

[iWidget Primer v1.0]

Abstract:

To help understand the `iWidget` specification, this primer shows how `iWidgets` can be developed with increasing sophistication regarding how they connect to and leverage the framework within which they have been embedded. This version reflects v1.0 of the specification.

Table of Contents

[1 Introduction](#)

[1.1 What is an iWidget?](#)

[2 Simplest iWidgets](#)

[3 Encapsulation](#)

[4 Customization attributes](#)

[5 Coordination](#)

[5.1 Eventing](#)

[5.2 Shared state](#)

[5.2.1 ItemSets](#)

[6 Inclusion on a page](#)

[7 Modes](#)

[8 NavState](#)

[9 Rewriting URIs](#)

[Appendix A Best Practices](#)

1 Introduction

This document's intent is to demonstrate to the `iWidget` developer how the specification impacts the coding of `iWidgets` and the functionality provided as a result of those impacts. The approach is to examine each feature area separately, though `iWidgets` can certainly combine any set of these for its actual operation. Each section highlights the changes made to the evolving example in order to include the functionality discussed in that section.

1.1 What is an iWidget?

An `iWidget` is a browser-oriented component (regardless of whether or not the "browser" or "Web platform" is provided by a traditional browser or some application like [XULRunner](#)) designed to work within the framework defined by the [iWidget specification](#). Such a component will only occupy a portion of the overall working canvas and is usually designed in a manner that makes it easy for the canvas assembler to connect the `iWidget` to other `iWidgets` on the canvas.

2 Simplest iWidget(s)

The simplest `iWidgets` ignore the page framework and simply produce their visualization, if they have one, for the user. Examples of the obligatory HelloWorld `iWidget` for various cases includes:

XHTML-style

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:iw='http://www.ibm.com/xmlns/prod/iWidget'>
  <head>
    <meta name="title" content="Hello World">
  </head>
  <body class="iw-Content">
    Hello World
  </body>
</html>
```

Examining the example of the simplest XHTML `iWidget` leads one to understand that the only connections it has to `iWidgets` is the inclusion of `xmlns:iw='http://www.ibm.com/xmlns/prod/iWidget'` on the `<html>` element and the class `"iw-Content"` on the `<body>` element. This first defines the `iWidget` namespace and indicates to the including `iContext` that an `iWidget` is being defined. The XML style uses this namespace to declare `iWidget`-specific elements/attributes, but the XHTML style simply uses it to declare this is an `iWidget` definition. The second connection declares that the `<body>` element directly contains the content for this `iWidget`. Later examples will move this class to a child of the `<body>` element as

additional information will be declared using other sibling elements.

XML-style

```
<iw:iWidget id="helloWorld" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget" id="HelloWorld">
  <iw:content>
    Hello World
  </iw:content>
</iw:iWidget>
```

Just script

```
iContext.handledEvents([ {onLoad      /* event name */,
                          null        /* payload type */,
                          "Load Event" /* event description */,
                          null        /* aliases */,
                          {iContext.getRootElement().innerHTML=Hello world},
                          null        /* extra handler attributes */} ])
```

Simple `iWidgets` can use script for improving user interactivity or validating input, but will need either to avoid defining script variables and functions or take care of encapsulation issues themselves, as they are ignoring the framework's support, for example:

Simple event handler

```
<span class="iw-iWidget" id="SimpleEventHandler">
  <span class="iw-content">
    <input type="text" onblur="if (null==this.value || 0==this.value.length)
                                alert('Error: value required!');"> </input>
  </span>
</span>
```

3 Encapsulation

The `iWidget` author is responsible for providing proper encapsulation support, but the `iContext` definition provides needed assistance in this task. We will use "`iContext`" to refer to the framework for executing an `iWidget` (including managing its lifecycle and connecting it to other `iWidgets`) and the services an `iWidget` can invoke on that framework. The following are recommended as best practices for providing encapsulation support within `iWidgets`. It is critical these best practice guidelines be followed so that things operate properly even when the user places two instances of an `iWidget` within a single DOM:

1. Make all script variables instance-oriented. The key supporting functionality for this is the method `iContext.iScope()`. The `iScope` method provides a means for reliably accessing an Object instance that `iContext` initialized for the purpose of supporting the `iWidget` author's need to encapsulate their script content. This Object instance can be accessed inside methods, event handlers, and more using:

