

# *Widget Programming Guide*

*Revision 1.0*

# *Table of Contents*

1	Introduction .....	3
1.1	Using this document .....	3
2	Creating your first widget.....	4
2.1	Defining a widget.....	4
3	Setting up an environment to run widgets .....	5
3.1	Hardware and software requirements .....	5
3.2	Understanding the development environment .....	5
3.3	Setting up Lotus Mashups .....	6
3.4	Deploying and running your “Hello World” widget .....	10
4	Widget basics.....	12
4.1	Defining a simple widget.....	12
4.2	Adding a widget to a Web page.....	13
4.3	Common widget practice.....	13
4.4	Widget life cycle .....	15
4.5	Widget attributes .....	15
5	Widget communication.....	17
5.1	Simple events.....	17
5.2	Widget event coordination.....	18
5.3	Payload and payloadType.....	20
6	Packaging and deploying widgets .....	21
6.1	Package types .....	21
6.2	Deploying widgets as WAR files .....	21
6.3	JAR widget deployment .....	22
7.1	Widget Factory .....	23
8	Widget NLS support.....	25
10	Feeds.....	26
10.1	Feed Readers .....	26
10.2	Feed Reader.....	26
10.3	Data Viewer.....	27
11	Proxy Server .....	29
12	Advanced Topics .....	30
12.1	Adding Back-end Java code to your widget .....	30
12.2	Enabling person tags.....	31
12.3	Events .....	31
12.4	Generic Google Gadget widget .....	33
12.5	Widget Authentication.....	35

---

# *1 Introduction*

Lotus Mashups is a graphical, browser-based system that supports easy, on-the-glass assembly of mashups. It includes the following components:

- A set of rich, out-of-the-box widgets
- A client-side widget rendering engine
- A lightweight Lotus Mashups server
- A catalog for discovering and sharing mashups

A mashup is typically characterized as a lightweight integration of applications via widgets. Widgets can be mashed and wired together in a browser. They use Web technologies such as HTTP, JSON, XML, JavaScript, Atom and RSS to deliver application content. A widget is a portable piece of code that can run in any Web application without requiring a separate compilation.

A number of frameworks exist today that can implement this paradigm, and many of them have overlapping concepts embedded within their component models. Due to the lack of open standards, these shared concepts are all surfaced in different manners, eliminating interoperability of the components between frameworks. Lotus Mashups has adopted the evolving iWidget specification. Lotus Mashups v1.0 is based on the 1.0 version of the specification. You can download a copy of this specification, along with a primer to get started at <http://www-10.lotus.com/ldd/mashupswiki.nsf/dx/widget-programming-guide>.

## *1.1 Using this document*

The primary focus of this document is to serve as a developer's guide for creating widgets using the Lotus Mashups framework, which is based on the iWidget specification v1.0. It covers several concepts, including the following:

- Context: Provides interfaces for overall management of widgets on a page
- Basic types: ItemSets, Events
- Basic widget components: Attributes and interaction with iContext
- Widget definition declarative syntax
- Widget instantiation syntax
- Widget packaging and deployment
- NLS support
- Widgets as OSGI bundles
- Widget as WAR files
- Lotus Mashups proxy
- Examples of some widgets

## 2 *Creating your first widget*

Two of the major design goals of the iWidget specification are simplicity and extensibility. These two design considerations allow developers to develop their Web 2.0 components their own way. As a developer, you may create very simple widgets or very complex widgets. In this chapter, we will look at how to create a very simple widget.

### 2.1 *Defining a widget*

Since there is really no interface to implement, creating a widget to display simple HTML markup is easy. The widget we will create in this section displays “Hello World” on the screen. You can do this by using your favorite text editor to create a simple XML file that contains the following strings:

```
<iw:iwidget name="HelloWorld"
  xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget"
  supportedModes="view" mode="view">
  <iw:content mode="view">
    <![CDATA[
      <h1>Hello World</h1>
    ]]>
  </iw:content>
</iw:iwidget>
```

Save this file on your local machine, and name it helloworld.xml.

NOTE: You may notice that `Hello World` is included in the CDATA tag. This is required to keep possible XML closing tags such as `>` from interfering with the greater or less than signs.

To run your new widget, you need to have an environment that provides the iWidget specification implementation runtime. Continue to the next chapter to learn how to set up the environment for running widgets.

## *3 Setting up an environment to run widgets*

To run a widget, you must deploy it into an environment that implements the IBM iWidget specification. In this section, you will learn how you can use the IBM Lotus Mashups environment as a model for setting up your own environment to run the widget you created in the previous section.

### *3.1 Hardware and software requirements*

IBM Lotus Mashups version 1.0 is currently supports Microsoft Windows environment. Later releases are planned to support other platforms.

The following content provides the software and hardware requirements to run Lotus Mashups on WebSphere Application Server (WAS).

- Hardware:
  - Processor: Intel P4 or AMD
  - Memory: 1GB or above
  - Hard disk space: At least 2GB of free space
- Software:
  - Java JDK 1.5
  - Microsoft Windows system
  - A browser such as Firefox 2/Firefox 3/IE 7/Safari 3

### *3.2 Understanding the development environment*

When developing a simple widget, you may only need to author a simple XML file, as you did earlier in this document. In this case, you do not need any tools other than a simple text editor. However, in more complex cases, you will need a more advanced environment. For example, some complex widgets may need server-side components such as e-mail widgets, map widgets, and more. In addition to a good code editor, you also need debugging tools on both the server and client side. Eclipse and IBM Rational Application Developer allow developers to author almost any type of applications. They also provide useful debugging tools for various programs. Although you may always choose your own favorite editors, we recommend Lotus Widget Factory, Eclipse IDE and IBM RAD for widget development. To debug your widgets in case you have errors, we have found the FireFox add-on firebug to be very useful.

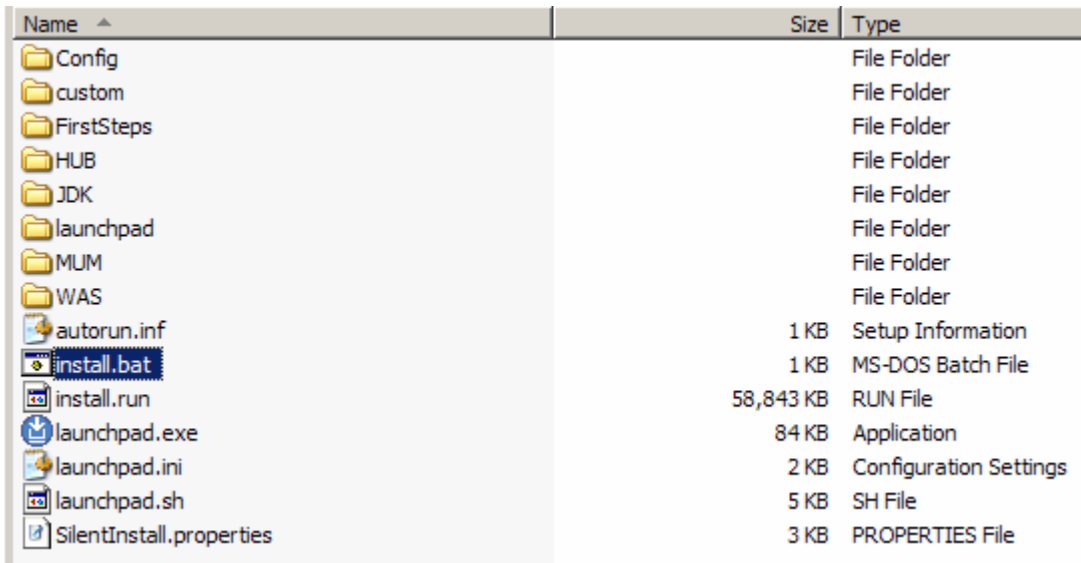
IBM is also in the process of adding new tools for creating widgets based on the iWidget specification. WebSphere sMash and Domino Designer have prototype extensions for widget development.

### 3.3 Setting up Lotus Mashups

Widget Factory and Rational Application Developer are ideal environments for developing widgets using the iWidget specification. At the time this document was created, these environments were not fully evolved for testing widgets. Therefore, we will use the Lotus Mashups installer to deploy and test widgets.

To install IBM Lotus Mashups, do these steps:

1. Download Lotus Mashups 1.0 installer from the IBM passport site or use the installer CD #1.
2. From the installer, click on install.bat to begin installation.

