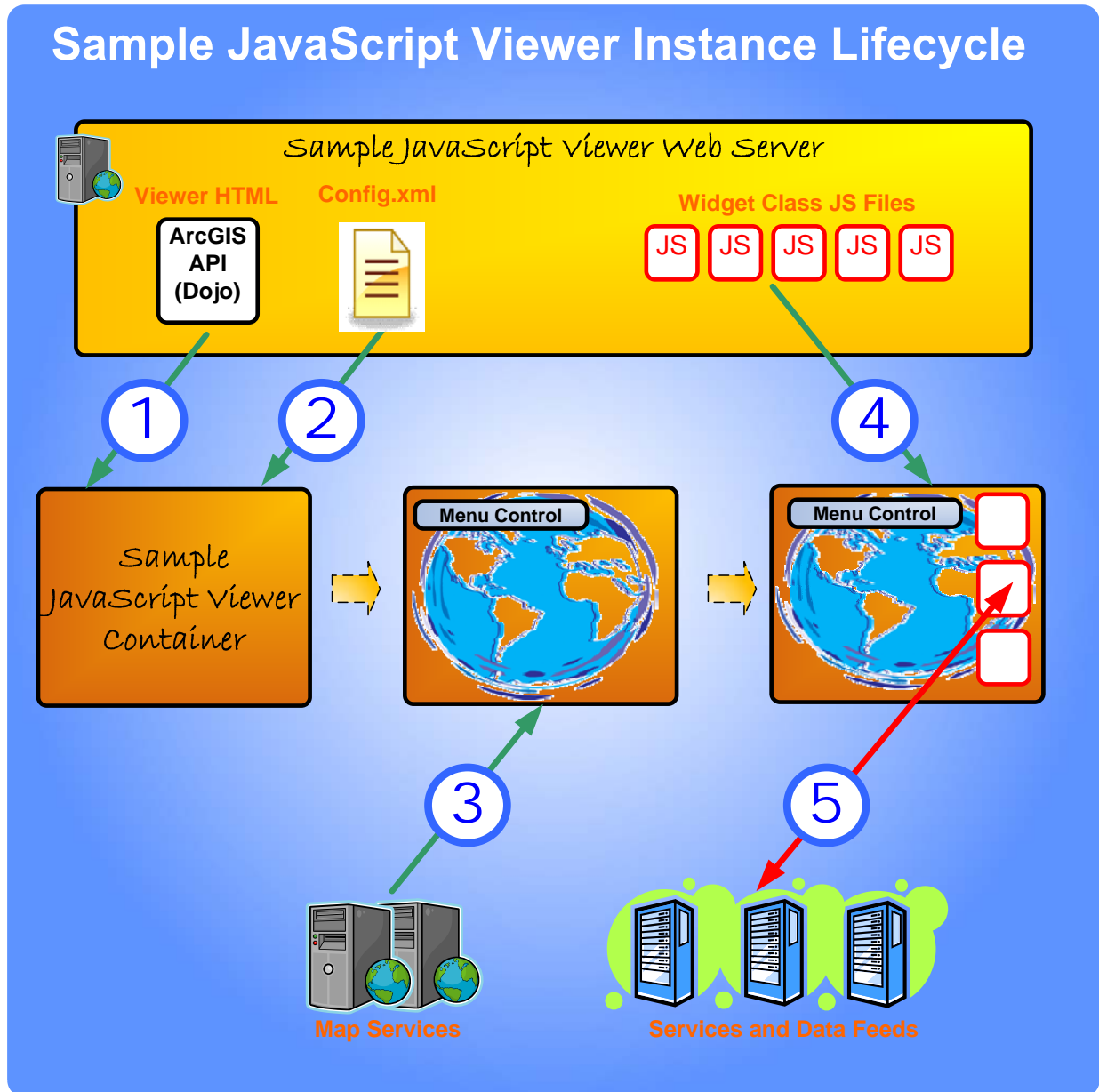


## 2.2 Sample JavaScript Viewer Lifecycle

A JavaScript Viewer application goes through a simple lifecycle from starting the application to the user interacting with Widgets. The five major events during the lifecycle are:

1. ArcGIS JavaScript API (Dojo) is loaded in the HTML page and the page is parsed and the *onLoad* event fires.
2. The JavaScript Viewer application loads the configuration XML file and applies it.
3. Based on the configuration file, the application loads the map layers from map servers such as ArcGIS Online or ArcGIS 9.3 servers. The application also constructs and displays the menu controller with branding information from the config XML file.
4. The Widget manager loads the Widget class files which are specified in the config XML file.

5. The users interact with the Widgets that conduct business logic.



## 2.3 Sample JavaScript Viewer Application

The Sample JavaScript Viewer application takes the programming complexity of managing maps, map navigation, application configuration, inter-component communication, data management, etc. out of developers' hands. It allows the Web developers, especially the developers who are working with ESRI ArcGIS technologies, focus their time and efforts on implementing the core business functions. In addition, the result of the business focused rapid

development, which is in the form of widgets, can be quickly deployed to existing JavaScript Viewer applications by simply adding configuration entry into the JavaScript Viewer application's configuration file.

The following screenshot demonstrates a typical JavaScript Viewer application container.

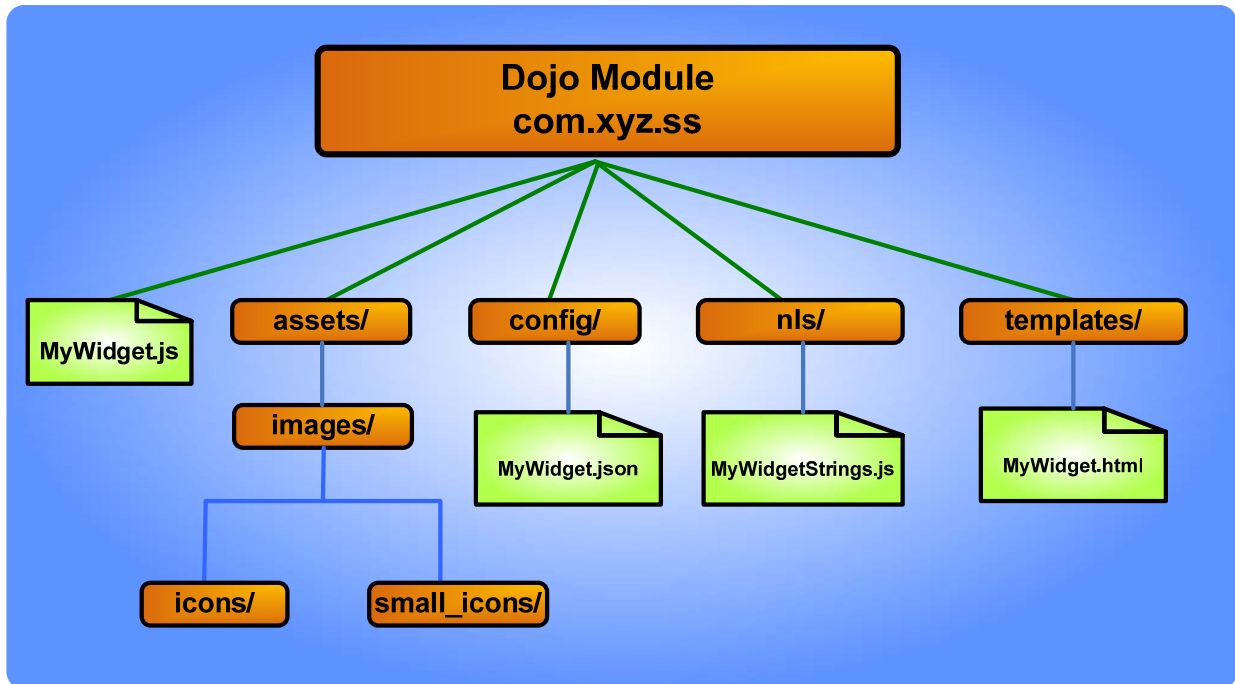


### 2.3.1 Understand Widget and Widget Programming Model

A Widget of the ArcGIS JavaScript Viewer for JavaScript is a set of text files that can be shared, moved and deployed to a JavaScript Viewer application. Generally, a developer might plan to



develop one or more Widgets and would package them in a Dojo module.