```
var root = iContext.iScope();
```

The `iWidget` author can specify an Object to use when creating this instance by specifying the Object's name using the "iScope" attribute on the root element of the `iWidget`'s definition.

2. Only directly reference the `iContext` functionality from the encapsulating object instance (using "this.iContext.___" or anonymous functions (e.g. `onclick="iContext.iScope().__"`). All other references are unlikely to resolve into the wrapper the `iContext` uses to scope its functionality to the instance of the `iWidget`.
3. Consider all script methods to be shared resources. This allows the `iContext` to manage the loading of scripts such that each script is loaded only once by the underlying platform. Since "iContext." will not be resolved into the proper instance reference for such scripts, methods needing the functionality `iContext` provides need to accept it as a parameter (for example function `foo(iContext, ...)`) It is expected many `iWidgets` will have a single core object definition that can be loaded as the encapsulating object instance referred to in #1, as in:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget">
  <head>
    <meta name="iScope" content="myStuff">
    <script src="http://example.com/myWidget/js/myStuff_v10.js" />
```

In this example, "http://example.com/myWidget/js/myStuff_v10.js" is the script to be loaded. It can potentially define multiple objects and can be only one of several such inclusions. The example uses the "iScope" attribute to cause the `iContext` to create an instance of the "myStuff" object for use as the encapsulating object instance.

4. Always use the `iContext.getElementById()` method rather than the document-level equivalent as this will scope the request to the content of the `iWidget`'s.

The following example seeks to demonstrate these concepts:

XHTML-style

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget" >
4   <head>
5     <meta name="iScope" content="myStuff">
6     <meta name="title" content="Encapsulation example" />
7     <script type="text/javascript" src="http://example.com/myWidget/js/myStuff_v10.js" />
8   </head>
9   <body onLoad="onLoadHandler">
10    <span class="iw-Content" id="HiThere">
```

```
11     Hello World (Encapsulated)
12     </span>
13 </body>
14 </html>
```

where myStuff_v10.js contains:

```
15 function myStuff()
16 {
17     function onLoadHandler(ievent)
18     {
19         this.iContext.getElementById("HiThere").innerHTML = "Hi There";
20         /* ... Note: the use of the ievent parameter is optional ... */
21     }
22 }
```

Where line 5 defines the object that is used as the encapsulating object instance for the `iWidget`. Line 7 loads the file that defines the referenced encapsulating object, and line 9 references an event handler for the `onLoad` event. The `iContext` loader first attempts to resolve defined event handlers against the encapsulating object instance, then against the global scripting scope, and finally as an inline handler (that is, an anonymous function). In this example, it resolves to the method definition found in line 17. The `onLoad` event handler (lines 17-21) leverages the reference to the `iContext` instance that provides functionality scoped to the `iWidget` in order to access a element within the `iWidget`'s markup in line 19.

4 Customization attributes

Most `iWidgets` are likely to customize some portion of the view their content provides to the user. Examples include a weather `iWidget` remembering a US user's preferred zip code or a stock `iWidget` remembering the set of symbols the user desires to see quotes for. The `iWidget` specification supports remembering such instance-level customizations as `iWidget` attributes. The `iWidget` attributes are stored as members of an `ItemSet`, which can be as simple as a collection of name/value pairs. The values for this predefined `ItemSet` (name="attributes") can either be specified in the manner shown in [Section 5.2](#) for shared `ItemSets` or using the `<meta>` element as shown below. The restriction on use of the `meta` tag is simply that the attribute names cannot overlap with those in [Section 5.1 of the specification](#), namely;

- title
- name
- description
- defaultHeight
- defaultWidth
- locale
- mode
- size

- author
- email
- website
- version
- globalAttributes

To illustrate, our example defines an attribute named "stock" and therefore becomes:

XHTML-style

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget">
4   <head>
5     <meta name="iScope" content="myStuff">
6     <meta name="title" content="Customization example" />
7     <meta name="stock" content="IBM GE" />
8     <script type="text/javascript" src="http://example.com/myWidget/js/myStuff_v10.js" />
9   </head>
10  <body onLoad="onLoadHandler">
11    <span class="iw-Content" id="HiThere">
12      Hello World (Encapsulated & Customized)
13    </span>
14  </body>
15 </html>
```

where myStuff_v10.js contains:

```
16 function myStuff()
17 {
18   function onLoadHandler(ievent)
19   { var s = "Current customization attributes:\n";
20     var attrs = this.iContext.getiWidgetAttributes();
21     var names = attrs.getAllNames();
22     for (int i=0; i<names.len; i++)
23     { s += names[i] + ": " + attrs.getItemValue(names[i]) + "\n"; }
24     this.iContext.getElementById("HiThere").innerHTML = s;
25     /* ... Note: the use of the ievent parameter is optional ... */
26   }
27 }
```

Where lines 7 defines a customization attribute for this `iWidget` and lines 19-23 access the set of customization attributes for the purpose of building a display string.