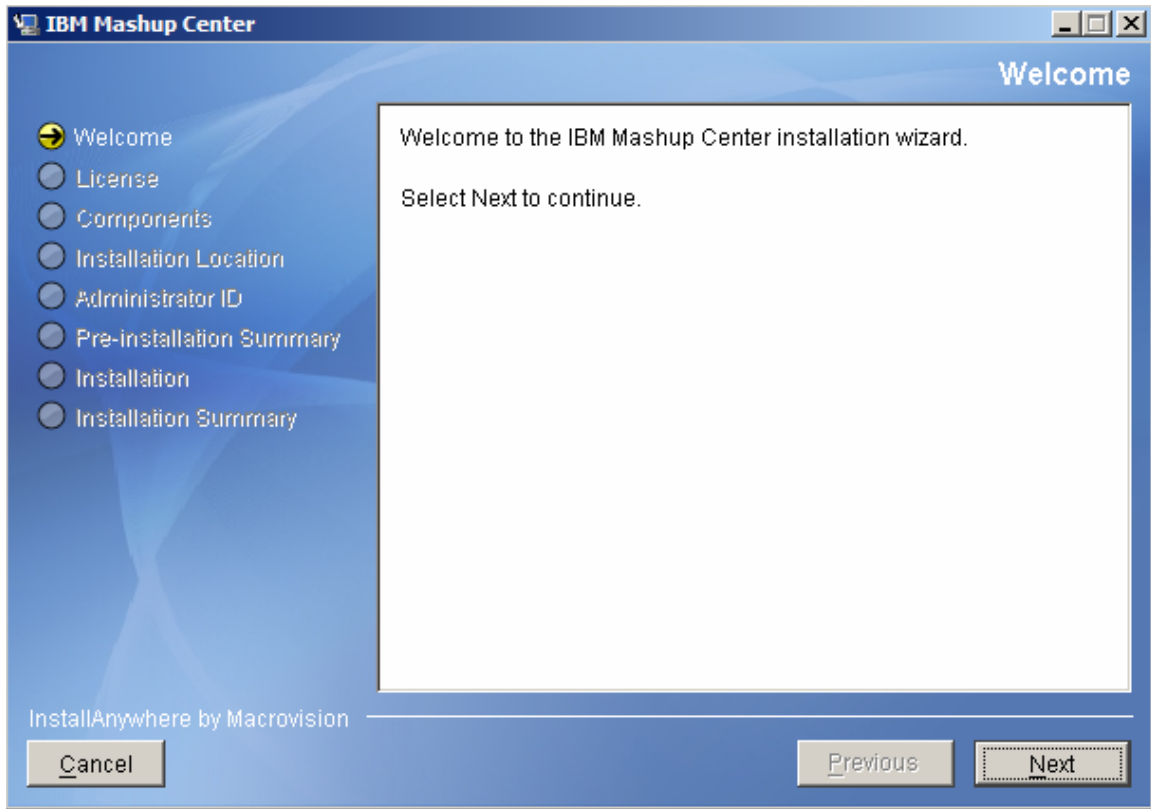


Name	Size	Type
Config		File Folder
custom		File Folder
FirstSteps		File Folder
HUB		File Folder
JDK		File Folder
launchpad		File Folder
MUM		File Folder
WAS		File Folder
autorun.inf	1 KB	Setup Information
<b>install.bat</b>	1 KB	MS-DOS Batch File
install.run	58,843 KB	RUN File
launchpad.exe	84 KB	Application
launchpad.ini	2 KB	Configuration Settings
launchpad.sh	5 KB	SH File
SilentInstall.properties	3 KB	PROPERTIES File

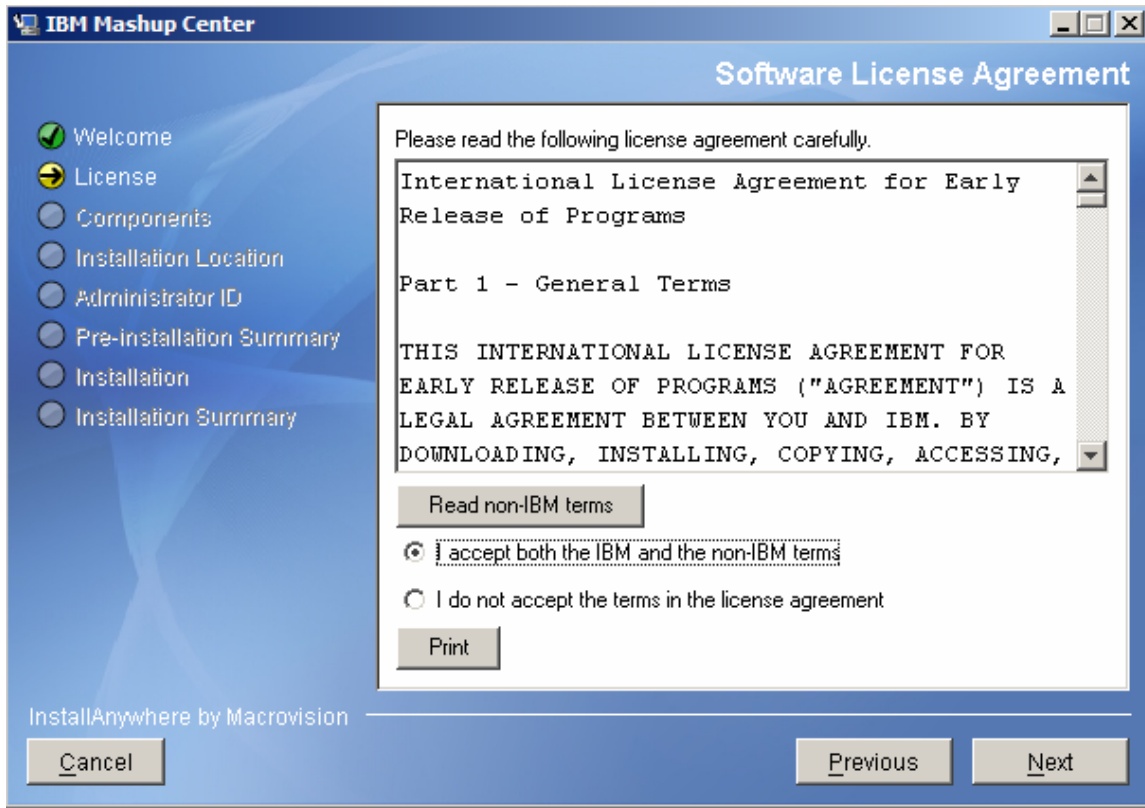
3. Lotus Mashups is packaged together with Infosphere Mashup Hub. Together, they provide a complete mashup-building solution called IBM Mashup Center. Later in the installation process, you will have an option to install Lotus Mashups without the MashupHub component and the vice versa. The first screen you should see in the installation process is the language selector screen. Select your language of choice, and click **OK**.



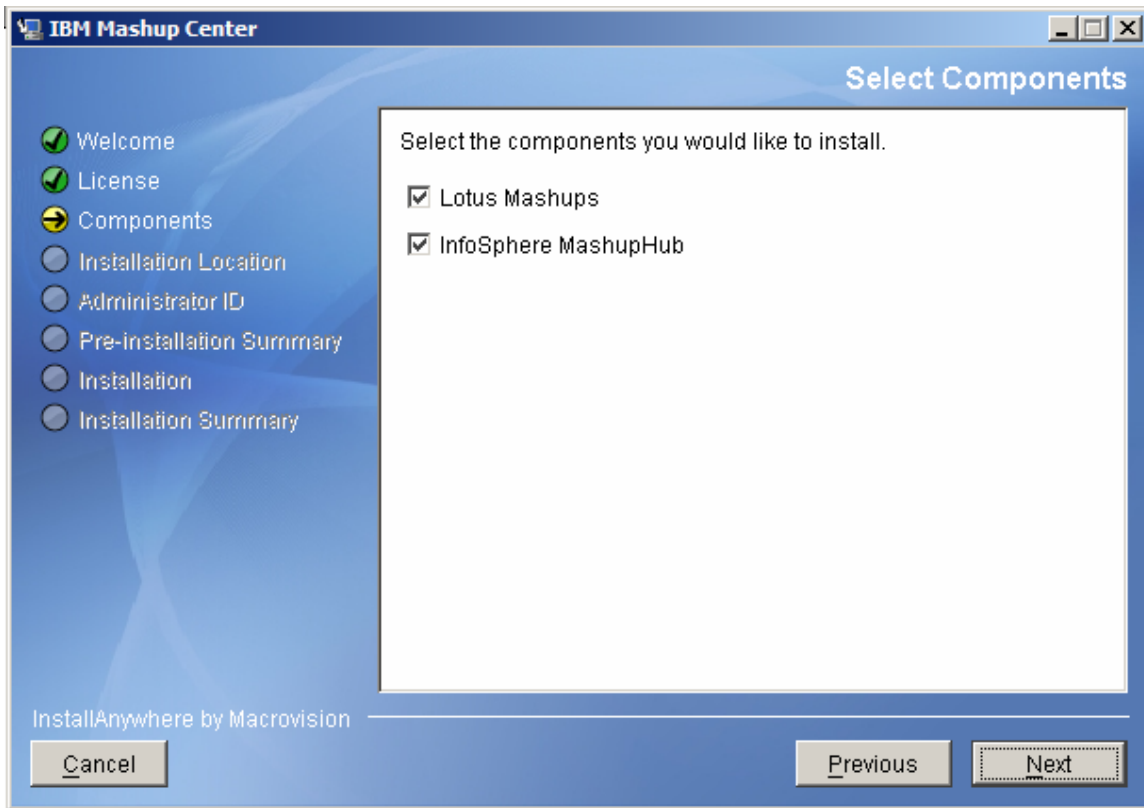
4. On the next window, click **Next** to continue.



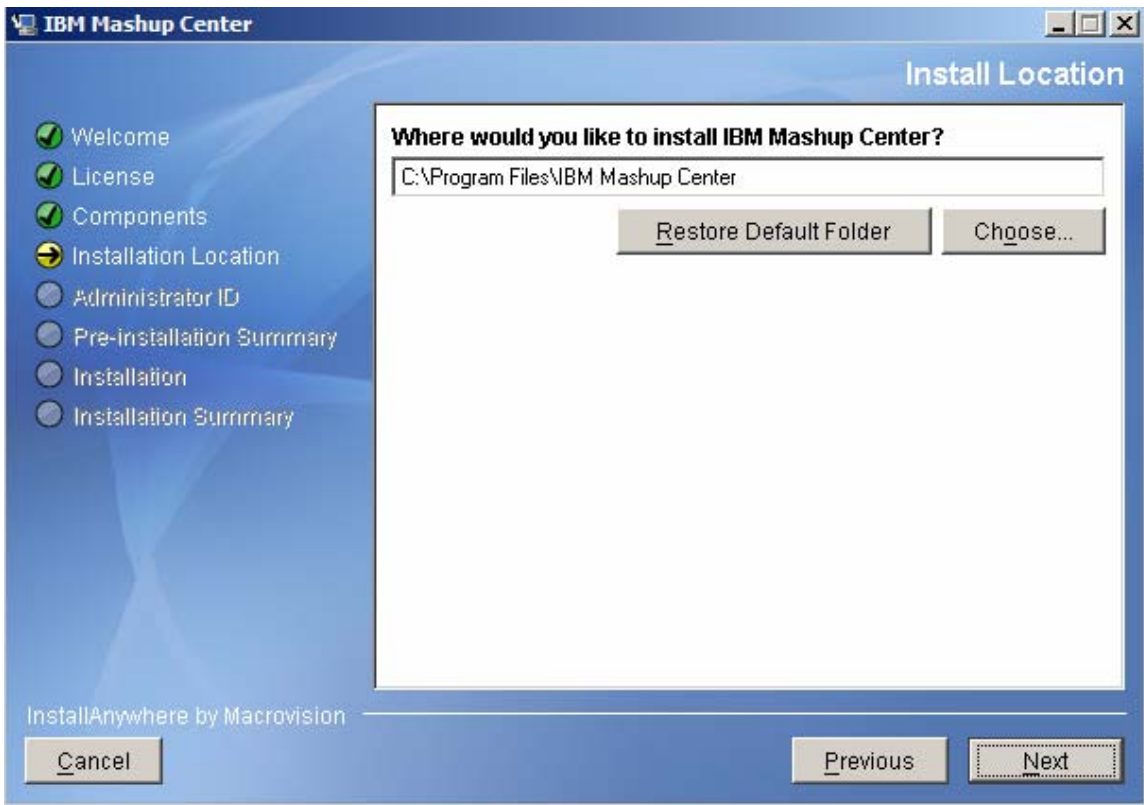
5. Accept the license agreement, and click **Next** on the next window.