A Widget can be as simple as a single JavaScript file. The above structure represents a fully-fledged Widget that has its own icon, template, configuration file and internationalization support.

It will be explained in more detail in later sections during the walkthrough of developing a Widget.

Typically, a Widget encapsulates a set of isolate and focused business logic that allows users conduct a task with it. In a service oriented environment, a Widget represents a service or an orchestration of services that users can clearly execute a business function. A Widget is not only visually interactive with user, but also could connect to server side resources such as Map Services from ArcGIS server or ArcGIS online.

A set of correlated Widgets plus a clearly define business workflow applicable to these Widgets can form 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 JavaScript Viewer allows developers easily develop their Widgets without handling low level detail mechanism of coding to integrate the Widgets into the JavaScript Viewer application.

The Widget programming model contains four JavaScript classes. They are all in the *com.esri.solutions.jsviewer* module, but for brevity will be referred to only by their class name. Here are the short descriptions. The later sections will cover the details of how they are used.

### **\_Widget<sup>2</sup> Class (\_Widget.js)**

This interface defines the communicational methods that will be used by the ***WidgetManager*** to manage the Widgets. Extends the ***dijit.\_Widget***, ***dijit.\_Templated***, ***dijit.\_Container*** and ***dijit.\_Contained*** classes.

### **\_BaseWidget Dijit (\_BaseWidget.js)**

This is the base Widget class that all Widgets should be derived from. By extending the \_BaseWidget class, a new JavaScript class will be recognized by the JavaScript Viewer's *WidgetManager* as a deployable Widget. In addition, extending the base Widget provides the prospective subclass with various low-level functions such as:

- Managing multiple panels and icons for those panels
- Interacting with the *WidgetFrame*
- Loading config data (XML, JSON or plain text)
- Querying child dijit

The \_BaseWidget can be instantiated for demonstration purposes, but it is not intended to be used in that fashion.

A JavaScript Viewer Widget **must** extend the \_BaseWidget class.

### **\_MapGraphicsMaintainer Class (\_MapGraphicsMaintainer.js)**

This class is a mixin<sup>3</sup> class for Widgets which need to add graphics to the map. It defines the common operations of adding graphics to and clearing those graphics from the map.

---

<sup>2</sup> The underscore character “\_” at the beginning of a Dojo classname indicates an abstract class which should be subclassed, not instantiated directly.

**WidgetFrame Dijit (WidgetFrame.js)**

This is the UI class which represents the frames in which Widgets reside in the container application. The *WidgetFrame* is a container dijit, and performs the dynamic resizing, minimizing and other window management code. As a Widget developer, the exact functioning of the *WidgetFrame* is not important, beyond an understanding of how the developer's Widget will arrive in the page and how the HTML template for the Widget will be nested in the page DOM.

## 2.4 Widget Naming Convention

**Widget class:** Recommend that a widget class has suffix "Widget", for example, *SomethingWidget.js*.

**Widget Configuration File:** Recommend that use the same name as Widget class except with an xml or json extension, for example, *SomethingWidget.json*.

**Widget I18N File(s):** Recommend that use the same name as Widget class except with "Strings" appended, for example, *SomethingWidgetStrings.js*.

**Widget template:** Recommend that it use the same name as the Widget class except with an html extension, for example, *SomethingWidget.html*.

---

<sup>3</sup> Dojo classes are multiple-inheritance. Mixin classes add functionality to their subclasses, but are not meant to be sole parent classes.

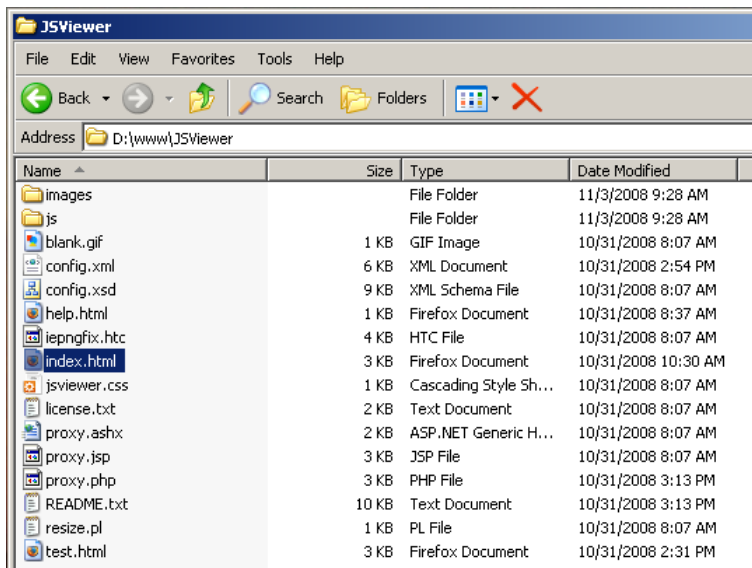
### 3 JavaScript Viewer Setup for Development

This chapter covers much of the same information as in the *README.txt* file which can be found in the *JSViewer* directory you extracted from the JavaScript Viewer ZIP file. This chapter will go into more detail and show how to correctly establish a new module to contain the new Widget.

*Note: Since there is no “compile” step with HTML and JavaScript, there is no discussion of “building” the JavaScript Viewer for deployment. As with any JavaScript, the source can be “minified” to reduce download and parsing time. The JavaScript Viewer is delivered non-minimized.*

#### 3.1 Unzip the source distribution

The zip file, *AS15095.zip* (named by Code Gallery, the number will be different), contains the root folder *JSViewer*. The name of this folder is not important; it can be renamed at will. For clarity, this document will refer to this folder as *JSViewer*. Copy this folder to the documents directory of your web server. As shown on the screenshot, the JavaScript Viewer was unzipped to *d:\www*. On this particular system, that translates to a URL of *http://server/JSViewer*. The path and URL will depend on your server.



The JavaScript Viewer can be deployed to Apache, IIS, or a Java application server such as Tomcat. This document assumes that one of these platforms is available and that it is configured correctly.

At this point, the JavaScript Viewer is capable of being accessed. It is not fully configured, but it can be tested. Access the URL of the JavaScript Viewer in a web browser to check.

## 3.2 Edit config.xml

In the *JSViewer* directory, there is an XML file named *config.xml*. This is the primary configuration point for the JavaScript Viewer. Open it in a plain text editor and modify it as per the information in *README.txt*.

Many URLs in *config.xml* are relative to the Dojo module. Opening *config.xml*, you will note many attributes such as:

- `icon="assets/images/icons/i_about.png"`
- `config="widgets/config/AboutWidget.xml"`

These are partial URLs which allow the JavaScript Viewer to find a file within a Dojo module. In the case of the config file above, on the sample test server the full URL to that file is:

`http://server/JSViewer/js/com/esri/solutions/jsviewer/widgets/config/AboutWidget.xml`

- The **black** part of the URL is where the JavaScript Viewer is installed.
- The **blue** part of the URL is where the JavaScript Viewer Dojo package is located.
- The **red** part of the URL is the part of the URL which is relative to the package.