5 Coordination

When a user interface is composed of components from disparate sources, as is expected to be normal with `iWidgets`, it is desirable for the `iWidgets` to be coordinated with activity outside of themselves (for example selecting a new location on a map causes other location-sensitive `iWidgets` to also reflect the new location). The `iWidget` specification defines two mediated mechanisms by which this coordination can be effected. While we will discuss the specifics of the two separately, there are some common themes worth mentioning first:

- Coordination is mediated by the `iContext` implementation: This allows the `iContext` implementation to determine a variety of factors, such as policy/security constraints on the distribution of the coordination information and transforms that should be applied.
- Descriptive metadata: Coordination information is described in metadata such that the effective distribution method can be determined by the `iContext` implementation rather than the `iWidget` specification. Possible models include pub/sub, explicit "page" wiring, user defined wiring, and more.

The `iWidget` specification defines the coordination mechanisms described in the next two sections, namely eventing and shared state.

5.1 Eventing

The `iWidget` specification defines an eventing infrastructure that provides a well known means for the `iWidget` to publish and receive events while providing flexibility relative to how the `iContext` implementation chooses to distribute those events. In order to accomplish this decoupling, it is preferred to have the `iWidget` definition explicitly declare those events that it can either source (or publish) as well as those it can consume (or process), including which handler should be invoked to do the processing. The red/bold lines in the example below add such definitions (lines 11-17) and an event handler (lines 34-41).

XHTML-style

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget">
4      <head>
5          <meta name="iScope" content="myStuff">
6          <meta name="title" content="Coordination example" />
7          <meta name="stock" content="IBM GE" />
8          <script type="text/javascript" src="http://example.com/myWidget/js/myStuff_v10.js" />
9      </head>
10     <body onLoad="onLoadHandler">
11         <span class="iw-EventDescription" id="stockListChanged">
12             <span class="iw-Type">string</span>
13         </span>
14         <span class="iw-PublishedEvents iw-HandledEvent">
15             <a class="iw-Event" name="#stockListChanged" />

```

```

16     <span class="iw-Handler">onStockListChanged</span>
17     </span>
18     <span class="iw-Content" id="HiThere">
19         Hello World (Encapsulated, Customized & Eventing)
20     </span>
21 </body>
22 </html>

```

where myStuff_v10.js contains:

