Developing Gadgets for webMethods Business Console

Innovation Release

Version 10.0

April 2017

WEBMETHODS

This document applies to Business Console Version 10.0 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 2014-2017 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at http://softwareag.com/licenses.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at http://softwareag.com/licenses and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at http://softwareag.com/licenses and/or in the root installation directory of the licensed product(s).

Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided for in your License Agreement with Software AG.

Document ID: BC-DG-100-20170419

Table of Contents

1 Introduction to Business Console

Overview

Business Console gadgets are independent pluggable components that can be rendered using the Business Console gadget framework on a My webMethods Server instance. The Business Console gadget framework is a client-side JavaScript framework, which enables rendering of gadgets in Business Console by using each gadget's metadata information. Business Console gadgets enable you to customize dashboards in Business Console .

Business Console offers built-in gadgets for creating dashboards. However, you can create your own gadgets and use them in Business Console dashboards.

Currently, the use of gadgets is limited to Business Console . This document provides guidelines for creating gadgets for use inBusiness Console . You will learn to:

- Program gadgets
- Create the user interface for gadgets
- Configure gadgets
- Test gadgets

Pre-requisites

To use this guide effectively, you should have good knowledge of using:

- JavaScript, XML, HTML, and CSS
- RESTful services
- AngularJS
- Portlet applications and web applications
- Composite Application Framework in Software AG Designer
- New Business Console Gadget wizard in Designer
- webMethods Business Console

For information about creating portlet applications and web applications in Software AG Designer, see *webMethods CAF and OpenCAF Development Help*.

For information about using gadgets in Business Console , see *Working with webMethods Business Console*

Using JavaScript for gadget development

Use JavaScript to program each gadget to handle business logic in an application. You can use JavaScript to process the data received from the server (using the underlying JavaScript APIs), perform data manipulation, and update the user interface of the gadget.

Gadgets can invoke RESTful services for:

- Retrieving data
- Updating the gadget user interface after receiving data
- Firing event (limited to AngularJS gadgets only) and notifying any updates to other gadgets

Understanding AngularJS and non-AngularJS gadget development

AngularJS is a client-side web application framework supported by Google, enables you to create Single Page Applications (SPA). AngularJS framework adapts and extends traditional HTML to present dynamic content through two-way data-binding that allows automatic synchronization of models and views.

AngularJS is built on a declarative programming model that places markers known as directives on the Document Object Model (DOM). DOM element manipulation is against the construct of AngularJS, but AngularJS allows DOM manipulation with the use of custom directives.

Non-AngularJS frameworks that support imperative programming model such as JQuery, allow remote selection of DOM elements, and manipulation of DOM elements. However, using element IDs for DOM selection and manipulation might not always be the best approach. For example, if a single gadget is embedded multiple times in a page, and if you use element IDs for DOM selection, only the first gadget in the DOM would be selected, and would ignore other gadgets. When a gadget is used multiple times in a single page, it helps to use AngularJS custom directives because the custom directives automatically pass the respective element references to the directives link function.

You should consider the differences between AngularJS framework and non-AngularJS framework, and choose either a non-AngularJS (imperative) approach or AngularJS (declarative) approach for developing gadgets. If you decide to use JQuery or any other alternatives for DOM manipulation, use AngularJS directives for DOM elements.

We recommend that you use AngularJS for developing gadgets.

Using Model View Controller (MVC) in AngularJS

AngularJS uses the following Model View Controller (MVC) architecture for organizing applications.

- Model for managing data received from a database or from a JSON file.
- View for displaying the model.
- Controller for programming the interaction between the Model and View.

Use the MVC architecture for developing the user interface of a gadget. MVC architecture helps in separating the gadget logic from gadget data and gadget view. For each gadget:

- Create a Model to manage the gadget data and respond to requests from View and instructions from Controller.
- Create a View to display the gadget data.
- Create a Controller to control the interactions between Model and View, receive input, validate input, and perform operations to modify gadget data.

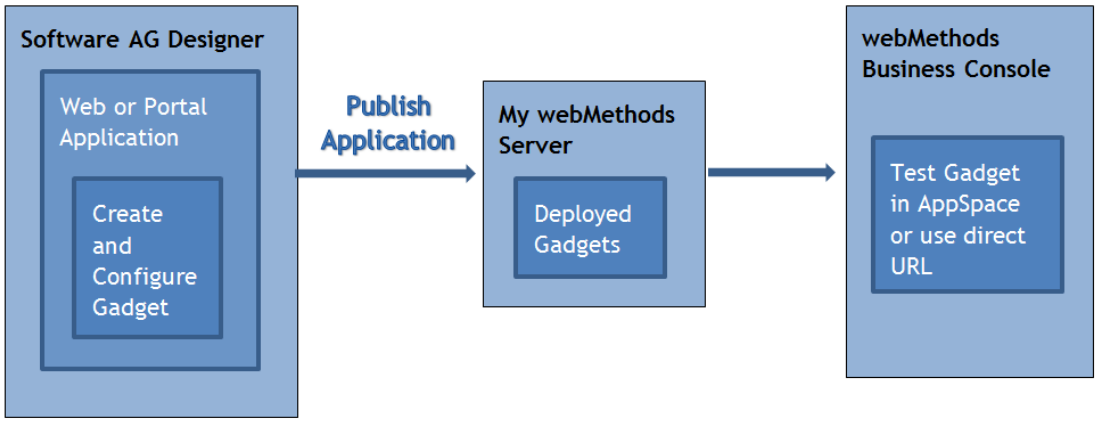
Organizing gadget files

Gadget information is organized in the following folders when you use the New Business Console Gadget wizard for creating a gadget in Designer. For information about the files in these folders, see *webMethods CAF and OpenCAF Development Help*.

Folder	Contains...
Images	Image files to be used by the gadget.
Scripts	JavaScript files for programming the gadget.
Views	HTML or XHTML files for defining the user interface of the gadget.
Styles	CSS files for defining the styles for the gadget.

Understanding Business Console gadget development

The diagram below shows the Software AG products required for developing and testing gadgets.



Steps you need to perform for creating, deploying, and testing gadgets:

1. Create a portlet or web application in the **UI Development perspective** in Software AG Designer.
2. Create gadgets in the application project.
3. Define the user interface and business logic for the gadgets.
4. Add functions to the gadget controller.
5. Publish the application to deploy the gadgets to My webMethods Server.
6. View gadgets either by using the gadgets in a Business Console dashboard or by using gadget's direct URL.

Creating an application

The first step in creating gadgets for Business Console is to create a portlet application or a web application for the gadgets to reside.

Use Composite Application Framework in Software AG Designer to create a portlet or web application project.