This allows the JavaScript Viewer to be relocated without having to reconfigure it, as well as saving a lot of typing out the module location.

## 3.3 Configure Widgets

As provided, the JavaScript Viewer has several configured Widgets. These Widgets are all configured with general purpose, publicly available services. Refer to *README.txt* for a discussion of the configuration options for all of the provided Widgets.

The format of the configuration file for individual Widgets is up to the developer. For the Sample Flex Viewer, the best practice is to use XML configuration. Widgets built for the JavaScript Viewer may use either XML or JSON formatted config files. JSON formatted text is very efficiently and simply interpreted in JavaScript. It is recommended that 3<sup>rd</sup> party Widgets use JSON for Widget configuration. For a brief description of JSON:

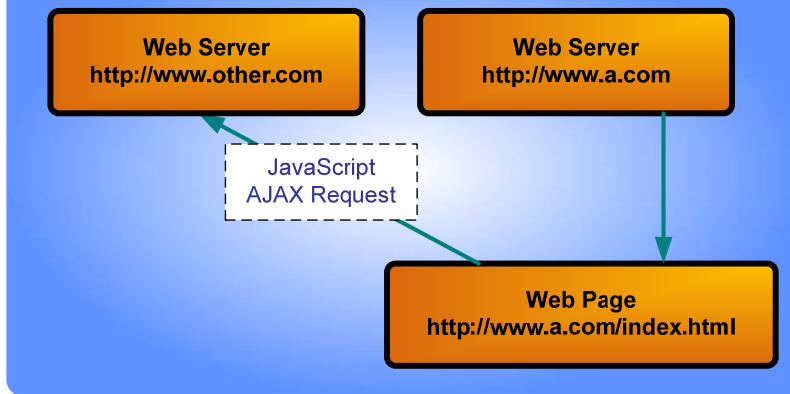
<http://en.wikipedia.org/wiki/JSON>.

### 3.4 Configure Proxy Support

As part of setting up the *config.xml* file, the `<proxytype>` element needed to have its value set to one of “apache”, “php”, “asp” or “jsp”. Some information on which to choose is in *README.txt*, but what follows is a more technical discussion of what this setting means.

As new types of AJAX-model web applications are being developed, there is a balance to be struck between security and functionality. The browser is an inherently insecure platform, and the browser developers attempt to fix this in different ways. One of the risks of the web platform is called a Cross Site Scripting (XSS) attack. XSS involves malicious JavaScript code being downloaded from a supposedly safe site, which when executed downloads additional code from a separate, malicious site.

To guard against this type of a security breach, web browsers do not allow JavaScript to download any additional scripts or data from a server other than the one it was downloaded from itself.

**Allowed****Not Allowed**

Nevertheless, there are many examples of times when the JavaScript Viewer application may wish to retrieve data from a website other than your internal one. For example, a GeoRSS feed from the Internet. To support this, there must be a way to make the request for the external data via the server from which the web page was downloaded. The answer is proxying.

Proxying sets up a relay such that a web request made to one server is forwarded to another, then the other's response is forwarded back to the original requestor. This requires a certain level of functionality to be available on the server. The JavaScript Viewer provides four different mechanisms to do this, based on the type and configuration of web server the JavaScript Viewer is installed on. The choice is made in *config.xml*. For information on how to choose a proxy type, see *README.txt*.

“apache” proxying is done via the `mod_proxy` module of the Apache web server. It is for JavaScript Viewer applications installed on the Apache web server, which do not have `mod_php` (version 5+ with the `cURL` extension) installed. For every external web site which is to be made

available to the JavaScript Viewer, a separate set of ProxyPass and ProxyPassReverse commands must be added to httpd.conf. This has the effect of making the external website appear as a part of your web server. When configuring a Widget to refer to one of these external web sites, enter the URL exactly as you would refer to it directly. The JavaScript Viewer DataManager component handles the responsibility of altering the URL to work through the Apache proxy.

*Note, "apache" proxying is not supported by the ArcGIS JavaScript API for its esri.request() function and for translating large GET requests into POST requests. As a result, "apache" proxying is not recommended, if PHP is available.*

"php" proxying is done via the proxy.php file which is distributed with the JavaScript Viewer. It is for JavaScript Viewer applications which are installed on a web server which has an installed PHP5 module with the cURL extension. The person installing the application must explicitly identify which external servers will be made available via the proxy. Open the proxy.php file and edit the array of permitted web servers in the first few lines of the PHP file. The JavaScript Viewer DataManager is responsible for altering the URL so that it is sent via the proxy, so it is not an issue when configuring the Widget.

"asp" proxying is done via the proxy.ashx file which is distributed with the JavaScript Viewer. It is for JavaScript Viewer applications which are installed on the Microsoft IIS web server, and requires the .NET 2.0 framework. There is no extra configuration to be made with this option. The JavaScript Viewer DataManager is responsible for altering the URL so that it is sent via the proxy, so it is not an issue when configuring the Widget.

"jsp" proxying is done via the proxy.jsp file which is distributed with the JavaScript Viewer. It is for JavaScript Viewer applications which are installed on a Java application server. It has been tested with Tomcat 5.0.28, and should work on newer servers. The administrator must explicitly identify which external servers will be made available via the proxy. Open the proxy.jsp file and edit the array of permitted web servers in the first few lines of the JSP file. The JavaScript Viewer DataManager is responsible for altering the URL so that it is sent via the proxy, so it is not an issue when configuring the Widget.



## 4 Develop a Widget for Sample JavaScript Viewer

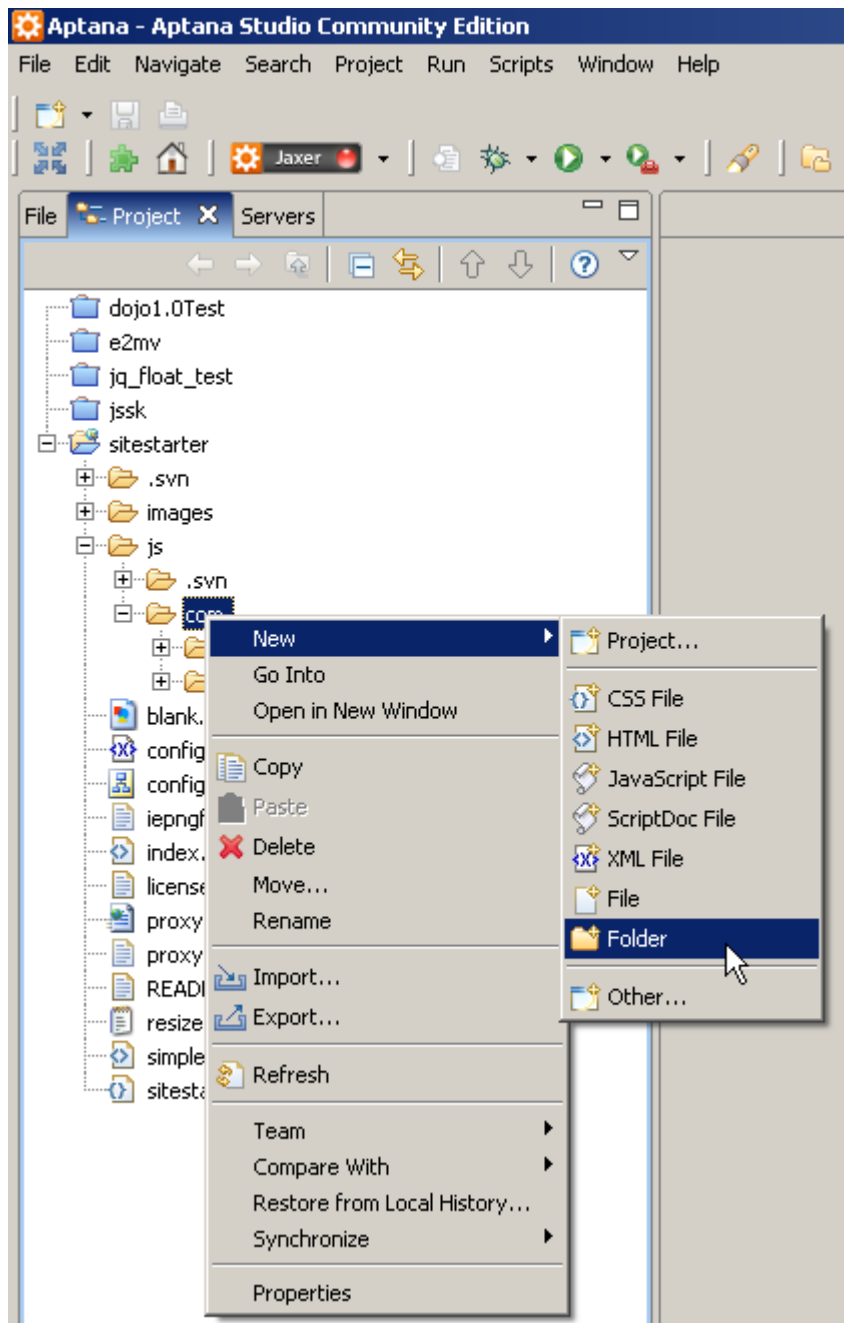
To create a Widget, a new *\_BaseWidget* subclass needs to be created, within a custom Dojo module. Here are the steps. The screenshots shown are from Aptana Studio, after the *JSViewer* directory has been brought into the IDE.