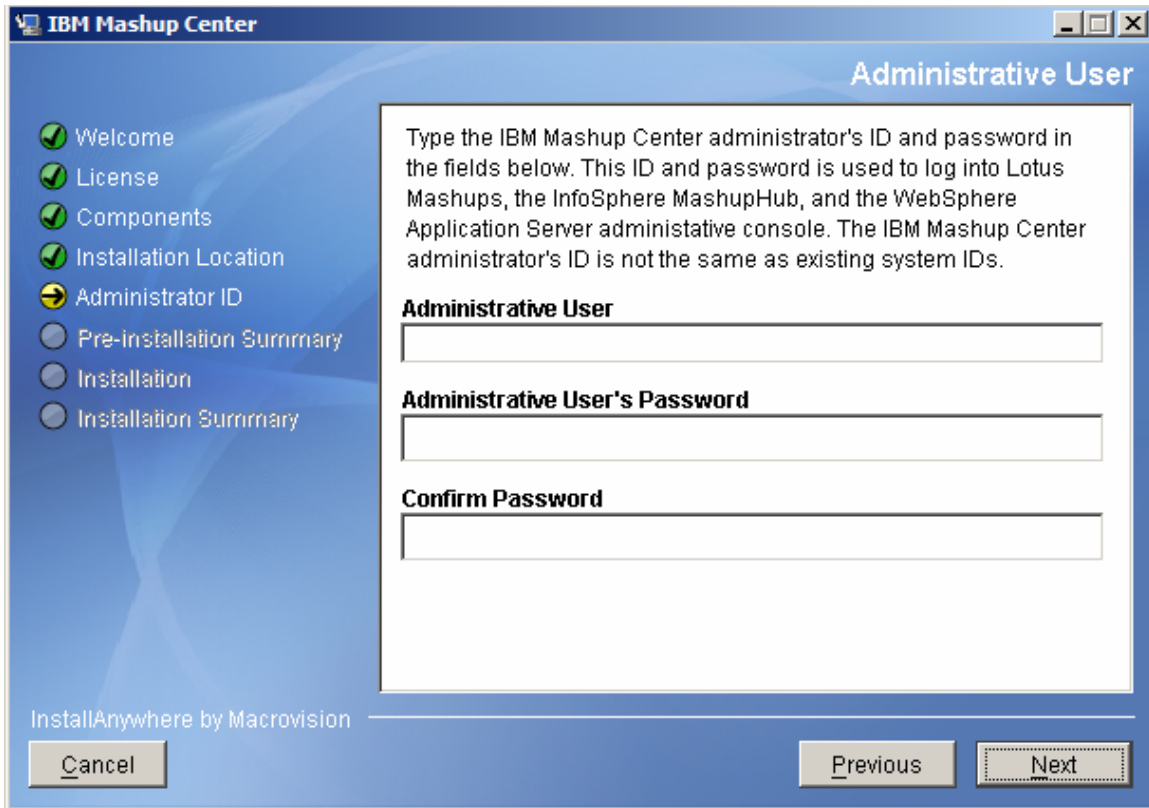
6. On the next window, you have a choice to install either MashupHub or Lotus Mashups or both. In production environments, you will most likely want to install MashupHub and Lotus Mashups on two different servers.



7. Specify the installation directory, and select **Choose**.



8. On the next window, type the administrator's ID and password that will be used for Lotus Mashups, MashupHub and the WebSphere Application Server administrative console.

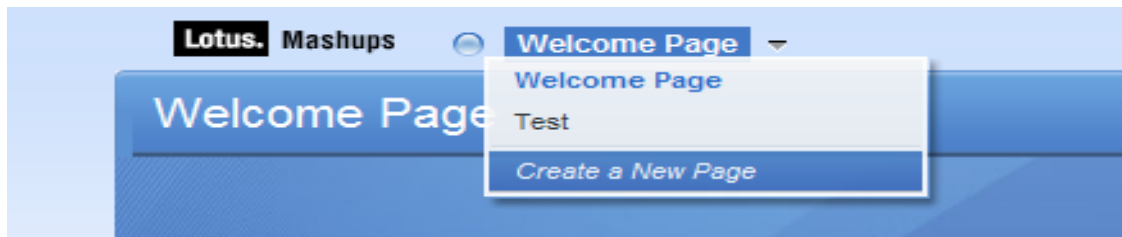


9. After the installation completes, go to <http://<host>:<default port>/mashups/enabler> to access Lotus Mashups.

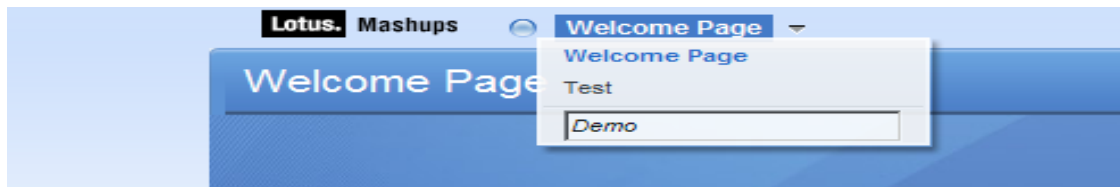
### 3.4 Deploying and running your “Hello World” widget

To get started, make sure the helloworld.xml file you created in section 2.1 is URL addressable. You can add this definition to any page created in Lotus Mashups. The following steps explain how to create a page and add a widget definition to the page source.

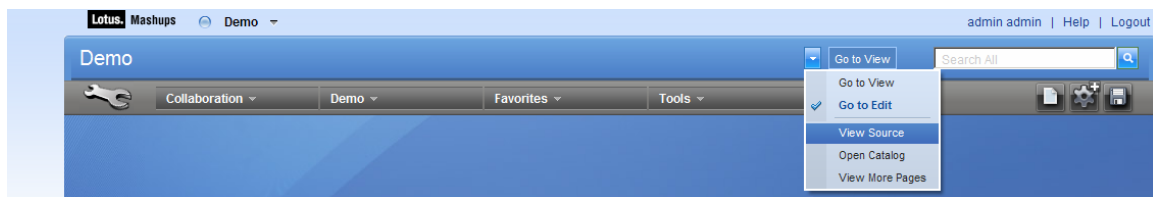
- (1) In the page navigation menu, select **Create a New Page**.



(2) Type a page name, for example **Demo**, and press the **Enter** key on your keyboard.



(3) After you create the page, you can change the page source by clicking **View Source** in the actions menu or by manually editing index.html in your file system, located at <LMinstall-root>/mm/public/nm/<usr>/<page-name>/index.html.



Here is the source code to add the Hello World widget to your page:

```
<div style="height: 100%;" widgettitle="FreeForm Layout" id="ns_68c37cd035a911ddb644f190425a465d" class="mm_iWidget">
  <a href="/mum/widget-catalog/freeFormLayout.xml" class="mm_Definition"></a>
  <span class="widgetcontainer">
    <span widgetstatus="loaded" locationstyle="position:absolute;left:38px;top:44px;width:400px;height:426;">
      <a href="/mum/widget-catalog/feedReader.xml" class="mm_Definition"></a>
      <span title="attributes" class="mm_ItemSet">
        </span>
      </span>
      <span widgetstatus="loaded" id="helloworld" class="mm_iWidget">
        <a href="http://localhost:8080/mm.liveobject/helloworld.xml" class="mm_Definition"></a>
      </span>
    </span></div>
<span id="hiddenWidgetsCorral" class="mm_iWidget">
  <a class="mm_Definition" href="/mum/widget-catalog/hiddenWidgetsCorral.xml"></a>
  <span class="hwc_container">
    </span>
</span>
</span>
```

You can package widgets as WAR files or OSGI plug-ins and then deploy them to Lotus Mashups. Normal recommended practice for deploying and adding a widget to your toolbox is to use the Hub interface. See *Packaging and deploying widgets* (section 6) for more information.

## 4 Widget basics

Widgets are browser-based components. They often extend a server-side component or are associated with a server-side data source. For example, you can associate the **Data Viewer** widget with a back-end data source to allow users to explore the data contained within the data source.

### 4.1 Defining a simple widget

A widget definition specifies what is needed to initialize the widget properly. You can create the widget definition either in XML or XHTML style. The following code defines a widget that displays a stock quote. It contains a definition file and a JavaScript file.