For more information about creating a portlet or web application project by using Composite Application Framework (CAF) in Software AG Designer, see *webMethods CAF and OpenCAF Development Help* .

To create a web or portal application project in Designer

1. In the **UI Development perspective**, select one of the following options depending on whether you want to create a web application or portlet application:
   - **File > New > Web Applications Project**
   - **File > New > Portlet Application Project**
2. In the application wizard, provide the project name, and click **Finish.** This will create an application project under which you can create multiple gadgets for Business Console.

Generating a gadget

To generate a new gadget in an application project

1. Select the **UI Development perspective** in Designer .
2. Select the web or portlet application project where you want to create the new gadget.
3. In **Solutions** view, expand **User Interfaces**, right-click on the project where you want to create a new gadget, and select **New Business Console Gadget**.
4. In the **New Business Console Gadget** wizard that opens, provide the following specification for the new gadget. The **New Business Console Gadget** wizard creates the configuration files and definition file for the new gadget.

Field	Description
Gadget Type	Specify AngularJS for AngularJS based gadgets or Default with empty stubs for non AngularJS gadgets.
Gadget Root Directory	Specify a name for the folder in which the new gadget should be stored under project's **WebContent** node (optional) . If you do not specify a folder name, the new gadget will be stored directly under project's **WebContent** node.
Gadget Name	Specify a name to identify the new gadget.
Gadget Title	Specify the title to be displayed on the gadget.
Preview Image	Browse and select an icon in `.png` or `.jpg` format for the gadget. The image size should

Field	Description
	not be more than 50KB, and the recommended size for the image is 70 X 70.
Settings Title	Specify a name for the gadget settings dialog box.
Description	Provide a description for the new gadget.
Gadget ID	This is an auto-generated identifier for the gadget.
Gadget Group Name	Specify ■ **Use project name** if you want to use the project name as the gadget group name. ■ **Use custom name** if you want to enter the name for the gadget group in the input field. The group name provided here will be used to categorize gadgets in the **Add New Gadget** dialog in Business Console.

Updating the user interface of a gadget (view.xhtml)

A new gadget will reside under a portlet application or web application project. If you have specified the root directory during gadget creation, the gadget would be in the root directory.

To update the user interface of a gadget

1. Navigate to **Portlet/Web Application Project > Gadget_root_directory > Gadget_name > views**.
2. Double-click **view.xhtml** and edit the file.
3. Provide the HTML code in **view.xhtml** to define the user interface of the gadget.

Adding functions to a gadget controller (controller.js)

After you define the user interface for the gadget, add the business logic for the AngularJS based gadgets in the `controller.js` file.

To define the business logic for an AngularJS based gadget

1. Navigate to **Portlet/Web Application Project > Gadget_root_directory > Gadget_name > scripts**.
2. Open **controller.js** for edit and specify the client-side business logic for the gadget.

Codeblock	Description
`URLS`	This section is for specifying a JavaScript array of RESTful service URLs for the gadget. You must provide relative URLs. Server details can be provided at runtime.
`init`	Constructor block which initializes the core services with the `$scope` object, including `config` (gadget configuration object), `restClient` (AngularJS based service to invoke the RESTful services), `eventBus` (AngularJS based object to pass events to the listening controllers and also receive events fired from other controllers), and `URLS` (the URL object mentioned in the URL section).
`defineScope`	This section allows you to define the JavaScript functions to be added to AngularJS `$scope` object. These functions can be invoked from any place where there is access to the controller's `$scope` object, even from `view.xhtml` by using appropriate AngularJS directives such as `data-ng-click`.
`defineListeners`	This section is for attaching the listeners to the AngularJS `eventBus` object.
`_handleEvents`	This section is for the event handling functions for every event handler.
`destroy`	This section gets invoked on controller unload. Use this to clean up any used object including event registration.

Deploying gadgets to My webMethods Server

After the gadgets are developed, deploy the portlet application or web application to MWS.

When you deploy a portlet application or web application, the gadgets in the application project are deployed to My webMethods Server, and the deployed gadgets are registered in Business Console.

To manually deploy the gadgets from a portlet or web application to My webMethods Server

1. Package the web applications as a war file.
2. Copy the war file to the directory: `Software AG_directory\MWS\server\server_name\deploy`.

Configuring Global Servers for Gadgets

You can configure global server settings for gadgets which are available for the configuration object.

To configure global server settings for gadgets

1. Log in as a sysadmin into My webMethods Server.
2. Navigate to **<Folders>** > **Administrative Folders** > **Administration Dashboard** > **Configuration** > **CAF Application Runtime Configuration**.
3. Click **Configure Global Defaults** and open **Environment Entries** under web application.
4. Click **Add New Entry** and add the following entries for each of the servers:

Field Name	Type	Description
gadgets.config.servers.host1.serverType	String	The value should be either MWS or IS or AA or other appropriate server type. Each of these acronyms expand to My webMethods Server or Integration Server or AgileApps respectively.
gadgets.config.servers.host1.host	String	localhost (or appropriate host name)
gadgets.config.servers.host1.port	Integer	8585 (or appropriate port)

Field Name	Type	Description
gadgets.config.servers.host1.protocol	String	HTTP (or appropriate protocol)

Viewing gadgets

You can view a gadget by using either the direct URL of the gadget, or by using the gadget in a Business Console dashboard.

Note: You must be logged on in Business Console to view a gadget.

Viewing gadgets in a browser

To view a gadget in a browser

1. Log on to Business Console.
2. Open another browser window.
3. Specify URL of the gadget in the format: `http://Host:Port/business.console.gadgets#/applicationName/gadgetName`.
4. View and test the gadget.

Viewing a gadget in Business Console

To view a gadget in Business Console

1. Log on to Business Console.
2. Create a dashboard. For more information, see the *Working with webMethods Business Console* guide.
3. Add the gadget to the dashboard.
4. Configure the gadget settings.
5. Check the view and behavior of the gadget in the dashboard.

   If you make any further changes to a gadget, publish the updated gadgets to My webMethods Server, and refresh the dashboard to view the gadget changes.

Exporting Gadgets

You can export a gadget from a project to a local file system directory . After exporting, you can import a gadget to another project. For information about importing a gadget, see "Importing Gadgets" on page 17.

To export a gadget in an application project

1. Select the **UI Development perspective** in Software AG Designer.
2. Select the web or portlet application project from where you want to export the gadget.
3. Right-click on a gadget and select **Export**.
4. In the **Export** wizard, navigate to **Select an export destination** > **Software AG** > and select **Export Business Console Gadget**.
5. Click **Next**.
6. In the **Export Business Console Gadget** wizard, provide the following specifications to export the gadget:

Field	Description
Project	Specify the project name of the gadget that you want to export.
Gadget	Specify the gadget name that you want to export.
To directory	Browse and select a folder in which the gadget should be stored.