### 4.1 Create the Dojo Module

In order to ensure that your Widget does not interfere with the base JavaScript Viewer code, and will be distributable independent of the Sample JavaScript Viewer, we will create a module for this Widget. The module could be called almost anything, as long as it is not the same as the Dojo module names (dojo, dijit, util, dojox) or the ESRI ArcGIS Javascript API name (esri). The JavaScript Viewer has taken a reverse IP module name approach, similar to that used in Java packages.

We will create the module for the company *XYZCorp*, whose web presence is at *www.xyz.com*. The module will be named "*com.xyz.widgets*".

Right click the *JSViewer/js/com* folder and proceed as following:



Name the new folder “xyz”. Then right-click on the new xyz folder and create a subfolder named “widgets”. The structure should be *JSViewer/js/com/xyz/widgets*.

Open the *index.html* file in the JSViewer folder.

There is a bit of JavaScript which defines the “*djConfig*” object. This is a Dojo construct, and it will allow us to define our new module and have it recognized by the system.

Change this part:

```
modulePaths:{
  'com.esri.solutions.jsviewer':'js/com/esri/solutions/jsviewer'},
```

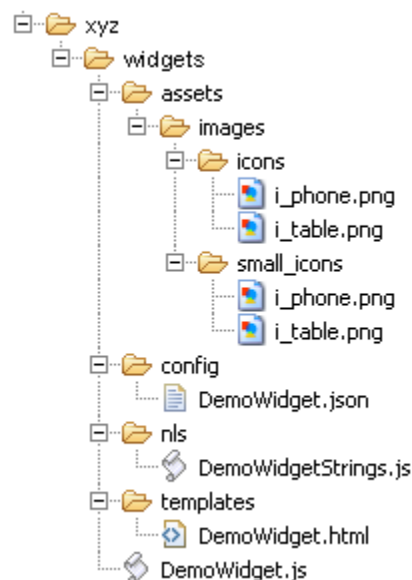
to:

```
modulePaths:{
  'com.esri.solutions.jsviewer':'js/com/esri/solutions/jsviewer',
  'com.xyz.widgets':'js/com/xyz/ss'},
```

## 4.2 Create the files for the Widget

A Widget at a minimum requires one file: the JavaScript file which defines it. However, for a more full-featured Widget, there will be multiple files to define it, including its HTML template, i18n support, icons, etc. We will demonstrate all of the above.



Within the *widgets* folder you created at the last step, create the following set of folders and files:



- Within *widgets/* is the JavaScript file which will define the Widget (*DemoWidget.js*).

- Within *assets/images/icons/* are the icons we will use for the Widget. These icons are 40

pixel by 40 pixel PNG files:  , 

- Within *assets/images/small\_icons/* are the smaller 20x20 versions of the full sized icons of the same name:  , 
- Within *config/* is the JSON configuration file for the Widget (*DemoWidget.json*).
- Within *nls/* is the i18n strings file for the Widget (*DemoWidgetStrings.js*).
- Within *templates/* is the HTML template for the Widget (*DemoWidget.html*).

### 4.3 Extend the `_BaseWidget` class

JavaScript itself is not a classical language. It is a prototypical object oriented language. As such, in order to make a wide array of developers comfortable with developing Widgets, the JavaScript Viewer uses Dojo's functions to create and extend classes. The JavaScript Viewer defines the `_BaseWidget` class from which we will create our child class, *DemoWidget*.

Open *com/xyz/widgets/DemoWidget.js* in a text editor. The basic code for extending `_BaseWidget` is:

```
dojo.provide("com.xyz.widgets.DemoWidget");
dojo.require("com.esri.solutions.jsviewer._BaseWidget");

dojo.declare(
    "com.xyz.widgets.DemoWidget",
    com.esri.solutions.jsviewer._BaseWidget,
    {
        // DemoWidget code goes here
    }
);
```

This code:

1. Tells what class this source code provides
2. Requires the availability of the superclass
3. Declares the *DemoWidget* class as a child class of `_BaseWidget`

All of the custom code which differentiates *DemoWidget* from *BaseWidget* will be inserted between the curly braces. We'll start with the basics, selecting the correct HTML template.

```
{
    // DemoWidget code goes here
    _module: "com.xyz.widgets",
    templatePath: dojo.moduleUrl("com.xyz.widgets", "~
        templates/DemoWidget.html")
}
```

## 4.4 Define the Widget's template (user interface)

JavaScript Viewer Widgets have an HTML template. This template defines how they look and to some extent behave within the generic *WidgetFrame*. A Widget can have a single panel, like the *AboutWidget* or the *OverviewWidget*, or they can have multiple panels, like the *LocateWidget*. These panels are all defined in the Template.

Open the *DemoWidget.html* file in a text editor. Your IDE may have filled your file with a template HTML document. Clear it, and replace it with this:

```
<div class="widgetContent">
    <div class="widgetPanel"
        buttonIcon="assets/images/small_icons/i_phone.png"
        buttonText="Panel 1">
        <p>This is panel 1</p>
    </div>
    <div class="widgetPanel"
        buttonIcon="assets/images/small_icons/i_table.png"
        buttonText="Panel 2">
        <p>This is panel 2</p>
        <p>Because it has more content, it is larger than panel 1</p>
    </div>
</div>
```

*Note, the class names on the div elements are important! They are used by the WidgetFrame class to correctly manage the visual behavior of the Widget.*

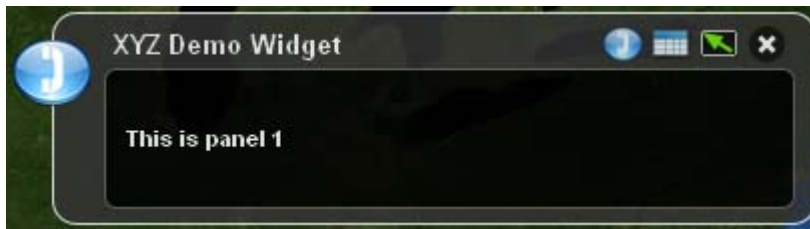
At this point, we can test our Widget. Open config.xml, and in the widgets section, add a new widget element like this:

```
<widget label="XYZ Demo Widget" icon="assets/images/icons/i_phone.png"
menu="menuWidgets">com.xyz.widgets.DemoWidget</widget>
```