Here is the XML-style definition file:

```
<iw:iwidget id="stock" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget"
  iScope="stock" supportedModes="view" mode="view" lang="en">
  <iw:itemSet id="attributes" private="true" onItemSetChanged="changeItemSet">
    <iw:item id="broker" readOnly="false" value="Etrade" />
    <iw:item id="company" value="IBM" />
    <iw:item id="stock" value="105"/>
  </iw:itemSet>
  <iw:resource uri="stock.js" />
  <iw:content mode="view">
    <![CDATA[
      <div><span id="company" class="companyLabel">loading...</span> Stock Quote: <span
id="stock">loading...</span></div>
      <div>Broker: <span id="broker">loading...</span></div>
      <div><input type="button" style="height=10px" name="send" value="Send Data"
onclick="iContext.iScope().sendData()"/></div>
      <div><input type="button" style="height=10px" name="configure" value="Edit Broker"
onclick="iContext.iScope().editBroker()"/></div>
    ]>
  </iw:content>
</iw:iwidget>
```

See the following notes:

- The namespace in the first line indicates this is a widget based on the IBM iWidget specification.
- The `supportedModes` string defines the modes that are supported by the widget. In this example, it supports `view` mode only.
- The `mode` string defines the default mode that is displayed when the widget is first loaded on the page.
- The `iScope` string provides the name of an object used to instantiate an encapsulation object instance for the widget. You will learn more about this in section 4.5.
- The `resource` tag specifies a shared resource. In this example, the resource is a JavaScript (JS) file. This file is shared across different widgets.

- The `itemSet` tag sets a default value for the widget attribute `company`. The name attribute in the `itemSet` tag indicates that any items defined within this tag are widget attributes. You will learn more about this in section 4.4.
- The `content` tag indicates that the content in the `<![CDATA[... ]]>` is wrapped up into fragments and also that it is valid XML syntax.

## 4.2 Adding a widget to a Web page

You can embed an instance of a widget into a Web page using *microformat* style. The following example adds a simple `stock.xml` widget to a Web page. The widget attributes that are defined in the widget definition can be overwritten by values defined in the widget instance. A unique ID for each widget instance on the page is required so that each widget instance has a unique reference point. You specify the ID as the value of the `id` attribute on the `mm_iWidget` element.

```
<span class="mm_iWidget" id="2" >
  <a class="mm_Definition" href="/iwidgets/stock/stock.xml" ></a>
  <span class="mm_ItemSet" title="attributes">
    <a class="mm_Item" href="#company" style="visibility:hidden">YAHOO</a>
    <a class="mm_Item" href="#stock" style="visibility:hidden">27</a>
    <a class="mm_Item" href="#broker" style="visibility:hidden" >Ameritrade</a>
  </span>
  <span class="mm_ReceivedEvent" >
    <a class="mm_SourceEvent" href="#1" style="visibility:hidden" >sendStockData</a>
    <span class="mm_TargetEvent" style="visibility:hidden" >onGetCompanyName</span>
    <!-- one per overridden handler attribute -->
  </span>
</span>
```

## 4.3 Common widget practice

You can add multiple widgets to a single Web page, and you can place multiple instances of a single widget in a single DOM. In order to do this, you must properly encapsulate each widget instance. Here are some guidelines to follow as you do this:

1. Use the `iContext` object to interact with the framework and other page components. An `iContext` object is provided with the widget framework. It provides the following set of services so that a widget can interact with the framework and other page components:
  - a. `getiWidgetAttributes()`; returns `ManagedItemSet` to provide access to the widget's customization attributes.
  - b. `getItemSet(itemSetName)`; returns an `ItemSet` corresponding to the requested name or creates a new one. Lotus Mashups forces the policy that all the `ItemSet` values are private `ItemSet`.
  - c. `iScope()`; returns the instance object that encapsulates all the widget script variables. You will learn more about this in section 2.
  - d. `getElementById()`; returns a dom element of the widget.
  - e. `getRootElement()`; returns the `root` element of the widget.
  - f. `getElementsByClass()`; returns an array of elements in the widget markup that have the supplied value as one of those specified by the element's `class` attribute.
  - g. `processMarkup()`; processes the markup so that it can be inserted into the widget's markup.
  - h. `processiWidgets()` requests that the `iContext` object process the subtree under the supplied node in order to resolve and instantiate any referenced widgets.
  - i. `getUserProfile()`; returns `ManagedItemSet` to provides access to the user's profile data.

- j. `getDescriptor()`; returns `ManagedItemSet` to provide access to attributes that both the `iContext` object and the widget require.
  - k. `iEvents` file contains an object that provides access to an event service. For example, `fireEvent` can be used to publish the following event:
 

```

iEvents.fireEvent(/*String*/eventName,payloadType,pay
load)

```
2. All script resources can be shared across different widget instances. For example, if a stock widget has a simple JS file called `stock.js`, and you need to put two stock widgets on a single page, `stock.js` is loaded only once and is shared between both instances.
  3. `iWidget` encapsulation class: All the script variables and functions should be encapsulated in a single script class. This ensures that a unique instance object can be generated by the framework for each widget instance on the page. An `iContext` object reference is set in this encapsulation instance object by the framework so it can be available to the widget. Do the following steps.
    - a. Include an encapsulation class in a JS file and define the URI to download the JS file by using the resource tag `<iw:resource uri="stock.js" />`. Because the framework supports a relative path, it will try to get the resource from the same path as the widget definition itself.
    - b. Define an `iScope` attribute in `iw:iwidget` tag in the widget definition:

```

<iw:iwidget name="stock"
xmlns:iw=" http://www.ibm.com/xmlns/prod/iWidget "
iScope="stock">

```

This attribute tells the framework to instantiate a new instance of `stock` class after loading `stock.js`.

- c. The following example shows how a `stock` class may look when using the DOJO framework:

```

dojo.declare("stock",null,,
{
  onLoad: function () {
  },
  onView:function()
  {
    var element = this.iContext.getElementById("stock");
    var attributesItemSet = this.iContext.getiWidgetAttributes();
    element.innerHTML = attributesItemSet.getItemValue("stock");
    var elements = this.iContext.getElementsByClassName("companyLabel");
    var element2 = elements[0];
    element2.innerHTML = attributesItemSet.getItemValue("company");
    var element3 =this.iContext.getElementById("broker");
    element3.innerHTML = attributesItemSet.getItemValue("broker");
  },
});

```



- d. All references to the `iContext` object are made by using `this.iContext`, as shown in the above example.
4. Always use `iContext.getElementById` or `getElementsByClass` to put a widget element in the dom tree.
5. `iContext.io` is a great way to get the base widget directory. By calling `iContext.io.rewriteURI()` method, a widget developer can call resources inside the widget package without hard coding a context root.

## 4.4 Widget life cycle

Lotus Mashups widgets support two events -- `onLoad` and `onUnLoad`.

### OnLoad

Each widget should implement an `onLoad` event handler in the encapsulation class. The `onLoad` event is invoked by the framework when the widget is fully loaded. For example, when a new widget is added to a page, the widget framework loads the widget definition and updates the page dom with the markup that is defined in the `iw-content` element of the widget definition. Then the framework loads all the shared resources that are defined in the widget definition. Next, the widget framework sources the `onLoad` event that is defined by the widget.

### OnUnLoad

The `OnUnLoad` event is distributed by the framework when a widget is unloaded from the page. The event can be used to release any resource that is managed by the widget.

## 4.5 Widget attributes

Widget attributes are typically used to customize the look and feel of a widget or configure a widget for a specific business purpose. For example, in a stock widget, a company name can be used as a widget attribute to allow the stock widget to display a stock quote for that company.

You can define widget attributes in the widget definition. For example, in the following XML syntax, you can define `bgColor` as an attribute of the `<iw:iwidget>` element:

```
<iw:iwidget name="stock"
xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget" iScope="stock"
bgColor="value">
```

You can define an attribute as an item within `itemset-attributes`, as shown here:

```
<iw:itemSet name="attributes" >
  <iw:item name="company" value="IBM" readOnly="false"/>
</iw:itemSet>
```

You can customize widget attributes in a widget instance on the page. If the same attribute is already defined in the widget definition, the value gets overwritten by the value defined in instance.

For example, the following code defines two attributes in the widget instance: `company=YAHOO` and `stock=27`:

```
<span class="mm_ItemSet" title="attributes">
  <a class="mm_Item" href="#company"
style="visibility:hidden">YAHOO</a>
  <a class="mm_Item" href="#stock"
style="visibility:hidden">27</a>
</span>
```

Finally, a widget can use `iContext.getiWidgetAttributes()` to get the value of an attribute.

Using the above example, in view mode,

`[Context.getiWidgetAttributes().getItemValue("company")]` should return `YAHOO`, and `IContext.getiWidgetAttributes().getItemValue("bgColor")` should return value.

## 5 Widget communication

### 5.1 Simple events

The widget framework supports simple events. Widgets can register an event name with an event handler within the framework. The widget itself or another page component can fire this event, which will trigger the framework to invoke the registered event handler. See the following notes:

1. Event names must start with `on`, for example `onStart` and `onStop`.
2. Widgets can register event handlers by defining them in the widget definition.
3. `XML-syntax` is an attribute of the `<iw:iwidget>` element.
4. `<iw:iwidget name="stock" xmlns:iw="http://www.ibm.com/xmlns/prod/iWidget" iScope="stock" onStockChange="updateStock">`
5. Event handlers must be defined in the widget encapsulation class. In following example, `updateStock` is defined in the `stock` class in `stock.js`:

```
dojo.declare("stock",null,null,{
  onview: function () {
    var element = this.iContext.getElementById("stock");
    var attributesItemSet = this.iContext.getiWidgetAttributes();
    element.innerHTML = attributesItemSet.getItemValue("stock");

    var element2 = this.iContext.getElementById("company");
    element2.innerHTML = attributesItemSet.getItemValue("company");
  },
  updateStock:function(iEvent) {
    var element2 = this.iContext.getElementById("company");
    element2.innerHTML = iEvent["payload"];
  }
});
```

Note:

1. Use `Context.iEvents.fireEvent()` to fire this event within the same widget `IContext.iEvents.fireEvent("onStockChange",null,"yourCo");`
2. The widget framework provides event services that allow other page components to fire the same event. For example, another dojo widget on the same page may use the following service to pass a new company name to the stock widget:  
`serviceManager.getService("eventService").fireEvent("widgetId","onStockChange","yourCo");`

## 5.2 Widget event coordination

Widgets commonly coordinate events. For example, if you click a company name in a source widget, then it may fire an event to a stock quote widget, and the stock quote widget updates immediately. The widget framework provides an eventing mechanism to support this scenario.