7. Click **Finish**.

   An archive (.zip) file that contains the exported gadget is generated.

Importing Gadgets

You can import an exported gadget from a local file system directory to any project. For information about exporting a gadget, see "Exporting Gadgets" on page 16.

To import a gadget in an application project

1. Select the **UI Development perspective** in Software AG Designer.
2. Right-click on the web or portlet application project where you want to import the gadget and select **Import**.
3. In the **Import** wizard, navigate to **Select an import source** > **Software AG** > and select **Import Business Console Gadget**.
4. Click **Next**.
5. In the **Import Business Console Gadget** wizard, provide the following specifications to import the gadget:

Field	Description
Project	Specify the project name for the gadget that you want to import.
Gadget Archive	Browse and select the folder where the gadget is located.

6. Click **Finish**.

   The gadget is imported into the application project.

Deleting Gadgets

You can delete a gadget from a local file system directory.

To delete a gadget in an application project

1. Select the **UI Development perspective** in Software AG Designer.
2. Navigate to the **Solutions** tab and right-click on the gadget in the web or portlet application project you want to delete and select **Delete**.

   The gadget is deleted from the application project after you right-click the project again and click **Refresh**.

2 Getting Started

Understanding and developing gadgets

The samples in this section explains how to understand and develop gadgets. You will first create a simple gadget, and then add more features to create a complex gadget.

Creating your first HelloWorld gadget

This section describes how to create and test your first HelloWorld gadget.

Creating MyPortletAppProject application

The first step in creating gadgets for Business Console is to create a portlet application or a web application for the gadgets to reside.

For more information about creating a portlet or web application project by using Composite Application Framework (CAF) in Software AG Designer, see *webMethods CAF and OpenCAF Development Help* guide.

To create MyPortletAppProject portal application project in Software AG Designer

1. In the **UI Development perspective**, select **File > New > Portlet Application Project**.
2. In the application wizard, provide the project name as **MyPortletAppProject**, and click **Finish**. This will create **MyPortletAppProject** application project under which you can create multiple gadgets for Business Console .

Creating HelloWorld gadget

To create a new gadget in an application project

1. Select the **UI Development perspective** in Designer .
2. In **Solutions view**, expand **User Interfaces**, right-click on **MyPortletAppProject** project where you want to create a new gadget, and select **New Business Console Gadget**.
3. In the New Business Console Gadget wizard, provide the following specification for the new gadget. The New Business Console Gadget wizard creates the configuration files and definition file for the new gadget.

Field	Specify
Gadget Type	AngularJS
Gadget Root Directory	gadgets.

Field	Specify
	Note: The gadgets directory will be created directly under MyPortletAppProject project, and will hold the gadget files.
Gadget Name	HelloWorld
Gadget Title	HelloWorld!
Description	My First Gadget
Gadget Group Name	MyGadgets

4. Click **Next**.
5. Click **Finish**.

Creating a view for HelloWorld gadget

To create a view for **HelloWorld** gadget

1. Select the **UI Development perspective** in Designer.
2. In Solutions view, expand **User Interfaces**, right-click on **MyPortletAppProject** project.
3. Open the **view.xhtml** file located under **MyPortletAppProject > gadgets > HelloWord > views**.
4. The `view.xhtml` file of the new HelloWorld gadget will contain only the basic HTML header as shown below.

```
<html>
<h3> HelloWorld Gadget</h3>
</html>
```

5. Add content to the view as shown below.

```
<html>
<h3> HelloWorld Gadget</h3>
    <div>
        Hello World!
    </div>
</html>
```

6. To style the text in the gadget, add a class as shown below.

```
<html>
<h3> HelloWorld Gadget</h3>
    <div class="hello-world">
        Hello World!
    </div>
</html>
```

7. To add styling to the css file:

a. Expand the **styles** directory.

b. Double click on **gadget.scss**.

c. Add the following to `gadget.scss`.

```
.hello-world{
    font-weight:bold;
    color:#ff0000;}
```

Deploying HelloWorld gadget

Publish the **MyPortletAppProject** application in Designer toMy webMethods Server.

When you publish a portlet application or web application, the gadgets in the application project are deployed to My webMethods Server, and the deployed gadgets are registered in Business Console.

To manually deploy the gadgets from an application to My webMethods Server:

1. Package the web application as a `.war` file.
2. Copy the `.war` file to this directory: Software AG_directory\MWS\server\server_name\deploy.

Testing HelloWorld gadget

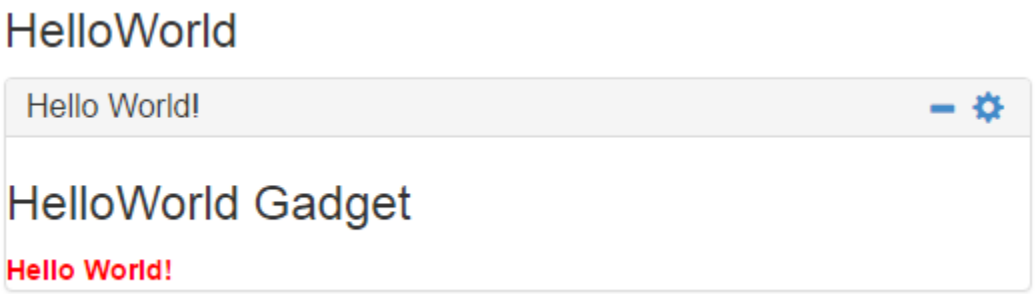
Test the HelloWorld gadget using following URL format:

```
http://<HOST>:<PORT>/business.console.gadgets#/<PROJECT_NAME>/<GADGET_NAME>
```