```

23 function myStuff()
24 {
25     function onLoadHandler(ievent)
26     { var s = "Current customization attributes:\n";
27       var attrs = this.iContext.getiWidgetAttributes();
28       var names = attrs.getAllNames();
29       for (int i=0; i<names.len; i++)
30       { s += names[i] + ": " + attrs.getItemValue(names[i]) + "\n"; }
31       this.iContext.getElementById("HiThere").innerHTML = s;
32       /* ... Note: the use of the ievent parameter is optional ... */
33     }
34     function onStockListChanged(ev)
35     {
36         var stocks = ev.payload;
37         var newList = null;
38         for (int i=0; i<stocks.len; i++)
39         { newList += stocks[i] + " ";}
40         this.iContext.getiWidgetAttributes().setItemValue("stock", newList);
41     }
42 }

```

5.2 Shared state

The other technique by which `iWidgets` can be coordinated with each other leverages the framework's support for shared state. This support has `iWidgets` declare named sets of shared state that the framework is then free to coordinate in any manner it chooses. Examples of such coordination includes providing multiple `iWidgets` access to the same underlying state object and providing a means by which the shared state is associated directly by the `iWidget`, but coordinated with other `iWidgets` either via updates to their independent shared state objects or via events. Since other coordination techniques can be provided by the framework, this specification does not define the coordination model, leaving it as an exercise for the framework developer(s). As for `iWidget` authors, they simply declare their shared state and ignore the issues related to effecting the underlying coordination.

The shared state model leverages the `ItemSet` concept (discussed in the next section) to provide a simple abstraction and interface to an underlying data store. In most cases, the `iWidget` author does not need to know the nature of the implementation lying behind the `ItemSet` interface and can treat it simply as a mechanism for grouping related data items that are individually accessed by name. The following adds the declaration for one `ItemSet` to our example.

XHTML-style

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget">
4   <head>
5     <meta name="iScope" content="myStuff">
6     <meta name="title" content="Coordination example" />
7     <meta name="stock" content="IBM GE" />
8     <script type="text/javascript" src="http://example.com/myWidget/js/myStuff_v10.js" />
9   </head>
10  <body onLoad="onLoadHandler">
11    <span class="iw-EventDescription" id="stockListChanged">
12      <span class="iw-Type">string</span>
13    </span>
14    <span class="iw-PublishedEvents iw-HandledEvent">
15      <a class="iw-Event name="#stockListChanged" />
16      <span class="iw-Handler">onStockListChanged</span>
17    </span>
18    <span class="iw-ItemDescription" id="stockDef">
19      <span class="iw-Type">string</span>
20    </span>
21    <span class="iw-ItemDescription iw-Repeats" id="pricesDef">
22      <span class="iw-Type">{price:number, dateTime:xsd:dateTime}</span>
23      <span class="iw-Description">An array of prices from various points in time</span>
24    </span>
25    <span class="iw-ItemSetDescription" id="stockDataDef">
26      <a class="iw-Item" id="stock" href="#stockDef"/>
27      <a class="iw-Item" id="prices" href="#pricesDef"/>
28    </span>
29    <span class="iw-ItemSet" id="stockData">
30      <a class="iw-ItemSetDescRef" href="#stockDataDef"/>
31    </span>
32    <span class="iw-Content" id="HiThere">
33      Hello World (Encapsulated, Customized, Eventing & SharedState)
34    </span>
35  </body>
36 </html>

```

where myStuff_v10.js contains:

```

37 function myStuff()
38 {
39   function onLoadHandler(ievent)
40   { var s = "Current customization attributes:\n";
41     var attrs = this.iContext.getiWidgetAttributes();
42     var names = attrs.getAllNames();
43     for (int i=0; i<names.len; i++)
44       { s += names[i] + ": " + attrs.getItemValue(names[i]) + "\n"; }

```

```

45     this.iContext.getElementById("HiThere").innerHTML = s;
46     /* ... Note: the use of the ievent parameter is optional ... */
47     }
48     function onStockListChanged(ev)
49     {
50         var stocks = ev.payload;
51         var newList = null;
52         for (int i=0; i<stocks.len; i++)
53             { newList += stocks[i] + " ";}
54         this.iContext.getiWidgetAttributes().setItemValue("stock", newList);
55     }
56 }

```

Note that Items are described separately (lines 18-24) from the sets that contain them so that they can be reused across sets. The ItemSet description (lines 25-28) defines the Items contained within this particular set and is separate from the declaration of an instance of the set (lines 29-31) so that set definitions can also be reused.

5.2.1 ItemSets

The ItemSet interface provides an abstract interface to an underlying data model. It should be straightforward to support this abstract interface for a variety of data types (for example JSON, XML, XForms, and more), potentially providing additional, and likely more advanced, functionality specific to the underlying data model. In all cases, the core semantics related to getting or setting the value associated with a particular name can be done through the ItemSet interface.

6 Inclusion on a page

Care has been taken to support simple means of including `iWidgets` into pages. Whether or not this is used by tooling authors is an implementation choice, but it does allow a page author to create a normal Web page that incorporates `iWidgets` and provide an `iContext` implementation (for example refer to an open source implementation) and have things run normally.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget">
4      <head>
5          ... page oriented stuff ...
6          <script type="text/javascript" src="http://example.com/iContextImpl.js" />
7      </head>
8      <body>
9          ... Some inline content ...
10         <span class="iw-iWidget" id="an_iWidget"
11             <a class="iw-Definition" href="http://example.com/iwidgets/someIWidget.xhtml" />
12         </span>

```