Source widgets can define published events. Published events are the events that a widget can send to other widgets. A target widget can define handled events. Handled events are the events a widget can receive. Lotus Mashups provides a convenient wiring function that allows the `publishEvent` from a source widget to be wired together with the `handledEvent` of the target widget. After the wiring is complete, the source widget can publish the event and trigger the handled event on the target side.

To understand the wiring process better, follow along with this scenario:

1. A source widget declares a published event in its widget definition, as follows:

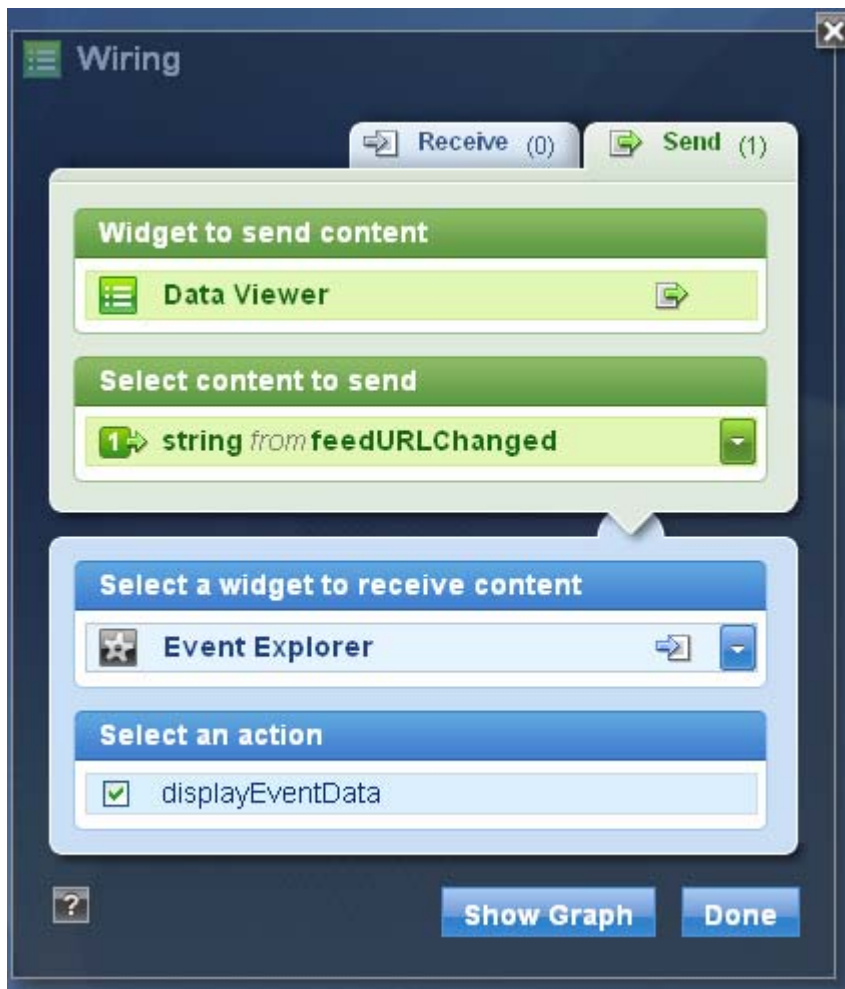
```
<iw:event id="cellDataSelected" published="true"
eventDescName="desc_cellDataSelected"/>
  <iw:eventDescription id="desc_cellDataSelected" payloadType="any"
    description="" lang="en">
    <iw:alt description="" lang="de"/>
    <iw:alt description="" lang="cn"/>
  </iw:eventDescription>
```

In this example, the source widget defines a publishedEvent called `cellDataSelected`.

2. A target widget declares a handled event in its widget definition, as follows:

```
<iw:event id="displayEventData" handled="true"
onEvent="displayData"
  eventDescName="desc_displayEventData"/>
<iw:eventDescription id="desc_displayEventData" payloadType="any"
  description="This event will take any type of data"
  lang="en">
  <iw:alt discription="" lang="de"/>
  <iw:alt discription="This event will take any type of data"
  lang="cn"/>
</iw:eventDescription>
```

3. Lotus Mashups provides a convenient wiring interface that allows users to wire two widgets together while editing the page. Here is an example of the wiring interface:



In this example, the **Data Viewer** widget's `cellDataSelected` event is wired to the **Event Explorer** widget's `displayEventData` event.

4. The source widget publishes the source event, so that it triggers the target event in target widget, as shown in the following string:

```
this.iContext.iEvents.publishEvent("cellDataSelected",
message.dataInfo.cellData);
```

5. The mashup author decides to preconfigure a static page that contains widgets that are wired together. To do this, the author must define a `ReceivedEvent` in the target widget instance. Using the example from above, the author adds a `ReceivedEvent` to the **Event Explorer** widget instance, as shown in the following code:

```
<span class="mm_ReceivedEvent" >
  <a class="mm_SourceEvent"href="#1" style="visibility:hidden"
> cellDataSelected </a>
  <span class="mm_TargetEvent" style="visibility:hidden" >
displayEventData </span>
</span>
```

## 5.3 Payload and payloadType

When a source widget publishes an event, the widget provides a data object called a *payload*. A payload can be a simple JavaScript object such as string and number. It can also be a very complicated JavaScript object. Usually the *payloadType* is defined when a published or handled event is defined, for example:

```
<iw:event id="updateStock" published="true"
eventDescName="desc_updateStock" />
<iw:eventDescription id="desc_updateStock" payloadType="company"
description="" lang="en">
  <iw:alt discription="" lang="de"/>
  <iw:alt discription="" lang="cn"/>
</iw:eventDescription>
```

Lotus Mashups also allows you to use the `payloadDef` element to describe a specific `payloadType`, for example:

    PayloadType -- "company" may contain 2 fields: name and address

```
    <iw:payloadDef name="company" >
        <iw:payloadDef name="name" type="string"
defaultValue=""
description="company name"/>
        <iw:payloadDef name="address" type="string"
defaultValue=""
description="company address"/>
    </iw:payloadDef>
```

Here are some guidelines that the wiring framework follow for events:

1. The wiring framework supports the payload type any. When publishing (sending) events, any means that the payload type that is sent is flexible and can be wired with any receiving event. When handling (receiving) events, any means that the event can receive events with any kind of payload type.
2. For payload types other than any, the wiring framework checks the type and allows wiring only when the published and handled events have a matching payload type.

## 6 Packaging and deploying widgets

### 6.1 Package types

Currently you can package widgets either as a Web application WAR file or an Eclipse plug-in resource bundle JAR file. Recommended best practice is to use WAR files since that is the supported use case with clusters. Regardless of which type you choose, the widget definition files must be accessible from a fixed URL. This is very important for widget registration. The resources and artifacts of the widgets must also be referenced relative to the location of the widget definition file. The deployment methods are different for WAR and JAR widgets.

### 6.2 Deploying widgets as WAR files

Since WAR-based widgets are in a Web application, you can deploy them to Lotus Mashups in two different ways. First, you can deploy the WAR files from the WebSphere Application Server (WAS) administrative console and register them (see section 6.2.1). Second, you can deploy them from MashupHub (see section 6.2.2)

#### 6.2.1 Deploying widgets from the WAS administrative console

To deploy a WAR widget, do the following steps:

1. Locate the widget registry directory. For WAS installations, the directory is `<LMInstallRoot>\mm\public\cat`.
2. Use either the master registry or the user's registry catalog file. Normally, right after Lotus Mashups is installed, you should find only one `default_catalog.xml` file. Changes made in this file are also made in the widget catalog for all other users. If a user has already generated a catalog, then the changes in the master catalog are not reflected in the user's catalog. In this case, you must remove the user's catalog and regenerate one. In this section, we use the master registry catalog file named `default_catalog.xml`.
3. Find the relative path to the widget definition XML file. For example, if your WAR widget has the context root `myWidget`, your widget definition XML file is named `helloworld.xml` and is located in a directory named `widgetDef`, then the relative path to access the widget definition should be as follows:

```
myWidget/widgetDef/helloworld.xml
```

4. Using your favorite text or XML editor, open the XML file and either add an entry to a catalog or create a new catalog and put the new widget inside it. Each catalog in the catalog registry represents a drawer in the toolbox of Lotus Mashups. Here is an example:

```
<category name="MyNewCata">
  <title>
    <nls-string lang="en">MyNewCata</nls-string>
  </title>
  <description>
```

```
<nls-string lang="en">Description of the MyNewCata</nls-string>
</description>

<entry id="helloworld" unique-name="helloworld" alias="helloworld">
<title>
  <nls-string lang="en">Hello World</nls-string>
</title>
<description>
  <nls-string lang="en">A simple Hello World widget</nls-string>
</description>
<definition>myWidget/widgetDef/helloworld.xml</definition>
<content>http://www.ibm.com</content>
<preview>http://www.ibm.com</preview>
<icon>myWidget/images/generic_widget_icon.gif</icon>
<entry>
</category>
```

5. Save the XML file, and restart the server. Now the widget should appear in the toolbox.

## 6.2.2 Deploying widgets through MashupHub

You can upload WAR widgets to MashupHub and then deploy them to Lotus Mashups. This adds the widgets to the Lotus Mashups toolbox. To deploy a widget using MashupHub, do the following steps:

1. From the MashupHub home page, click the **Upload Widget** link.
2. In the **Upload Widget** window, select **iWidgets** as the source, and click **Next**.
3. In the **Upload or Register Widget** window, select one of the following two options:
  - Upload the WAR file – the widget is deployed to Lotus Mashups and appears in the toolbox
  - Host the widget remotely and register it as a URL – the widget is sourced from its original location, and the widget appears in the toolbox

## 6.3 JAR widget deployment

JAR widgets are packaged as Eclipse plug-in resource bundles. Widgets developed in this way extend the Eclipse plug-in extension point of `com.ibm.mm.widgets.iwidget`, which is defined by Lotus Mashups. To deploy this type of widget, do the following steps:

1. (1) Locate the Lotus Mashups Eclipse plug-in directory.

For WAS installations, the directory is `<LMInstallRoot> \mm\eclipse\plugins`.