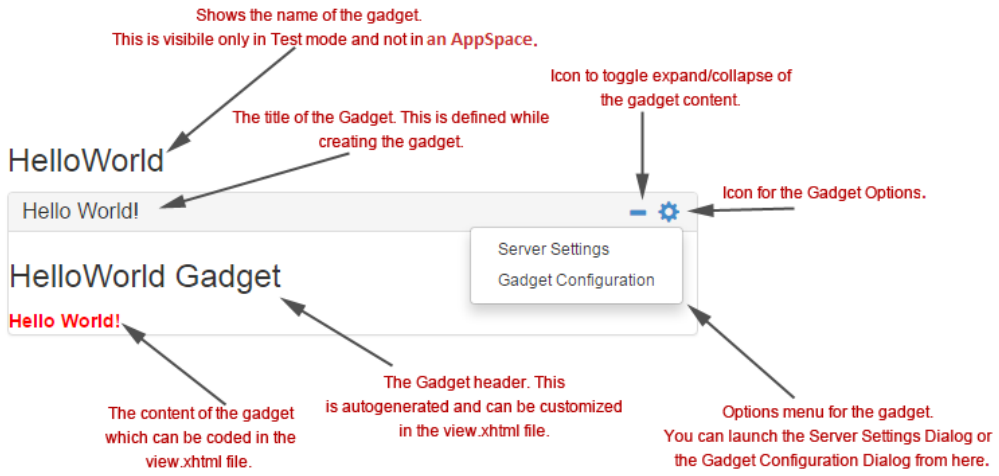
Type the following URL in your browser by replacing `<HOST>` with the host name of My webMethods Server.

```
http://<HOST>:8585/business.console.gadgets#/MyPortletAppProject/HelloWorld.
```

HelloWorld gadget displays as shown below.



The diagram below shows the structure of the gadget.



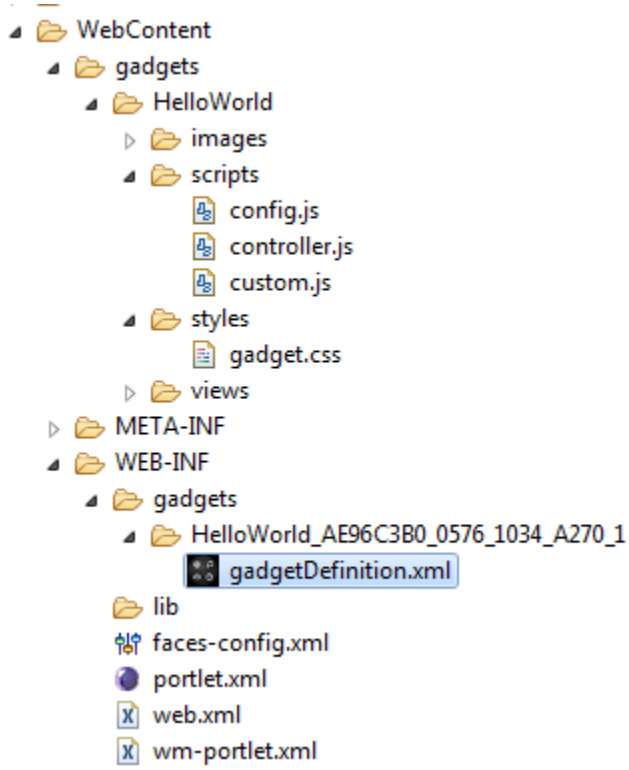
Using RESTful services with gadgets

You can enhance the HelloWorld gadget to display data from My webMethods Server. To do this, you would need to make a REST call to your My webMethods Server. Use an existing RESTful API (`<HOST>:<PORT>/rest`) that shows My webMethods Server node information, and display that information in the gadget.

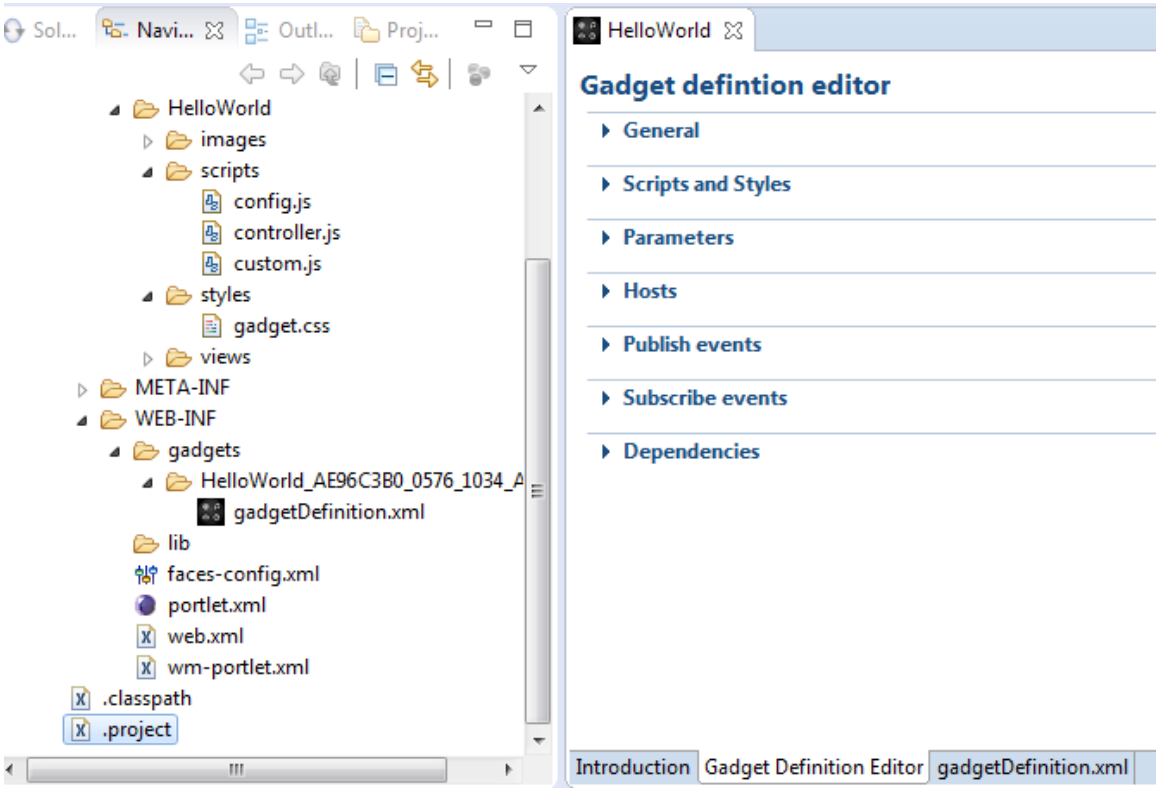
Defining the server

1. Define the My webMethods Server from where the data must be fetched. If you have already defined the My webMethods Server during gadget creation, skip this step.

a. Open the **gadget-deninition.xml** located under **WEB-INF>gadget>Hello_World<ID>**, for example, ID is `HelloWorld_AE96C3B0_0576_1034_A270_1`.



b. On the right pane, click **Gadget Defintion Editor**.



c. Expand the **Hosts** section, and click **Add**.

d. Enter the following:

Field	Fill...
Name	MWS1
Host Name	localhost (or appropriate host name)
Port	8085 (or appropriate port)
Server Type	MWS

e. Click **OK**.