```

13
14     ... More inline content ...
15
16     <-- Load a definition for next iWidget -->
17     <span class="iw-iWidget" id="other_iWidget"
18         <a class="iw-Definition" href="http://example.com/iwidgets/anotherIWidget.xhtml" />
19     </span>
20
21     ... More inline content ...
22
23     <-- Place 2nd copy of an iWidget -->
24     <span class="iw-iWidget" id="other_iWidget"
25         <a class="iw-Definition" href="http://example.com/iwidgets/anotherIWidget.xhtml" />
26     </span>
27 </body>
28 </html>

```

7 Modes

Modes provide a convenient means for `iWidgets` to define different markup for different purposes. Examples include those which the `iContext` will also understand (for example "view", "edit," and "help" modes) and those that only the `iWidget` understands (for example a custom mode for "shoppingCart"). The `iContext` interprets such custom modes as if they are the "view" mode with the exception that any content section marked as being for the custom mode will be shown to the user, for example;

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget">
5     <head>
6         /* ... */
7     </head>
8     <body>
9         /* ... */
10     <span class="iw-Content view">
11         Sample "view" mode content
12     </span>
13     <span class="iw-Content edit">
14         Content for editing the iWidget's customization attributes
15     </span>
16     <span class="iw-Content shoppingCart">
17         Content for the custom "shoppingCart" mode
18     </span>

```

```
19 </body>
20 </html>
```

The current mode for the `iWidget` can be retrieved from the `iDescriptor` set using a construct such as:

```
var mode = iContext.getiDescriptor().getItemValue("mode");
```

When the `iWidget` wishes to change the mode, it fires an event for the `iContext` to receive. In a symmetric way, the `iContext` will fire the same event when it originates a mode change. This event can be fired using a construct such as:

```
iContext.iEvents.fireEvent("onModeChanged", "", {"newMode": iContext.constants.mode.EDIT});
```

When the `iContext` receives such an event and is able to locate content marked as being for the new mode, it will hide the currently visible content and show the identified content. As a result, for v1.0, the `iContext` is only required to support one mode being active at any point in time. Should the `iWidget` desire to make more than one mode visible at a time (for example; both "edit" and "help"), it has the full responsibility for managing the show and hide of the secondary content (for example the "help" content). In addition, the `iWidget` needs to be aware that the scoped operations (for example `getElementById`) of the `iContext` still apply to the content related to the mode of the primary view (for example the "edit" content).

8 NavState

Navigational state is a concept well known in the portlet programming model whereby the portlet (the component analogous to an `iWidget`) informs the portlet runtime (analogous to the `iContext`) about the transitory state needed to render the current view for the user. This allows the containing framework to implement various user experiences without the component caring about the different state handling required by the user experience. A particular use case this can enable is the user bookmarking the page and returning to the same rendered state of all of the components at some later point in time. This concept is mapped into the `iWidget` space by the predefined event "onNavStateChanged." This event allows cases where both the `iWidget` and the `iContext` implement the navigational state concept to provide user experiences such as loading a page with the `iWidget` already navigated to something other than its default markup (perhaps a detail view rather than a summary view or a map with a set of markers/locations already defined, and more). If either the `iWidget` or the `iContext` do not implement this concept, the user will always initially view the default rendering state of the `iWidget`.

The `iWidget` and `iContext` communicate changes in the `iWidget`'s navigational state using the predefined event "onNavStateChanged." This event uses a payload type of "String," and the means by which the actual state gets

placed into or read out of that string is the responsibility of the `iWidget`, though a JSON object would be an obvious choice. This string is an opaque object to the `iContext` as it only stores the string and supplies it back to the `iWidget` as appropriate, for example when initializing a page loaded from a bookmark that has stored such state. All of the details related to such storage is the responsibility of the `iContext` (for example if stored in the URL, appropriate encoding / decoding must be done by the `iContext`). When loading a page where there is navigational state defined for the `iWidget`, the `iContext` will fire the event providing the navigational state after it has fired the `onLoad` event as this allows the `iWidget` to properly initialize before having to process the supplied navigational state.

XHTML-style

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml"
4      xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget">
5      <head>
6          <meta name="iScope" content="navState">
7          <meta name="onNavStateChanged" content="onNavStateChanged">
8          <meta name="onModeChanged" content="onModeChanged">
9          <script type="text/javascript" src="http://example.com/myWidget/js/navState_v10.js">
10         <!-- reference some json support off apache.org -->
11         <script type="text/javascript"
12             src="http://svn.apache.org/viewvc/incubator/shindig/trunk/php/js/json.js?
revision=606463&pathrev=606463">
13         </script>
14         </head>
15         <body onLoad="onLoadHandler">
16             <span class="iw-Content view">
17                 Primary view
18                 <input type="button" value="Switch to secondary view"
19                     onclick="iContext.iScope().switchView('SecondaryView');" />
20             </span>
21             <span class="iw-Content SecondaryView">
22                 Secondary view
23                 <input type="button" value="Switch to primary view"
24                     onclick="iContext.iScope().switchView('view');" />
25             </span>
26         </body>
27     </html>

```

where `navState_v10.js` contains:

```

27  function navState()
28  {
29      var nState=new Object();
30
31      function onNavStateChanged(ev)
32      { nState = ev.payload.parseJSON();
33        if (nState.currentMode != this.iContext.getiDescriptor().getItemValue("mode"))
34          { this.iContext.iEvents.fireEvent("onModeChanged",null, "{newMode:'"+nState.currentMode+'"}"); }
35      }

```

```

36 function onModeChanged(ev)
37 { nState.currentMode=this.iContext.getiDescriptor().getItemValue("mode");
38   this.iContext.iEvents.fireEvent("onNavStateChanged",null, nState.toJSONString());
39 }
40 function switchView(newView)
41 { nState.currentMode=newView;
42   this.iContext.iEvents.fireEvent("onNavStateChanged",null, nState.toJSONString());
43   this.iContext.iEvents.fireEvent("onModeChanged",null, "{newMode:'"+newView+"'}");
44 }
45 }

```

This example loads methods supporting JSON from apache.org (lines 11-12), defines two views using customized modes (lines 15 and 20), and stores the current mode as a field in a JSON Object used to encapsulate its navigational state. Event handlers are registered (lines 7 and 8) for the relevant events. When these events are received (lines 31 and 36 respectively), the code keeps the Object storing the navigational state (lines 32, 37 and 41), the navigational state registered with the `iContext` (lines 38 and 42), and what is being displayed to the user (lines 34 and 43) in sync with each other.

9 Rewriting URIs

There are two significant reasons URIs often need to be rewritten in environments such as those supporting `iWidgets`:

1. Either the JavaScript sandbox or the network configuration require that the access be routed through the server that sourced the overall page. As the component cannot reasonably know how to provide such routing, assistance in rewriting its URIs is needed.
2. The URI references a relative resource, but both the deployment details on the server hosting the `iWidget` and how that server can be accessed from the user agent are unknown to the `iWidget`. Again, assistance to rewriting such URIs is needed.

The specification provides two means for requesting assistance in rewriting URIs:

1. An optional method is defined (`iContext.io.rewriteURI`) that accepts a URI as a parameter and returns a URI that can be used to access the underlying resource.
2. A class of "iw-RewriteURI" can be placed on any element and the `iContext` replaces the value of the URI-oriented attribute of that element (for example the "src" attribute on an `` tag or the "href" attribute of an `<a>` tag) with what would have been returned if the optional method `iContext.io.rewriteURI` had been called. If this method is not implemented, the value of the URI is left unchanged.

Appendix A: Best Practices

A.1: Type descriptions

Type descriptions are used in several places in the `iWidget` specification. In order to improve interoperability, the following provides an ordered list for the preferred means to declare a type:

1. **Simple types** ("String", "Number", and "Boolean"): These will be the most exchangeable types and therefore are encouraged whenever other considerations don't demand more involved types.
2. **Arrays** ("*some type*"): This indicates that the type will be repeated within an array structure. Examples where this is useful includes specifying points on a map or graph. The enclosing square brackets are the key that this style of definition is in use.
3. **Schema primitive datatype definitions prefixed with xsd** ("xsd:dateTime", "xsd:duration", etc.): The complete list of these types is in the XML Schema Datatype Datatype specification (<http://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes>). Note that the base primitive types (xsd:string and xsd:boolean) overlap the simple types from #1 and therefore should not be used.
4. **JSON-like description** ("{field1Name:field1Type, field2Name:field2Type}"): This defines an object that has the named fields (or properties for those preferring that terminology), each of which will be of the declared type (recursively using this best practice for the type definition). The enclosing curly braces are the key that this style of definition is in use.
5. **XML Schema defined datatypes**: In this case, the type provides a URI from which the definition can be loaded, including a reference to the particular item (for example "http://example.com/mySchema.xsd#myType")
6. **ItemSets**: When ItemSet is used as the type, the type should be described as "itemset:{setDescName}," where "{setDescName}" is replaced with the name in the ItemSet's description.