- (2) Copy the JAR file into this directory.

(3) To allow the new widgets to appear in the user's toolbox, remove and regenerate each user's registry file.

(4) Restart the server. Now the widgets should appear in the toolbox. If a user does not see these widgets in the toolbox, you may need to regenerate the user registry. To do this, remove the user registry file, restart the server, and log in.

## 7. Widget development tools

With servers capable of rendering and mashing up widgets, various tools are evolving for developing widgets. Two of these tools are Lotus Widget Factory and WebSphere sMash. Widget Factory is based on the same stack as Portlet Factory, but Widget Factory is focused more on widgets. WebSphere sMash is a new tool that you will find interesting if you are into dynamic scripting platforms such as Groovy and PHP.

### 7.1 *Widget Factory*

Widget Factory provides an easy-to-use development environment enabling developers of all skill levels to rapidly create dynamic widgets without writing code. Widget Factory contains dozens of software automation components known as **builders** that you can snap together using a wizard-based user interface to create **models**. These models generate the code and metadata that comprise your widgets.

We recommend that you first install IBM Mashup Center on your machine, and then install Widget Factory separately.

To create a simple widget using Widget Factory, do the following steps:

1. From the Widget Factory Designer menu, select **File >New >Lotus Widget Factory Project**. The default project creation settings are set up for your project to automatically deploy to your local Lotus Mashups WAS server. Some sample widgets are included.
2. Name the project.
3. Deploy the project. Creating a project launches an initial deployment to the server, which takes a approximately 20 seconds.  
Once you have completed the initial deployment, any subsequent changes you make in the project are immediately copied to the server without requiring any redeployment.
4. To create a widget, launch the Widget Factory **New Model** wizard.
5. Click **File >New >Lotus Widget Factory Model**, and then select the project to contain your new model from the list of model types already available. For example, the **Excel Widget** choice launches a wizard that guides you in creating a widget model that displays data from an Excel spreadsheet.
6. Fill in the fields, and name the model. A new model is created with a number of builder calls in it. You can run the new model as-is or modify it by adding new builder calls or changing the inputs.
7. Any Widget Factory model that contains a **Widget Adapter** builder call can be published to the Lotus Mashups toolbox. To publish the widget to Lotus Mashups, do the following steps:
  - A. Right-click on your project and select **Widgets >Publish Widgets to Lotus Mashups toolbox**.
  - B. Type your credentials (user name and password) for your Lotus Mashups account.
  - C. Select the widgets you want to publish from the summary dialog box.
  - D. Click **OK**. The selected widgets are installed in the Lotus Mashups toolbox, using the categories specified in each widget model's Widget Adapter.

### 7.2 *WebSphere sMash:*

WebSphere sMash is both a development and execution environment for dynamic Web applications. The sMash development team is working toward providing an application builder environment for developing widgets using a Web interface.



## 8 *Widget NLS support*

All widgets delivered by IBM that are based on the iWidget specification must, at a minimum, support group-one languages. To support additional languages, do the following steps:

1. Make sure all the strings displayed on your widget screen are not hard coded. These strings should come from a resource or property file instead.
2. In your main widget JavaScript file, add the following line:

```
dojo.require("dojo.i18n");
```

3. Organize all your language-dependent strings by replacing `<lang>` with the language code, for example `en-us` for American English.

```
js\nls\<<lang>\myWidgetStrings.js
```

4. Register the resources in the `onLoad` method so that the widget's JavaScript can reference them, for example:

```
dojo.registerModulePath("myWidgetModule ",
    this.iContext.io.rewriteURI("js"));

dojo.requireLocalization("myWidgetModule ",
    "myWidgetStrings");

this.resourceBundle = dojo.i18n.getLocalization("myWidgetModule
",
    "myWidgetStrings ");
```

5. Now you can reference any string defined in the `myWidgetStrings.js` file. For example, if you have defined the string `myWidgetStr01 = this is just a string`, then you can reference it like this:

```
This.resourceBundle.myWidgetStr01
```

The Lotus Mashups build process has added an ANT task to convert the resource property files to JavaScript resource files so that you can use either the property file or the JS resource file. If you choose to use the resource property files for the NLS support, these files will have to be converted during the build process.

### 9 Widget development – OSGI

You can package widgets in multiple ways. Packaging OSGi plugins as widgets is one of the ways, since this is not a supported use case for V1.0, we will leave that to the developers and in later versions we will add support for it.

# 10 Feeds

## 10.1 Feed Readers

In the Lotus Mashups programming paradigm, all client-server communication happens via feeds. Lotus Mashups provides a graphical browser-based mashup builder for viewing and assembling feeds and widgets. All the feeds published on the Mashup Hub are read through the feed reader widgets described below. The Mashup builder has two several out-of-the-box widgets that allow you to read and integrate feed data. This section describes these widgets.

## 10.2 Feed Reader

Lotus Mashups provides a general feed reader widget that reads and displays Web content provided via RSS and Atom feed formats. After you configure the widget, the **Feed Reader** widget checks for new content at a user-defined interval.

### User interace

When added to a page, the **Feed Reader** widget displays the contents of the default feed that is defined in the widget definition file. If no default feed exists in the file, the widget will not display any data at all. The default feed is defined in the `feedReader.xml`, which is part of the default catalog. See the following value:

widget when the page is in edit mode. Here are some descriptions of the configuration fields:

**Feed URL:** The data feed URL.

**Title:** The title that displays in the widget.

**Items to display:** The number of items to display per page. If more data than is specified here is retrieved, the widget displays a paging option.

The following check boxes are available in the configuration window:

**Publish the link URL after click:** When selected, the widget publishes the URL of the selected item to any wired listeners. This URL is the `link` attribute of the feed entry. Any widget that is wired to receive the URL retrieves the details of the item. For example, the **Web Site Displayer** widget invokes the URL and displays the returned content (as long as the returned content is HTML markup).

**Open links directly onto site:** This option instructs the widget to open or display the contents of the selected item directly in the current browser. The content retrieved is defined by the `link` attribute of the selected entry.

**Show more details:** Instructs the widget to show the title only or both the title and summary content of an entry.

The **Advanced** section of the configuration window has the following options:

**Refresh rate:** Specifies the how often the feed gets refreshed.

**Display as title:** Determines which field in the feed is used as the entry title in the displayed feed contents. The options are the entry title (Article), link reference (Link), the description (Description), or the publication date (PubDate).

**Parameters:** The widget can parse any URL parameters and present an input box for their values. This means the URL can be changed in the widget if the arguments have other options. You can add a new parameter by clicking the **App Parameters** button.

### Wiring the Feed Reader widget

As we discussed in the configuration section, you can configure the **Feed Reader** widget to publish the URL of a selected item. When the widget is configured to publish a selected URL, it publishes the following event:

```
(eventName,payloadType) = ("ItemSelected","string")
```

This means that the event named `ItemSelected` with a payload type of `String` is published. The string is the link attribute in the feed definition of the selected item. The **Web Site Displayer** widget is the natural partner of the **Feed Reader** widget when the link URL results in HTML that can be rendered.

The **Feed Reader** widget can also receive an event with the feed URL that provides its content. The widget will listen for the following event:

```
(eventName,payloadType) = ("getFeedFromURL","JSON")
```

The event payload is a JSON object in the form `{url:<the_feed_url>}`.

## 10.3 Data Viewer

Lotus Mashups provides a **Data Viewer** widget that is designed for viewing tabular data. You can specify which data to display, and configure it to display various styles. You can also wire it to receive content and publish the selected table content. See the following sections for more information.

### Data Format

The **Data Viewer** widget expects to receive content in string format. It parses the input string and displays the encoded row and column data. The content must be in the following format:

```
column1,column2,...,columnN|type1,type2,...,typeN|val11,val12,...,val1N|...|valM1,valM2,...,valMN
```

where:

```
columnI = the Ith column name (First Name),
typeI = the Ith column's data type (string),
valIJ = the value of the Ith row and Jth column of data
```

If the data value (`valIJ`) contains a comma (,) or a vertical bar (|), the value must be replaced by its unicode equivalent:

Char	char code	replacement	example
,	44	&#44;	a,b becomes a&#44;b
	124	&#124;	a b becomes a&#124;b

### Configuring the Data Viewer widget

You can configure the **Data Viewer** widget when the page is in edit mode. Here are descriptions of the configuration fields:

**Content URL:** URL for the widget's content. The widget expects the HTTP content type to be `text/plain`, and the text must be in the format outlined above.

**Rows Per View:** The number of rows to display on a page.

**Selected Fields:** Allows you to select which columns of the returned data to display. The fields that begin with a double underscore (\_\_) are assumed to be hidden. Other fields are displayed by default.

The **Advanced** section of the configuration window allows you to filter or highlight data based on simple matching criteria. If the value of a selected column matches the selection criteria, the value is displayed or highlighted.

### Wiring the Data Viewer widget

You can wire the **Data Viewer** widget to send events with payloads that contain cell or row data. When you open the wiring interface, you will see the following events:

**cellValueSelected:** This event publishes the value of a selected cell to a receiving widget, for example:

```
(event,payload) = (cellSelectedValue,string)
```

In this example, the string is a string-based representation of the cell data.

**cellDataSelected:** This event publishes the column, type, and value of a selected cell to a receiving widget, for example:

```
(event, payload) = (cellDataValue, string)
```

In this example, the string represents the standard data format `column | type | value`.

**rowDataSelected:** This event publishes the column, type, and value of each cell in the selected row to a receiving widget, for example:

```
(event, payload) = (cellDataValue, string)
```

In this example, the string represents the standard data format `column1, ..., columnN | type1, ..., typeN | valM1, ..., valMN`,  
The Mth row is selected.

**rowJsonDataSelected:** This event publishes the column, type, and value of each cell in the selected row to a receiving widget, for example:

```
(event, payload) = (cellDataValue, json)
```

In this example, the value is a JSON representation of the row data.

The **Data Viewer** widget can also receive the event inventory with a payload in the defined format. The event is of the form `(event, payload) = (inventory, string)`, where the string is the tabular data in the standard widget format.