f. Save the server changes.

g. Verify that the server is successfully added by checking the **config.js** located under **MyPortletAppProject > WebContent > gadgets > Hello_World > script** . You will find an auto-generated structure which shows the server details.

```
"config": {
    "params": {
        "servers": {
```

```
            "MWS1": {
                "serverType": "MWS",
                "host": "localhost",
                "port": "8585",
                "protocol": "http"
            }
        },
    },
    "title": ""
}
```

Writing business logic to invoke RESTful services

To write the business logic to invoke the RESTful API

1. Open the **gadgets > HelloWorld > scripts > controller.js** in the editor.
2. Decide when the RESTful service should be invoked. To invoke the RESTful service on gadget load, the call should be made through the `init` block. By default, some RESTful API invocation stubs are auto-generated when a gadget is created. You can enhance the generated stub or create your own.
   a. To invoke the RESTful service on gadget load, add the following code in the `init` section of the gadget's **controller.js** file.

```
init : function($scope, restClient,eventBus,log,config) {
       try{
         .....
            this.$scope.restInvocationCORS(config); //ADD THIS BLOCK IN
            YOUR INIT
         ....
}
```

   b. Replace the `restInvocationCORS` function auto generated under the `defineScope` block with the following code.

```
this.$scope.restInvocationCORS = function(gadgetConfig) {
    var $scope = this;
    var selectedAlias = "MWS1";
    $scope.Math=window.Math;  // Enable the Javascript Math function
    $scope.restClient.url("/rest")  //Provide the server alias to connect
    to
        .serverAlias(selectedAlias)
        .remote(true)
        .cors(true)
        .scope($scope)
        .gadgetConfig(gadgetConfig)
        .success(function(response, $scope) {
            $scope.restData = response;   // The RESPONSE will be
            captured in a variable called restData
        }).error(function(response, $scope, status, headers, config) {
            $scope.eventBus.fireEvent(NotificationConstants.ERROR,
            "Unable to invoke REST " + gadgetConfig.params.
             servers[selectedAlias].host + ":" + gadgetConfig.params.
             servers[selectedAlias].port +
            "/rest for gadget MWS Remote");
        }).invoke();
}
```

Note: The server response is captured in a variable called `restData`. This object is then assigned to the AngularJS `$scope` object. Assigning it to the `$scope` object will make the object available for user interface rendering. A sample response for the RESTful API (`localhost:8585/rest`), is of the following structure:

```
{
    "host": "<HOST>",
    "nodeName": "<HOST>-node<NUMBER>",
    "httpPort": "8585",
    "httpsPort": "0",
    "frontEndUrl": "http://<HOST>:<PORT>",
    "clusterRoles": "[notification, search, taskengine,
     autodeploy]",
    "uptime": "17461.0",
    "freeMemory": "741135632",
    "maxMemory": "954728448"
}
```

Adding user interface code to XHTML

To display information in a table

1. Use HTML table tag to create a table.
2. Add the HTML below to the gadget's `view.xhtml` file.

```
<table class="table table-bordered">
    <thead>
        <tr>
            <th>Host</th>
            <th>Port</th>
            <th>Uptime (sec)</th>
            <th>Free/Max Memory(MB)</th>
        </tr>
    </thead>
    <tr>
        <td>{{restData.host}}</td>
        <td>{{restData.httpPort}}</td>
        <td>{{restData.uptime}}</td>
<td>
    {{Math.round(restData.freeMemory/1000000)}}/{{Math.round(restData.
    maxMemory/1000000)}}
</td>
    </tr>
</table>
```

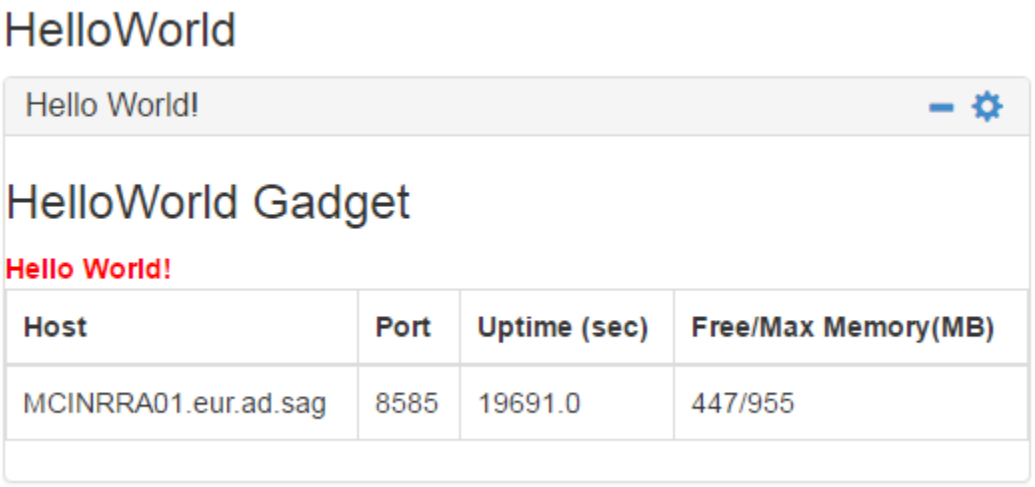
Note: `table`, `table-bordered` from bootstrap enhances the look and feel of the table. You can add more styles to the `gadget.scss` if required.

The `restData` object that was associated with `$scope` object is directly accessible in the user interface. If `restData` is an Array, you can iterate `restData` by using an `data-ng-repeat` tag of AngularJS to populate multiple rows of the table.

Deploying and Testing the Gadget

1. Publish the application to deploy the enhanced gadget.
2. Test the HelloWorld gadget using the following URL format: `http://<HOST>:<PORT>/business.console.gadgets#/<PROJECT_NAME>/<GADGET_NAME>`.

   The enhanced gadget is displayed as shown below.



Offline caching of REST services data

Business Console gadgets access data from the server using the REST services. This mandates network availability and causes network traffic. With the offline caching capability, the user can configure the gadgets to store data offline and access the data from the cache without requesting the server.

Every gadget in the AppSpace has a separate **IndexedDB**, **<Gadget Name>_<Gadget ID>** that can contain multiple offline stores. Every store consists of *Key* and *Value* pairs.

Note: The *Key* should be unique for each entry in the store, otherwise, the *Value* is overridden.

Configuring gadgets to cache data offline

The offline caching capability allows you to configure gadgets to store data offline.