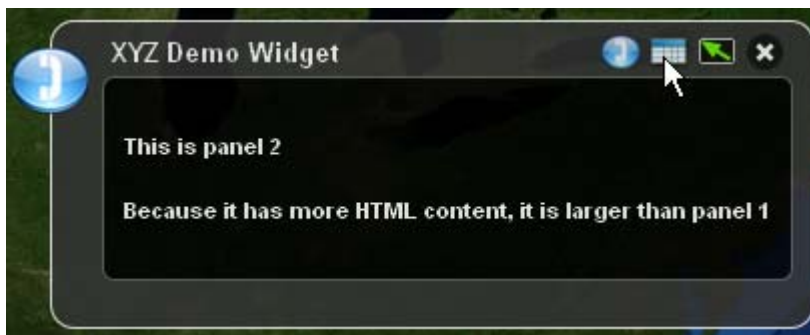
Launch the JavaScript Viewer. *DemoWidget* will appear in the Widgets menu:



And selecting it will bring it up:



Clicking on the second icon that looks like a table switches to the second panel of the Widget:



Hovering over the buttons gives us Tooltips. The *WidgetFrame* behaves as expected, cooperating with the other *WidgetFrames*, and able to maximize, minimize and close.

## 4.5 Make it configurable

At the moment, there are no configuration options for *DemoWidget*, other than the label, icon and which menu it is found in, which are defined in the main *config.xml*. Let's change that.

Open *config/DemoWidget.json* in a text editor. We will enter some JSON-structured text so that we can pass a URL to our widget:

```
{
  demo: {
    url: http://www.esri.com,
    linkText: "ESRI.com"
  }
}
```

We will alter the widget entry in *config.xml* to add a “config” attribute, which points to our JSON file:

```
<widget label="XYZ Demo Widget"
  icon="assets/images/icons/i_phone.png"
  menu="menuWidgets"
  config="config/DemoWidget.json">com.xyz.widgets.DemoWidget</widget>
```

Reopen the *DemoWidget.js* file which defines our Widget. Add a new function to *DemoWidget*, after our *templatePath* definition:

```

{
    linkText: "",
    url: "",

    _module: "com.xyz.widgets",
    templatePath: dojo.moduleUrl("com.xyz.widgets", "
        templates/DemoWidget.html"),

    postMixinProperties: function() {
        this.inherited(arguments);

        if (this.configData) {
            this.url = this.configData.demo.url;
            this.linkText = this.configData.demo.linkText;
        }
    }
}

```

“*postMixinProperties*” is part of the Dojo Widget lifecycle, and it occurs at a time when it is good to read and react to the Widget configuration. The Widget lifecycle is discussed here:

<http://dojotoolkit.org/book/dojo-book-0-9/part-3-programmatic-dijit-and-dojo/writing-your-own-widget-class/widget-life-cycl>.

Note that we did not have to write any code to fetch the information from the config URL, that is handled for us in *\_BaseWidget*.

The last step will be to alter the HTML template to make use of the URL which we have read from our config file.



```

<div class="widgetContent">
  <div class="widgetPanel"
    buttonIcon="assets/images/small_icons/i_phone.png"
    buttonText="Panel 1">
    <p>Go to <a href="${url}">${linkText}</a></p>
  </div>
  <div class="widgetPanel"
    buttonIcon="assets/images/small_icons/i_table.png"
    buttonText="Panel 2">
    <p>This is panel 2</p>
    <p>Because it has more content, it is larger than panel 1</p>
  </div>
</div>

```

Now, Panel 1 of *DemoWidget* looks like this:



The `${url}` and `${linkText}` markers in the template were automatically substituted with the values of *this.url* and *this.linkText*.

## 4.6 Add i18n support

An important part of any code which might be accessed by non-english speaking users is the ability to internationalize the user interface. Currently in *DemoWidget*, there are some hard-coded English language strings:

- "Panel 1" (on the phone button)
- "Panel 2" (on the table button)
- "Go to"
- "This is Panel 2"
- "Because it has more content, it is larger than panel 1"

What we want to do is isolate these strings in a separate language-specific resource file, and then refer to the strings by their identity. Dojo provides complete i18n support, which is sensitive to the browser setting of the user. So if we define a Spanish translation, and a user visits your JavaScript Viewer web application with their browser language code set to “es”, they will automatically get the Spanish user interface.

*Note: For unknown reasons, Internet Explorer users have reported that the correct internationalization strings were not used, despite browser language settings. This appears to be browser-specific. To override the browser locale, add a “locale” attribute to djConfig, specifying the correct NLS code. All users will get this localization, regardless of browser settings.*

We will do the first step: abstract the user interface strings to the default resource file. Translation of these strings is left to the user. See the translations for the ESRI-provided widgets for examples.

First, open the `nls/DemoWidgetStrings.js` file with a text editor. Enter this:

```
{  
    panelTitle1: "Panel 1",  
    panelTitle2: "Panel 2",  
  
    msgFollowLink: "Go to",  
    msgPanel1: "This is Panel 2",  
    msgPanel2: "Because it has more content, it is larger than panel 1"  
}
```

Now, reopen `DemoWidget.js`, and make some changes to support i18n:

```

dojo.provide("com.xyz.widgets.DemoWidget");
dojo.require("com.esri.solutions.jsviewer._BaseWidget");
dojo.require("dojo.i18n");
dojo.requireLocalization("com.xyz.widgets", "DemoWidgetStrings");

dojo.declare(
    "com.xyz.widgets.DemoWidget",
    com.esri.solutions.jsviewer._BaseWidget,
    {
        linkText: "",
        url: "",

        _module: "com.xyz.widgets",
        templatePath: dojo.moduleUrl("com.xyz.widgets", "
            templates/DemoWidget.html"),

        i18nStrings: null,

        postMixInProperties: function() {
            this.inherited(arguments);

            if (this.configData) {
                this.url = this.configData.demo.url;
                this.linkText = this.configData.demo.linkText;
            }

            this.i18nStrings = dojo.i18n.getLocalization(
                this._module, "DemoWidgetStrings");
        }
    }
);

```

This populates the i18nStrings variable with all of the information in the resource file **in the correct language**, if available. If you have not bothered to make an “en-gb” resource file, then

the “en” resource will be returned, if available. If there is no “en” resource, then it will default to the resource at the root of the *nls/* folder, which is what we have defined.

The last step is to apply the changes to the template, to remove the hard-coded English and replace it with markers which will be automatically substituted:

```
<div class="widgetContent">
  <div class="widgetPanel"
    buttonIcon="assets/images/small_icons/i_phone.png"
    buttonText="${i18nStrings.panelTitle1}">
    <p>${i18nStrings.msgFollowLink}
      <a href="${url}">${linkText}</a>
    </p>
  </div>
  <div class="widgetPanel"
    buttonIcon="assets/images/small_icons/i_table.png"
    buttonText="${i18nStrings.panelTitle2}">
    <p>${i18nStrings.msgPanel1}</p>
    <p>${i18nStrings.msgPanel2}</p>
  </div>
</div>
```

Reload your JavaScript Viewer application, and it will look much as before, but if you alter the *DemoWidgetStrings.js* file, those changes will be reflected.