## *11 Proxy Server*

Web browsers impose a security restriction on network connections, including HTTP requests. In short, a script or application cannot make a connection to any Web server other than the one its Web page originated. Some browsers allow cross-domain connections if the option is enabled, but this is not a general deployment scenario. The nature of Lotus Mashups requires that it consume feeds from various sources that will almost certainly originate from different domains. To this end, Lotus Mashups embeds a proxy server. Instead of making a request directly to the cross-domain Web service, the request should be directed at the proxy server. The proxy will redirect the call to the intended Web service and pass the return to the calling application.

In order to invoke the proxy server, you must use a URL with a specific format. Lotus Mashups directs HTTP requests with the following format to the embedded proxy server

`/mum/proxy/<protocol>/<URL>`

where `<protocol>` is the communication protocol (HTTP), and `<URL>` is the target URL.

For example, to access the Web service at <http://www.ajax.com/abc?arg1=val1>, you would use the URL `/mum/proxy/http/www.ajax.com/abc?arg1=val1`.

You could alternatively use your own proxy if there is one available on your server. The settings for the proxy url comes from bootstrap.jsp2 under `<install_root>/mm/eclipse/plugins/enabler_1.0.jar/bootstrap`.

Some of out of the box proxy settings can be changed from proxy plug-in under `<install_root>/mm/eclipse/plugins/com.ibm.mm.framework.proxy_1.0.0.jar/web-inf/proxy.config.xml`. Refer to admin topics under Mashup Center product wiki.

## 12 Advanced Topics

### 12.1 Adding Back-end Java code to your widget

Most widgets only consume external feeds and do not need any custom back-end processing except the built-in proxy. However, when custom back-end processing is needed, widget developers usually create a servlet that they access within their widget via an Ajax request.

In this section, we show you how to add servlets to your OSGI plug-in without the need to deploy them into a Web application. This is beneficial because it keeps your servlet Java code bundled in the plug-in where your widget resource files are.

The extension point that is used to register a servlet and to map it to an alias is `org.eclipse.equinox.http.registry.servlets`. The servlet class is based on the standard `javax.servlet` API. Therefore, you need to add the `javax.servlet` and `javax.servlet.http` packages in the **Imported Packages** section of `manifest.mf`.

Here is a servlet extension point example:

```
<extension point="org.eclipse.equinox.http.registry.servlets">
  <servlet alias="/fooPath" class="com.my.plugin.FooPathServlet"/>
</extension>
```

To avoid possible conflicts with other plug-ins, we recommend that you prefix the aliases with your plug-in ID, for example:

```
<extension point="org.eclipse.equinox.http.registry.servlets">
  <servlet alias="/myplugin.iWidget/fooPath"
class="com.my.plugin.FooPathServlet"/>
</extension>
```

**Tip:** From your JavaScript code, you can use the `widgetSupport.widgetUri` method (available when using `com.ibm.widgets.support.js`) to compute the URL of the servlet without hard coding the context root. This makes your code more portable when the plug-in is installed on a different host Web application server.

Here is an example using the `dojo.io.bind`:

```
dojo.xhrGet( {
  handleAs: "text",
  url: widgetSupport.widgetUri("com.my.plugin/fooPath"),
  load: function(/*String*/data, ioArgs ) {
    //do something
  },
  error: function( /*Object*/error, /*Object*/ioArgs){
    alert( error.message );
  },
});
```

Note: Above approach is not recommended since updates to product binaries will remove your changes. Alternatively you can either package your server code in the widget war file or create your own OSGi plug-ins.

## 12.2 Enabling person tags

In this section, we discuss how to enable person tags in your widget.

Semantic tagging allows you to add special meaning to your markup. For example, adding a person tag to a person name will automatically add a pop-up button to the name when users hover their cursors over it. When clicked, the button will open a pop-up window that shows business information about the person.

The semantic tagging service is globally available to your widget. You can access it by simply using the `SemTagSvc` object. You do not need to include any other script in your code. However, we recommend that you always test the availability of the object. Here is an example:

```
if ( typeof SemTagSvc != "undefined"){    //do something }
```

To use person tags, you must format the HTML using HTML classes known by the semantic tag parser. Typically, you will use `span` HTML elements to format the entry, for example:

```
<span class="vcard" style="font-weight:normal;color:#578CCA;">
  <span class="fn"> John Smith </span>
  <span class="email"
style="display:none;">john.smith@acme.com</span>
</span>
```

When parsing the DOM, the `vcard` class is recognized as a person tag, and the semantic tag parser will automatically add the necessary code to show the pop-up button and the business card pop-up window.

If you are creating the markup dynamically, then you must explicitly call the semantic tag parser on the DOM node that contains the new markup, for example:

```
divNode.innerHTML = ' <span class="vcard" style="font-
weight:normal;color:#578CCA;"> '+
  '<span class="fn"> John Smith </span>'+
  '<span class="email"
style="display:none;">john.smith@acme.com</span>'+
  '</span>';

if ( typeof SemTagSvc != "undefined"){
  SemTagSvc.parseDom(null, divNode);
}
```

## 12.3 Events

Widgets can either receive (subscribe to) events or send (publish) events. When you wire widgets, you specify both the sending and receiving events based on the widget's metadata. In this section, we will show examples of how widgets can declare their wiring model using the widget definition XML file. Using the XML file is just one way to accomplish this. For more information, refer to the wiring model section in this guide.

### **Publishing events**

The example we will show here is found in the contacts widget located in the `com.ibm.mm.widgets.samples` plug-in.

In this example, the widget declares to be the source of an event called `PeopleSearch` with a payload type of `people`. The payload type needs to be defined in the widget declaration as well.

```
<iw:payloadDef name="people">
  <iw:payloadDef name="userId" type="string" defaultValue=""
description="Intranet email address"/>
</iw:payloadDef>
<iw:publishedEvents>
  <iw:event eventName="PeopleSearch" payloadType="people"/>
</iw:publishedEvents>
```

The wiring interface uses the declaration above to allow you to create new wires between widgets. Now the widget needs to add logic in its code to publish the event to all the wires based on a user action.

Fortunately, there is a simple API to publish the event:

```
publishPerson: function( message ){
  var node = message.source;
  var contact = node.contact;
  if ( !contact.group ){
    this.widgetClass.iContext.iEvents.publishEvent("PeopleSearch",
{ "userId": contact.email }, "people");
  }
}
```

The `publishEvents` API is accessible from the `iContext.iEvents` object. The `iContext` object is accessible from the widget JavaScript class defined with the `iScope` attribute. To avoid conflict, we recommend that you do not keep a global reference of the `iContext` object but rather pass that reference to the JavaScript objects that need it. In the example above, `publishPerson` is a method of a Dojo widget that keeps reference of its widget class in the `widgetClass` field.

### Subscribing to events

Similar to publishing events, the widget declares to be on the receiving end of an event using the `<iw:handledEvents>`, for example:

```
<iw:handledEvents>
  <iw:event eventName="PeopleSearch" payloadType="people"
onEvent="handlePeopleSearch"/>
</iw:handledEvents>
```

The `onEvent` attribute is the method that is called by the framework when an event is actually fired. The method takes an `iEvent` object as a parameter and gets implemented in the widget class, which in turn can dispatch it to the right widget as needed.

In the profile widget located in the `com.ibm.mm.widgets.samples` plugin, the method is implemented as follows:

```
handlePeopleSearch: function( iEvent ){
  var userId = iEvent.payload.userId;
  var tabContainer = dojo.widget.manager.getWidgetById(
this.domID + "profileTabContainer");
  var searchContentPane =
dojo.widget.createWidget("ContentPane", { "label": userId } );
  searchContentPane.closable='true';
  tabContainer.addChild( searchContentPane );
```

```
        tabContainer.selectChild( searchContentPane );

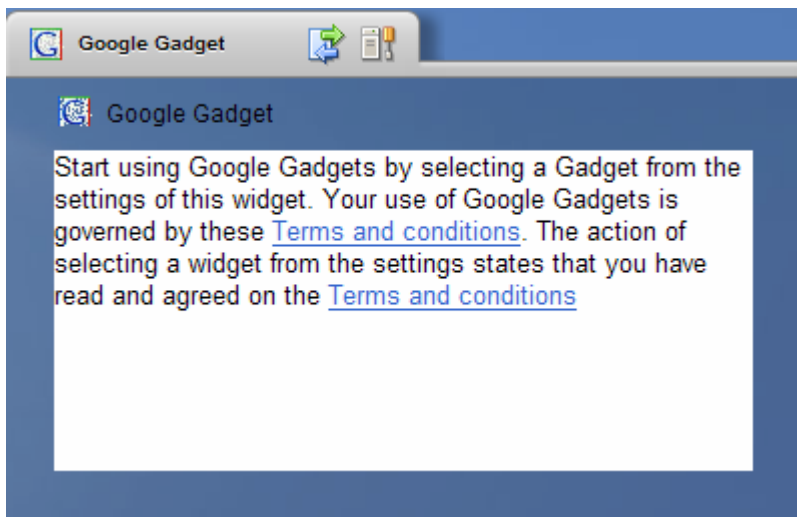
        var profileWidget =
dojo.widget.createWidget("profilewidget", {"userId" :userId } );
        searchContentPane.addChild( profileWidget );
    }
```

## 12.4 Generic Google Gadget widget

The generic **Google Gadget** widget provides access to the Google Gadget directory where you can select the Google Gadget of your choice. When you select a Google Gadget, the Google Gadget widget automatically generates the preferences user interface based on the user preferences defined in the Gadget module file.