1. Create a gadget. For information about creating a gadget, see "Creating HelloWorld gadget " on page 20.
2. Enable or disable the offline caching capability for gadgets as follows:
   a. Navigate to **gadgetDefinition.xml** of a respective gadget.

b. Select the **Gadget Definition Editor** tab and expand **Parameters**.

c. Click **Add**.

d. Enter the following parameter in the **Name** and **Value** fields respectively:

Name	Value
`Offline`	True

3. Configure offline stores in the **IndexedDB** of gadgets by configuring the REST GET call URL as follows:

Note: You can configure multiple offline stores for a gadget.

a. Navigate to the gadget controller.

b. Define the REST service URL with the caching configuration as follows:

Ensure that the URL is accessible from *$scope.url* .

```
{
          TASK_INBOX_GET:{url:'/rest/pub/opentasksearch', method:'GET'
                          isArray:true, serverType:SERVER_TYPES.TE,
                          headers : {"Content-Type": "application/json"
                         ,"Accept":"application/json"},
                          caching : {name:'OfflineTaskInbox',
                              key : {value:'TaskInbox'},
                         strategy : 'CacheOnly'
                                    }
                         }
       }
```

where:

- *Name* is the unique OfflineStore name of a gadget.
- *Key* is the key value for which the data is stored. *Key* accepts the following variants as values:
  - `key:{value:'TaskInbox'}` is the constant value configured as a key.
  - `key:{requestKeyParamName:'cacheKey'}` key is extracted from the request parameter.
  - `key:{requestKeyPath:'payload.key'}` key is extracted from the request payload using the JSON path configured.
  - `key:{responseKeyPath:'response.id'}` key is extracted from each item in the response array. The configuration for caching is:

```
caching : {name:'OfflineTaskInbox',
                    key:{responseKeyPath:'response.id'},
                  isArray:true
                 }
```

- *Strategy* is the configuration that controls the lifecycle of the cached content. *Strategy* accepts the following variants as values:

  Note: If no value is specified for *Strategy* , then `NetworkFirst` is specified as the default value.

  - `strategy:'CacheOnly'`, where the data is fetched only from the cache. If the data is not found in the cache, then fetch the data from the server and store it in the cache for the first time.
  - `strategy:'CacheFirst'`, where the data is first fetched from the cache and then receives new data from the server and updates the cache.
  - `strategy:'NetworkFirst'`, where the data is always fetched from the server. If the user is offline then the data is fetched from the cache.

Invoking POST calls

To understand how to make POST calls, let's use the Task Engine RESTful service for My webMethods Server to create a task instance of a task type from a gadget. The examples below describe how to accept a task type ID in the gadget, pass the task type ID as POST request, and display the task instance in the gadget.

Adding a POST call in controller

1. Create a HelloWorldTask task type for the task application, and note the taskType ID.
2. In the `defineScope` block in controller, add the code in the `try` block as shown below.

```
defineScope: function(){
           var _this=this;
           ....
          _this.$scope.tasks= new Array();
// TAKING AN ARRAY OF TASK INSTANCES. THIS WILL BE DOUBLE BINDED TO THE UI
           _this.$scope.createTaskInstance=function(){
//FUNCTION TO INVOKE FROM THE UI
               var selectedAlias = "MWS1";
//SELECTED MWS SERVER ALIAS
               var $scope=_this.$scope;
               var requestData = {
//POST CALL REQUEST DATA
                   "taskTypeID":_this.$scope.config.params.taskTypeId,
// REPLACE THIS WITH THE TASK TYPE ID
                   "taskInfo":{
                   "name":"hello world1"
//YOU CAN GIVE ANY NAME TO THE TASK INSTANCE CREATED
                   }
                };
           _this.$scope.restClient.url("/rest/pub/opentask")
//Provide the server alias to connect to
                    .serverAlias(selectedAlias)
                    .method("POST")
```

```
//HTTP METHOD TO BE INVOKED
                    .requestData(requestData)
                    .remote(true)
                    .cors(true)
                    .scope($scope)
                    .gadgetConfig($scope.config)
```

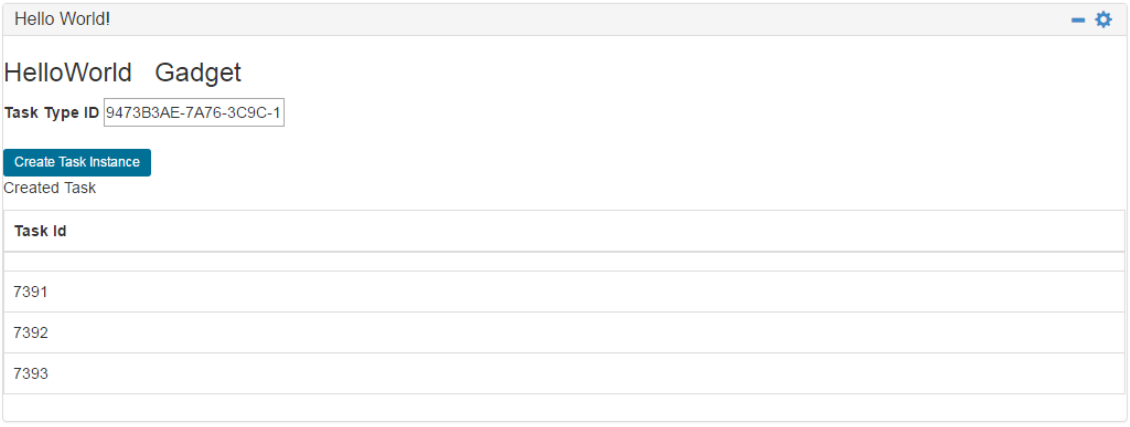
Defining the user interface

1. Open the `view.xhtml` file and add the following code.

```
<html>
<h3> HelloWorld Gadget</h3>
<label>Task Type ID</label>
<input type="text" ng-model="config.params.taskTypeId"></input><br/><br/>
<input class="btn bc-button" type = "button" ng-click="createTaskInstance()"
value="Create Task Instance"></input>
<p>Created Task</p>
<table class="table table-bordered table-responsive" width="100px">
    <thead>
        <tr>
            <th>Task Id</th>
        </tr>
    </thead>
    <tr ng-repeat="taskId in tasks">
        <td>{{taskId}}</td>
    </tr>
</table>
</html>
```

2. Deploy the gadget.
3. Access the gadget using the direct URL for the gadget.
4. Specify the task type ID.
5. Click **Create Task Instance** on the gadget.

   Task instance of the specified task type will be created. The task IDs of the task instances will be displayed in the gadget UI as shown below.