To support specific languages, create folders under the *nls/* folder, named by the language code (e.g. “en”, “fr”, “ru”). Copy the *DemoWidgetStrings.js* file to that folder, and update the content to reflect the language for that code:

```
msgFollowLink: "Allez a"
```

## 4.7 Access a Map

The JavaScript Viewer is a map-centric application. It means that there are always one or more maps (from ArcGIS map services) active once the JavaScript Viewer application starts.

There is a public property, *map*, defined in the class *\_Widget*. Once a Widget is loaded, the *WidgetManager* will pass the current active map object reference to the Widget. The map property of Widget is a type of *esri.Map*, which is from the ArcGIS JavaScript API. Thus, you can access all the map features provided by ArcGIS JavaScript API. The Widget code will be written just like writing a regular ArcGIS JavaScript application.

Here is an example where a button added to *DemoWidget*. When the button is clicked, the map will be set to center at a new location (Los Angeles). A new string has been added to *DemoWidgetStrings.js* (*btnRecenter*: "Recenter Map"). The HTML template has been adjusted to add the button below the <p> on the first panel:

```
<button name="btnRecenter" type="button"
        dojoType="dijit.form.Button">${i18nStrings.btnRecenter}</button>
```

The DemoWidget.js class has the most changes:

```
dojo.provide("com.xyz.widgets.DemoWidget");
dojo.require("com.esri.solutions.jsviewer._BaseWidget");
dojo.require("dojo.i18n");
dojo.requireLocalization("com.xyz.widgets", "DemoWidgetStrings");
dojo.require("dijit.form.Button");

dojo.declare(
    "com.xyz.widgets.DemoWidget",
    com.esri.solutions.jsviewer._BaseWidget,
    {
        linkText: "",
        url: "",

        _module: "com.xyz.widgets",
        templatePath: dojo.moduleUrl("com.xyz.widgets", "templates/DemoWidget.html"),

        i18nStrings: null,

        postMixInProperties: function() {
            this.inherited(arguments);
            if (this.configData) {
```

```

        this.url = this.configData.demo.url;
        this.linkText = this.configData.demo.linkText;
    }
    this.i18nStrings = dojo.i18n.getLocalization(
        this._module, "DemoWidgetStrings");
},

postCreate: function(){
    this.inherited(arguments);
    dojo.parser.parse(this.domNode);
},

startup: function() {
    this.inherited(arguments);
    if (this._initialized) { return; }

    this.getAllNamedChildDijits();
    this.connects.push(
        dojo.connect(this.widgets.btnRecenter, "onClick",
            this, "onRecenter"));
},

onRecenter: function() {
    var p = new esri.geometry.Point(-118.24799,33.975004);
    this.map.centerAt(p);
}
}
);

```

Of special note is the line in `postCreate`:

```
dojo.parser.parse(this.domNode);
```

This line is necessary, since we defined part of our Widget UI to use Dojo dijits (the input tag with a “dojoType” attribute). In order to parse the template and substitute the Dojo-specific items in it, we need to manually call the Dojo parser on our template. The right time to do this is in *postCreate*.

Here is what you will get after reloading the JavaScript Viewer application:



Clicking the button has the effect of panning the map to center on Los Angeles, California.

## 4.8 Display Widget Graphic Data on Map

Since adding graphics to the map is an operation that many Widgets have in common, it has been separated into its own mixin class, `_MapGraphicsMaintainer`. Currently our class declaration looks like this:

```
dojo.declare(
    "com.xyz.widgets.DemoWidget",
    com.esri.solutions.jsviewer._BaseWidget,
    {
        ...
    }
);
```

What we want to do is add `_MapGraphicsMaintainer` as a mixin class:

```
dojo.require("com.esri.solutions.jsviewer._MapGraphicsMaintainer");
dojo.declare(
    "com.xyz.widgets.DemoWidget",
    [com.esri.solutions.jsviewer._BaseWidget,
     com.esri.solutions.jsviewer._MapGraphicsMaintainer],
    {
        ...
    }
);
```

This adds some useful functions to our class:

- `addGraphic(*esri.Graphic* g)`
- `clearGraphics()`
- `onMapClick(evt)`

The actual mechanics of the maintenance of the Graphics in the map are handled by `_MapGraphicsMaintainer`.

### Clear Graphics when Widget closed

The Dojo *dijit.\_Widget* class defines two lifecycle methods: *startup* and *shutdown*. When a Widget is opened, *startup* is called, and when it is closed, *shutdown* is called.

We already demonstrated startup in the previous section, where we used it to hook up our button event handler. We need to add a shutdown handler if we wish to add behavior to our shutdown procedure, like clearing the graphics:

```
shutdown: function() {  
    this.clearGraphics();  
    this.inherited(arguments);  
},
```

The “this.inherited(arguments)” line is similar to calling “super” in Java or “base” in .NET.

## 4.9 Receive Data from Map (draw tool)

For GIS web application, in addition to visualize data, the map is a data source. There will be cases where the Widget needs to receive point (click), polygon, polyline, etc information generated by user on the map. The JavaScript Viewer and its Widget Programming model allow a widget request and receive such map data via events.

Here is an example code segment that sends an event to ask JavaScript Viewer’s Map Manager to active the draw tool for collecting user’s click point on map (*btnDraw* is a named *dijit.form.Button* which has been placed inside a *widgetPanel* div in the template):



```

startup: function() {
    ...
    this.connects.push( ↵
        dojo.connect(this.widgets.btnDraw, "onClick", ↵
            this, "onDraw"));
},
shutdown: function() {
    this.clearGraphics();
    dojo.publish("widgetDrawRequestEvent", [null]);
    this.inherited(arguments);
},
onDraw: function() {
    var params = {
        geometryType: esri.toolbars.Draw.POINT,
        onDrawEnd: dojo.hitch(this, function(geometry) { ↵
            this.addGeometry(geometry); }),
        label: "Draw Point"
    };
    dojo.publish("widgetDrawRequestEvent", [params]);
},
addGeometry: function(geometry){
    if (geometry) {
        var outline = new esri.symbol.SimpleLineSymbol( ↵
            esri.symbol.SimpleLineSymbol.STYLE_SOLID, ↵
            new dojo.Color([255,0,255]), 1);
        var symbol = new esri.symbol.SimpleMarkerSymbol( ↵
            esri.symbol.SimpleMarkerSymbol.STYLE_CIRCLE, ↵
            10, outline, new dojo.Color([255,0,255,0.5]));
        var graphic = new esri.Graphic(geometry, symbol);
        this.addGraphic(graphic);
    }
}
}

```

In the above example, an event is published which is requesting that the map's tool mode be set to drawing points, and the callback function for drawing those points is the *addGeometry* function on this object. When the point is returned, the Widget can do anything it wants with the

geometry. In this case, we are building on the previous section and showing how to add and clear graphics from the map. The use case could be that when the user is ready to send a point (such as in an identify task) to this widget, the user clicks the button. Then the user will just click the map and the point information will be received by this Widget.

Within the data structure passed to the MapManager:

***geometryType***: the string token indicates which of the ArcGIS JavaScript API's draw tools to be active. The constants are defined in *esri.toolbars.Draw*:

**EXTENT**  
**FREEHAND\_POLYGON**  
**FREEHAND\_LINE**  
**LINE**  
**MULTI\_POINT**  
**POINT**  
**POLYGON**  
**POLYLINE**

***onDrawEnd***: the callback function that the MapManager will call once the drawing is done.

***label***: the text which will appear in the status area of the application's banner when the Widget tool is active. Here is an example when the Identify tool is active:



For a list of all intra-application messages like "*widgetDrawRequestEvent*", see Section 5.1.1.