### User interface

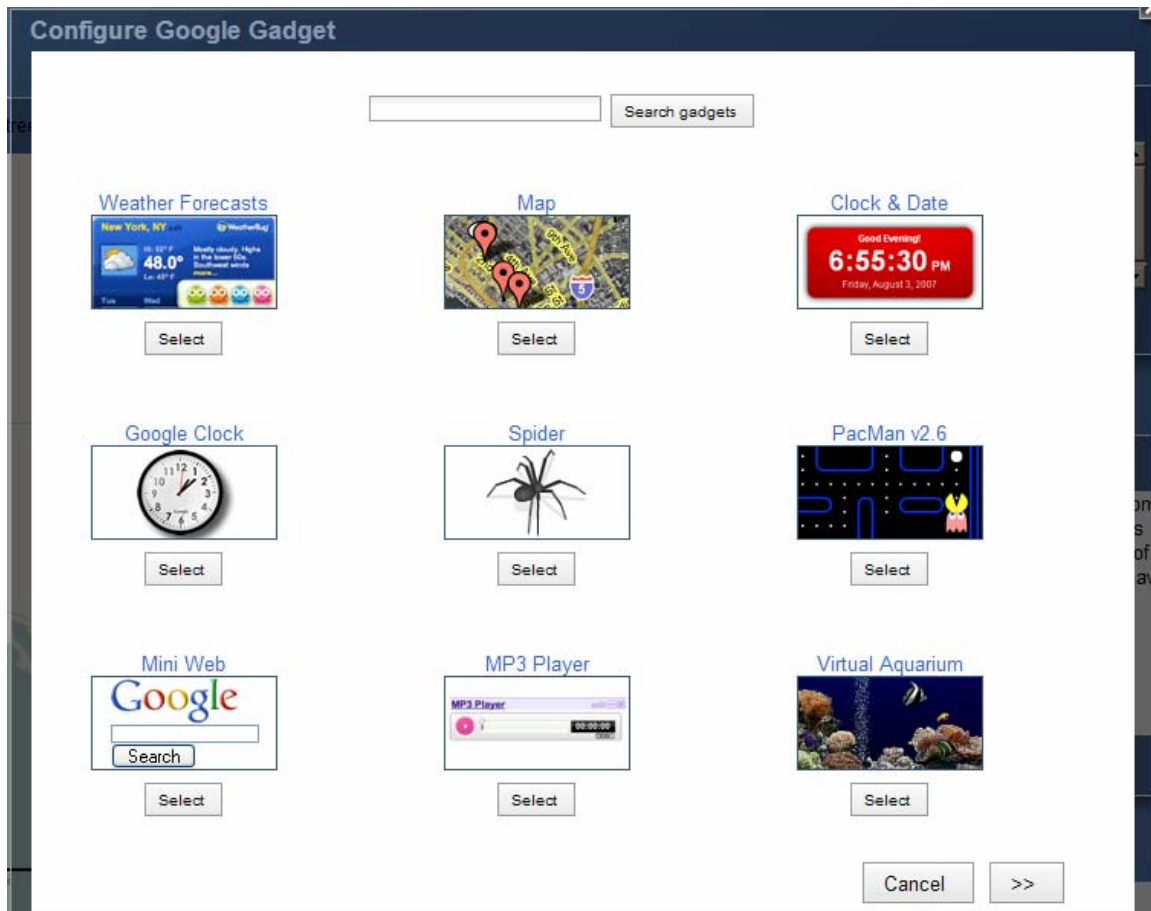
When you first drag the Google Gadget widget onto the page, the following text displays with an option to switch to edit mode and select the gadget to display.



### Selecting a gadget:

You can switch to edit mode by selecting edit settings from widget context menu. The google gadgets directory panel displays as below with a search box that allows you to search on all possible gadgets on google web site. You can browse through the display results using '>>' (next) or '<<' previous buttons.

You need to click on 'select' button under the gadget to select a particular gadget. Basing on your selection next screen will display all possible parameters you can pass to that gadget.

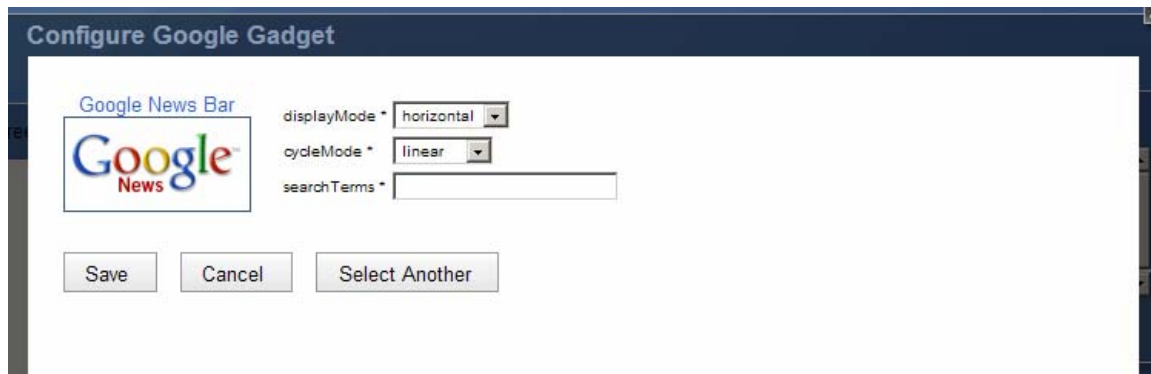


**Gadget Preferences:**

The preferences user interface is automatically generated by the widget. Each user preference specified in the Gadget is converted into an input control as follows:

- String      Text box
- Bool        Combo box with True/False
- Enum        Combo Box with enum values
- Location    Text box
- List        Composite UI made of
  - A list box for holding the values
  - An input box for adding a new value
  - Add Button
  - Remove Button

**Preferences user interface**



After making changes to preferences, you can click **Save** to update the Gadget rendering.

Sometimes, the width and height of a particular Google Gadget are dynamic and can vary basing on data. You can resize the **Gadget** to adjust the height basing on your need. You can also change it to point to another gadget by going back to edit settings and using **Select Another** push button option.

### Wiring

Wiring option for Google gadget is very powerful that enables all the possible preferences the gadget can accept as actions that it can receive. This feature unlocks all those gadgets to talk to other widgets at on the page.

## 12.5 Widget Authentication

In this section, we will show you an example of how you can provide widget authentication.

### Use case:

You need to write a widget that access a feed server that requires user authentication. An example might be a widget that displays a list of activities for a particular user. The user should have to authenticate only once during the session.

### The LoginWidget - common Dojo widget and the authentication proxy

At this time, the Lotus Mashups proxy does not support authenticated feeds. As a temporary solution, the `com.ibm.mm.widgets.samples` plug-in provides a proxy that supports authentication. From the user interface, the sample plug-in also provides a `LoginWidget` Dojo widget that works with the authentication proxy. Using the `LoginWidget` widget in your widget is beneficial because it shields you from future code change.

*Note:* At this time, user credentials are stored in the session itself. This means that users only need to authenticate once per session. However, if a session terminates, users must re-authenticate.

To use the `LoginWidget` widget, add the following directive in your JS file:

```
dojo.require("com.ibm.mm.widgets.samples.common.LoginWidget");
```

The `LoginWidget` widget acts as a wrapper for other content widgets. It automatically detects if authentication is required. If authentication is required, it displays the login screen in place of the main widget window.

The main parameter for the LoginWidget widget is a loginConfig object, as shown in the following code:

```
var loginConfig = {
  doCreateLoginScreen: function(){
    return new com.ibm.mm.widgets.samples.common.BaseIWidget({
      postMixInProperties: function(args, frag, parent){
        //Template variable replacement
        this.widgetRoot = widgetSupport.widgetUri(
          "com.ibm.mm.widgets.demo/activities/"
        );
        this.templatePath =
"com.ibm.mm.widgets.demo/activities/templates/activitiesLogin.html";
com.ibm.mm.widgets.samples.common.BaseIWidget.prototype.postMixInProp
erties.apply(this, arguments);
      }
    });
  },
  doCreateContentWidget: function( loginWidget ){
    return new com.ibm.mm.widgets.demo.activities.ActivitiesWidget(
      { loginWidget: loginWidget}, document.createElement("div")
    );
  }
};
this.mainWidget = new com.ibm.mm.widgets.samples.common.LoginWidget(
  {loginConfig: loginConfig,}
);
```

The doCreateLoginScreen method is a callback that allows you to provide your own login screen widget. You can provide any user interface you want for the login screen, but there are a few requirements that you must follow. For example, you must provide the following dojoAttachPoint:

- userIdNode for the user Id
- pwdNode for the password
- loginButtonNode for the login button

Here is an example login screen:

```
<div>
  <label for="user">User name:</label>
  <br/>
  <input id="userid" dojoAttachPoint="userIdNode" name="userId"
class="text" type="text" />
</div>
<div>
  <label for="password">Password:</label>
  <br/>
  <input dojoAttachPoint="pwdNode" class="text" type="password"
name="pwd" />
</div>
<div>
  <br/>
```

```
<input class="text" type="button" value="Login"
dojoAttachPoint="loginButtonNode" />
</div>
```

The LoginWidget widget provides a bind method that is very similar to the dojo.io.xhrGet method, for example they take the same Request object. The main difference is that the LoginWidget widget routes the request to the authentication proxy by passing the URL as a parameter. The proxy maintains the credentials for the current user and uses them to authenticate with the target server. If no credentials are found or the credentials are invalid, the proxy sends an error code 407 (HTTP\_PROXY\_AUTH). The LoginWidget widget picks up the error and, as a result, automatically displays the login screen.

This means that you need to keep a reference to the LoginWidget widget and call its bind method in place of the regular dojo.io.xhrGet method.

In the following example, we use another common object called com.ibm.mm.widgets.samples.common.AuthFeedParser that provides parsing functions for authenticated widgets:

```
dojo.declare("com.ibm.mm.widgets.demo.activities.ActivitiesFeedWidget",
com.ibm.mm.widgets.samples.common.commonWidget,
{.....
  postCreate: function(){
    this.authParser = new
com.ibm.mm.widgets.samples.common.AuthFeedParser( this.feedUrl );
    this.authParser.loginWidget = this.loginWidget;
    this.authParser.nodeLoaderCallback = function( node, entry ){
      alert( node );
    };
    this.activitiesNode.innerHTML = this.message;
    dojo.subscribe(this.authParser.eventNames.feedLoaded, this,
"onActivitesLoaded" );
    this.authParser.load();
  }, .....});
```