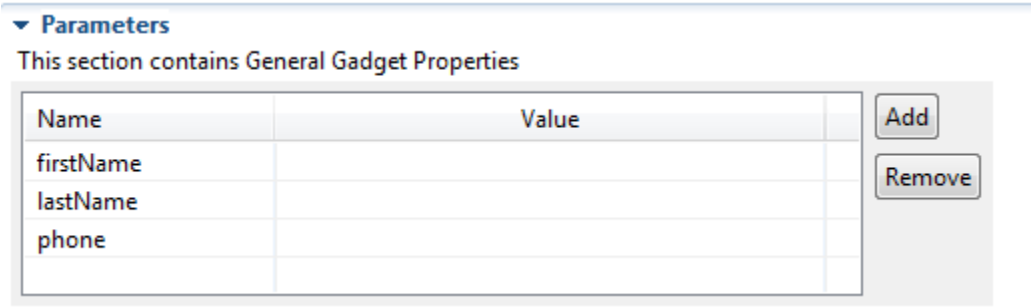
Using forms with gadgets

This section explains how to create a gadget with HTML forms and submit the form data.

Building the gadget

1. Navigate to **WebContent > WEB-INF > gadgets > HelloWorld_<ID> > gadgetDefinition.xml**
2. Select the **Gadget Definition Editor** tab and expand **Parameters**.
3. Click **Add**.
4. Enter the following parameters in **Name** field and leave the **Value** field empty:
   - **firstName**
   - **lastName**
   - **phone**
5. Save the editor.

   The added parameters are displayed as shown below.



Adding HTML user interface code to show form

1. Navigate to `view.xhtml` of the gadget.
2. Add the following HTML tag.

```
<html>
<body>
<h3>HelloWorld Gadget</h3>
...
<div class="container-full">
<form role="form" name="myForm">
    <div class="form-group row">
        <label for="firstName" class="col-md-4">First Name:</label>
        <input type="text" class="col-md-8 remove-paddings" name="firstName"
id="firstName" data-ng-model="config.params.firstName"></input>
    </div>
    <div class="form-group row">
```

```
        <label for="lastName" class="col-md-4">Last Name:</label>
        <input type="text" class="col-md-8 remove-paddings" name="lastName"
id="lastName" data-ng-model="config.params.lastName"></input>
    </div>
    <div class="form-group row">
        <label for="phone" class="col-md-4">Phone:</label>
        <input type="text" class="col-md-8 remove-paddings" name="phone"
id="phone" data-ng-model="config.params.phone"></input>
    </div>
    <input class="btn bc-button row" type="button" value="Submit Form"
onclick="submitForm()"></input>
</form>
</div>
...
</body>
</html>
```

Adding JavaScript code to submit form

To add the JavaScript logic to submit form

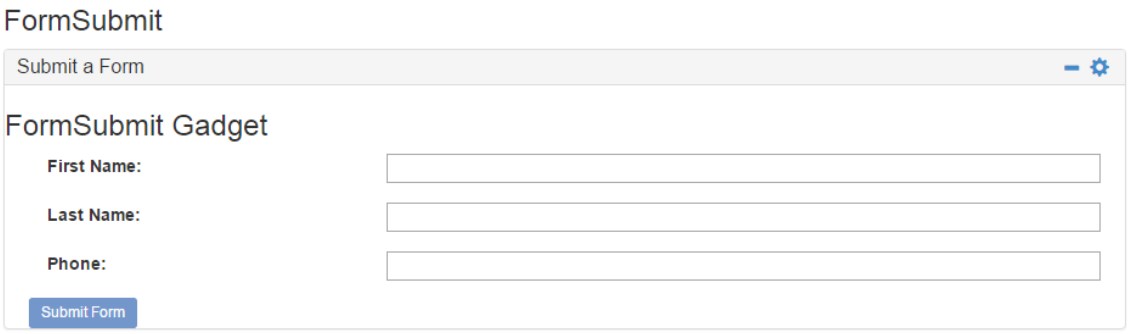
1. Get form values.
2. Construct the URL to submit form.
3. Create `<head>` tag under `<html>` tag.
4. Add the code block given below.

```
<head>
<script>
    function submitForm(){
        var firstName= document.getElementById("firstName").value;
        var lastName= document.getElementById("lastName").value;
        var phone= document.getElementById("phone").value;
        var href ="";
        if(window.location.href.indexOf("?")>0){
            href = window.location.href.substring(0,
            window.location.href.indexOf("?"));
        }else{
            href=  window.location.href;
        }
        var actionUrl = href+"?firstName="+firstName;
        actionUrl =actionUrl+"&amp;lastName="+lastName;
        actionUrl +="&amp;phone="+phone;
        window.location.href=actionUrl;
        window.location.reload();
    }
</script>
</head>
```

Deploying and testing the gadget

1. Publish the application to deploy the updated gadget.
2. Test the gadget using the direct URL: `http://<HOST>:<PORT>/business.console.gadgets#/MyPortletAppProject/HelloWorld`.

   The updated gadget is displayed as shown below.



Display form values in another gadget

1. Create a **FormDisplay** gadget with the following options:
   - **Gadget Name**: FormDisplay
   - **Gadget Title**: Form Display
2. Click **Next** and click **Finish**.
3. Edit the `gadget-defintion.xml` to add the parameters.
4. Enter the following parameters in **Name** field and leave the **Value** field empty:
   - **firstName**
   - **lastName**
   - **phone**
5. Change the `view.xhml` of the FormDisplay gadget to include the following code.

```
<form role="form">
    <div class="form-group">
        <label for="firstName">First Name:</label>
        <label>{{config.params.firstName}}</label>
    </div>
    <div class="form-group">
        <label for="lastName">Last Name:</label>
        <label>{{config.params.lastName}}</label>
    </div>
    <div class="form-group">
        <label for="lname">Phone:</label>
        <label>{{config.params.phone}}</label>
    </div>
</form>
```

6. Deploy the gadgets.
7. To test the gadgets, log on to Business Console using the URL format: `http://<HOST>:<PORT>/business.console`.
8. Click on **Dashboards > Plus** icon.
9. Create a dashboard and add these two gadgets side by side.

10. Click **Submit Form** on the HelloWorld gadget.

    HelloWorld gadget passes the form values to the FormDisplay Gadget through the URL.

Communicating between two gadgets

Communication between gadgets is possible by using a custom JavaScript service called `EventBus` provided by the gadget framework. Each gadget can act as an event subscriber or publisher or both. This section provides basic information for using the `EventBus` service.