## 4.10 Control Navigation from a Widget

When developing a widget, there are cases that you might want to control the map navigation such as to enable pan, zoom in, zoom out, etc.

For example, when a widget is closed, you want to make sure the map is no longer in a drawing mode and enable the pan tool for the map. Here is the code:

```
dojo.publish("widgetNavRequestEvent", [esri.toolbars.Navigation.PAN]);
```

As with the draw event, it is the *MapManager* which is listening for the *widgetNavRequestEvent*. In the above code, you publish a “*widgetNavRequestEvent*” and provide the constant for the Pan tool. Here is the list of map navigation tool constants defined in *esri.toolbars.Navigation*:

**PAN**

**ZOOM\_IN**

**ZOOM\_OUT**

## 4.11 Retrieve information via a proxy

As discussed in Section 3.4, if the JavaScript Viewer Widget needs to retrieve data from an Internet web site which is not the same web site that the JavaScript Viewer application was downloaded from, we need to use the JavaScript Viewer proxy, whichever type of proxy was configured in *config.xml*.

Normally, a Dojo application might use code like the following:

```
var params = {
    url: "http://www.esri.com/somedata.json",
    handleAs: "json",
    load: dojo.hitch(this, "onLoad"),
    error: dojo.hitch(this, "onError")
};
dojo.xhrGet(params);
```

However, assuming that the JavaScript Viewer application did not get downloaded from [www.esri.com](http://www.esri.com), this XMLHttpRequest (XHR) would fail, because the browser guards against XSS attacks. In order to safely retrieve this via the proxy, all of the parameters are the same, just instead of calling `dojo.xhrGet` directly, publish an event like the following:

```
var params = {
    url: "http://www.esri.com/somedata.json",
    handleAs: "json",
    load: dojo.hitch(this, "onLoad"),
    error: dojo.hitch(this, "onError")
};
```

```
dojo.publish("dataRequestEvent", [params]);
```

The *DataManager* reacts to the *dataRequestEvent* and alters the URL to match the *proxytype* defined in *config.xml*. It then makes the *dojo.xhrGet* request via the proxy. The callback functions in the *params* object are called just as they would have if the Widget had called *dojo.xhrGet* directly. This is the JavaScript Viewer solution to making the use of the defined proxy as transparent as possible.

## 5 The Sample JavaScript Viewer Core Code

The Sample JavaScript Viewer source code is freely available to public. As long as following the license agreement (refer to the **license.txt** file inside the release package), there is no restriction on customizing and extending the viewer's core functionalities.

The Sample JavaScript Viewer is designed specifically for easy customization and extending. There are two design approaches enable the high cohesion and low coupling composite application architecture, Dojo Publish & Subscribe and the utilization of Dependence Injection (DI). To customize the Sample JavaScript Viewer core, it's important to understand the two approaches.

### 5.1 Dojo Publish and Subscribe

A key aspect of the JavaScript Viewer is the lack of internal references between the various modules. As any application grows in complexity, the number of interconnections between the various parts increases exponentially. One effective means to reduce the number of interconnections is to have a notification-based internal communications scheme. When one part of the application wishes to influence another part, it sends a message on a generic event bus, and all other parts of the application which need to react to that message subscribe to that type of message independently.

This means that module A can communicate with module B, but neither A nor B hold a reference to the other.

The Dojo toolkit's implementation of a notification is called Publish and Subscribe. Objects which wish to be notified of a particular class of message subscribe to it. Then objects wishing to send that type of message publish them.

The Dojo Book has a section on Publish and Subscribe here: <http://dojotoolkit.com/book/dojo-book-0-9/part-3-programmatic-dijit-and-dojo/event-system/publish-and-subscribe-events>

The JavaScript Viewer programming model is *loosely coupled*. The various parts (ConfigManager, MapManager, Widgets, etc.) do not hold references to each other (exception: WidgetManager has references to all instantiated Widgets). When communication has to take place between parts of the JavaScript Viewer, the way to do it is via Dojo notifications. If return communication is required (like with drawing), a callback is passed in the event message.

### 5.1.1 JavaScript Viewer Subscription List

The JavaScript Viewer uses Publish and Subscribe extensively to pass application events in a decoupled fashion. Section 4.9 covered a sample graphic draw tool, which used the widgetDrawRequestEvent. In that case, the DemoWidget published the message, and it was the MapManager which had subscribed to the event. The effect was that the MapManager activated the drawing tool. Event names are spelling- and case-sensitive! Different parts of the JavaScript Viewer are subscribed to different kinds of events:

- Controller:
  - configLoadedEvent
  - mapLoadedEvent
  - mapToolChangedEvent
  - statusChangedEvent
- DataManager:
  - configLoadedEvent
  - dataRequestEvent
  - widgetLocationsChangedEvent
  - locationsRequestEvent
- MapManager:
  - configLoadedEvent
  - menuItemClickedEvent
  - widgetHighlightEvent
  - widgetNavRequestEvent
  - widgetDrawRequestEvent
  - mapResizeRequestEvent
- WidgetContainer:
  - showWidget (*specific for WidgetManager to WidgetContainer*)
  - mapResizedEvent

- WidgetManager:
  - configLoadedEvent
  - mapLoadedEvent
  - menuItemClickedEvent
- ServiceAreaWidget:
  - knownLocationsChangedEvent

*Note: Subscriptions for the \*Manager and other core dijit occur in their postCreate event handler. The objects which subscribe to configLoadedEvent unsubscribe from that event after it is called, to prevent re-configuration.*

Any part of the JavaScript Viewer can publish events of any of these types, but it may not make sense to do so. Some events have a single use, such as the mapResizedEvent. This is raised when the JavaScript Viewer is in full-page mode and the browser window changes size. The best way to demonstrate the use of these events is in the existing JavaScript Viewer source.

Events which would be likely candidates to be raised by Widgets, in detail:

- **statusChangedEvent:** This message is listened to by the Controller, the Dijit which displays the banner and menus across the top of the application. Sending this event with a String argument updates the status field in the banner.
- **dataRequestEvent:** As discussed in Section 4.11, the recommended method for Widgets to make cross-domain AJAX data requests is to form the parameters for an XMLHttpRequest as per dojo.xhrGet(), then pass those parameters in a dataRequestEvent. The JavaScript Viewer's DataManager is subscribed to this event, and handles the translation of the request to the proxy. The callbacks passed in the event are called exactly as though the dojo.xhrGet had been called directly.
- **widgetLocationsChangedEvent:** This event is to be raised specifically by Widgets which create locations, usually by downloading a data feed or performing some type of analysis. In the 1.0 release of the JavaScript Viewer, the GeoRSSWidget, LocateWidget and LiveLayerWidget all publish this event when they receive new information. This allows their locations (such as a geocoded address) to be shared with other Widgets. The ServiceAreaWidget uses this functionality by subscribing to the knownLocationsChangedEvent.

- **locationsRequestEvent:** This event is similar to the widgetLocationsChangedEvent. Where a call to widgetLocationsChangedEvent will cause the known locations to be pushed out to all listeners, the locationRequestEvent parameters have a callback which is called, passing the known locations. This is a “pull” of data, rather than triggering a “push” to all Widgets which subscribe to widgetLocationsChangedEvent. The ServiceAreaWidget publishes this event on startup to populate its list of known locations.
- **widgetHighlightEvent:** This event causes the InfoPopup to appear in the map, as well as the red highlight. It is common for all Widgets which maintain a list of geographic results to publish this event. For Widgets which mixin \_MapGraphicsMaintainer, the connection between clicking on a graphic in the map and the publishing of widgetHighlightEvent is automatic.
- **widgetNavRequestEvent:** This event is used to change the map navigation mode to one of the esri.toolbars.Navigation constants. This event is not as common as the widgetDrawRequestEvent. Passing a null parameter in the event reverts the map navigation to the default mode.
- **widgetDrawRequestEvent:** This event is used to change the map navigation mode to one of the esri.toolbars.Draw constants. This event is common, as it is used by any Widget which wishes to use mouse interaction with the map. The key to this event is passing a callback which is called when the digitization of the geometry is complete. Then the Widget can process the geometry as appropriate, like building a spatial query or adding a graphic to the map. Passing a null parameter in the event reverts the map navigation to the default mode.

## 5.2 Dependency Injection

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.



By contrast, 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 JavaScript Viewer, the alternative implementations are the widgets. DI is also used to create custom widget template. The following code shows how DI is used by WidgetManager to load widgets into the containers without knowing what specific implementations of the widgets are.

```
loadWidget: function(label) {
    var defn = this.widgetDefinitions[label];
    var paramStr = "";
    if (defn.config) {
        paramStr = "{ config: '" + defn.config + "'}";
    }

    var loadStr = "var w = new " + defn.widgetType + "(" + paramStr + ")";
    eval(loadStr);

    w.setTitle(defn.label);
    w.setIcon(defn.icon);
    w.setConfig(defn.config);
    w.setMap(this.map);

    this.widgets[label] = w;
}
```

In above code, once the widget module is loaded, it will be treated as though it is a subclass of `_Widget`. The `Widget` extends `_Widget` just for this DI usage. The `WidgetManager` doesn't need to know what kind of `Widget` it is or what it does. The `WidgetManager` only need to call the methods defined in the `_Widget` to communicate with the loaded `Widget`. Therefore, the implementation of a `Widget`, regardless how complicated it could be, is totally independent of

the WidgetManager or the whole JavaScript Viewer container. The developer can freely develop a Widget without constrained by the JavaScript Viewer. It's the recommended best practice to maintain such flexibility through DI when you customize the Sample JavaScript Viewer core. This allows you continue build and increase the functionality horizontally, meaning to just add Widgets instead of create features right into the core. This approach keeps your custom application simple and lean and avoids increased complexity from added features.

## 5.3 Logging and Error Handling

As a browser-based application, the JavaScript Viewer has access to the default logging mechanism: the browser's error console. The implementation varies from browser to browser, but generally it is available.

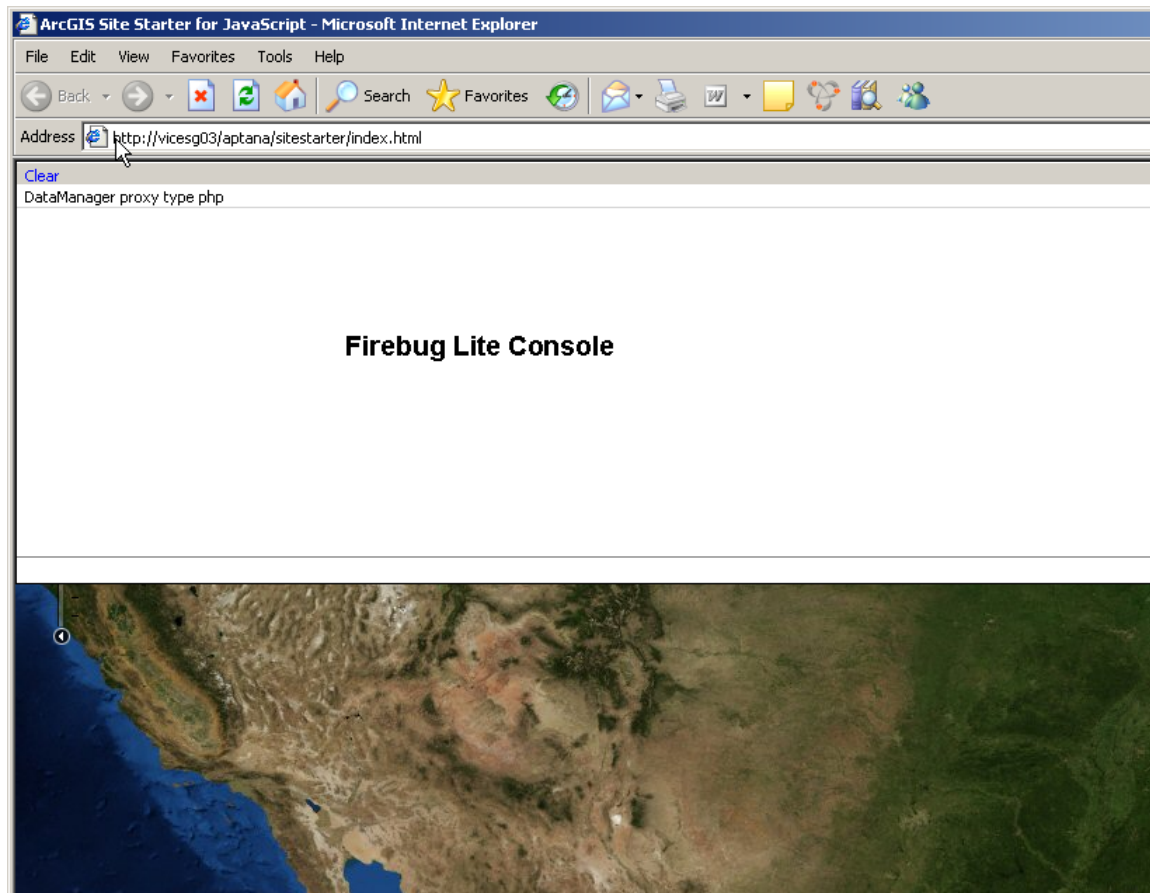
*Note: It is recommended that development take place using Firefox with the Firebug extension, as it provides the most comprehensive browser-based tools available. All new functionality or new installations should be established and initially tested using these tools for best results.*

### 5.3.1 djConfig.isDebug

In **index.html**, in the section where the djConfig object is configured, there is an option called "isDebug" which is set to false in the default distribution of the Sample JavaScript Viewer.

Setting this value to **true** has the following effects:

- In Firefox, with Firebug installed and enabled: no effect
- In all other browser configurations, Firebug Lite is enabled, and the console is displayed using HTML and JavaScript in the browser window itself:



Setting `isDebug` to true ensures that the console is available in every browser, at the expense of making the app mostly unusable, so set `isDebug` to false before deploying.

### 5.3.2 Console

When developing new functionality or coping with a troublesome deployment, you can add logging statements to the JavaScript code to help diagnose problems. There are several places in the JavaScript Viewer code where logging has been commented out to reduce the “noise” in the console. As an aid to installing or developing, these lines may be uncommented, and/or new logging statements added.

The basic usage of the console is to use one of the following methods: `log`, `debug`, `dir` or `error`.

- `console.log(message)`: basic logging. Well supported by Firefox and Safari even when `isDebug` is false. Prints a basic message out to the console screen.
- `console.debug(message)`: temporary logging. Unless `isDebug` is true (i.e. Firebug Lite is running) or Firebug is enabled, this will cause the JavaScript Viewer to halt, since `debug`

is a Firebug method, not a browser method. Always remove or comment console.debug statements prior to deploying.

- console.dir(object): very useful for developing with Firebug. Writes out all properties of the object to the console. Complex properties (like nested objects) can be expanded right in the console. Limitation similar to debug: Always remove or comment console.dir statements prior to deploying.
- console.error(message): an error has been caught. A highlighted message is written to the console. It is usually useful to append the error object to the message so that the developer/administrator/user can determine what happened.