To make a gadget trigger events, the `fireEvent` method of `EventBus` is used. The first argument is the Event Name, the second argument is the payload or the data to be passed, and the third argument is an optional context.

```
this.eventBus.fireEvent("SOME_EVENT_NAME", "Some Event!");
```

To make a gadget receive events, define the listener in the `defineListener` block in the controller of the subscribing gadget and then put the handling logic in the `_handleEvents` block.

```
this.eventBus.addEventListener("SOME_EVENT_NAME",this._handleEvents.bind(this));
_handleEvents:function(eventType,payload,context){
            /* Logic to handle events
             */
             switch(eventType){
             case "SOME_EVENT_NAME":
                    /* Add Event Handling Logic for GLOBAL_EVENT */
                    this.$scope.exampleHandleEventAction(payload); //ONCE EVENT
IS RECEIVED , INVOKE THE exampleHandleEventAction function on $scope.
                   break;
             }
      },
```

Implementing communication between gadgets

Create a new HelloWorld2 gadget to trigger events, and pass information from HelloWorld2 gadget to the HelloWorld gadget.

1. Create HelloWorld2 gadget and provide the following to the new gadget.
   - **Gadget Name**: HelloWorld2
   - **Gadget Title**: Hello World 2
2. Open the `view.xhtml` file of HelloWorld2, and add the following code.

   ```
   <input type="button" value="Click to Publish Data" ng-click="publishData()">
   </input>
   ```
3. Open the `controller.js` file of HelloWorld2, and add the following code under the `defineScope` block.

   ```
   defineScope : function() {
       var _this=this;
   ```

```
    this.$scope.publishData= function(){
       _this.eventBus.fireEvent("PUBLISH_DATA", "Hello using Event");
    }
  }
```

4. Open the `view.xhtml` file of HelloWorld, and add the following code.

```
<h3>HelloWorld Gadget</h3>
<div class="hello-world">Hello World!</div>
{{data}}
```

5. Open the `controller.js` file in HelloWorld, and add the following code under `defineListeners` block.

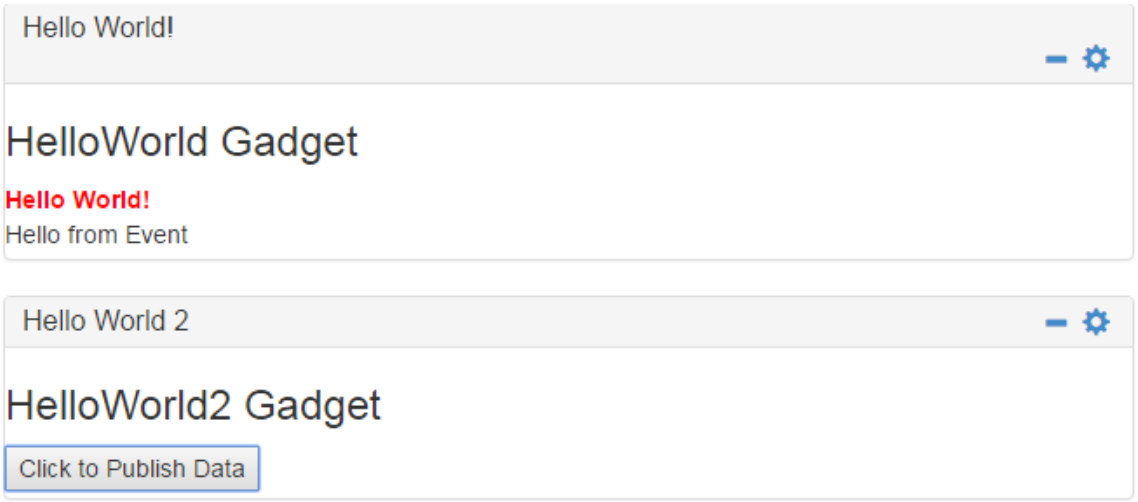
```
defineListeners:function(){
             this._super();
             this.eventBus.addEventListener("PUBLISH_DATA",this.
             _handleEvents.bind(this));
}
```

6. In the same `controller.js` file of HelloWorld, add the following code under `defineScope` block.

```
_handleEvents:function(eventType,payload,context){
              if(eventType=="PUBLISH_DATA"){
                  this.$scope.data=payload;
              }
       },
```

7. Deploy both the gadgets.
8. To test the gadgets, log on to Business Console using URL format: `http://<HOST>:<PORT>/business.console`.
9. Create a dashboard and add the two gadgets side by side.
10. Click **Click to Publish Data** in HelloWorld2 gadget.

    HelloWorld gadget displays Hello using Event text as shown below.



Using third party libraries

If you want a gadget to accept one or more locations as input and plot these locations in a Google Map. The gadget must use Maps API

Google Maps is one of the most popular libraries, which provides a map implementation. However, to use Google Maps in gadget, you need an API Key. You can use a standard API key or use a premium Key if you have Maps license.

To generate your API key, use URL "https://developers.google.com/maps/documentation/javascript/get-api-key". Most of the third party libraries can be downloaded in your local system and then these library files can be added under the scripts folder.

Google Maps restricts the use of maps by downloading. Gadget code must directly refer to Google maps library. This leads to synchronization issue as the gadget has to wait for the library to be loaded from the URL. To achieve this, gadget framework includes a `lazyLoader` (`ocLazyLoader`) module to load all the modules.

Writing user interface code for using third-party libraries in gadget

1. Navigate to the `view.xhtml` file of the gadget.
2. Add the following code:

```
<div class="table">
    <div class="row remove-margins table-row " ng-repeat="item in locations
track by $index">
        <input type="text" ng-model="item.location" class=
"table-cell location-publish-text" placeholder="Enter city, country, zip
 code etc."></input>
        <div style="white-space: nowrap;" class="table-cell">
            <!-- remove button should not be shown if it is a first entry -->
            <button type="button" class="bc-button" data-ng-click="addRow()">
                <i class="fa fa-plus"></i>
            </button>
            <button type="button" class="bc-button"
                data-ng-click="removeRow($index)">
                <i class="fa fa-minus"></i>
            </button>
        </div>
    </div>
    <div class="row remove-margins table-row ">
        <input type="button"
        value="Publish Locations" ng-click="publishLocation()" class=
"bc-button table-cell"></input>
        <input type="button"
        value="Clear" ng-click="clearLocation()" class="bc-button
        table-cell"></input>
    </div>
</div>
<div oc-lazy-load="['js!https://maps.googleapis.com/maps/api/js?key=
 YOUR_API_KEY']">
    <div class="row remove-margins">
        <h4>Maps Gadget</h4>
```

```
        <label class="location-maps-header"> Use this gadget to display
         locations.
        </label>
        <googlemaps id="google_map_canvas" style="height: 760px;">
</googlemaps> <!--CONTAINER FOR THE MAP. CUSTOM DIRECITVE -->
    </div>
</div>
```

a. Replace the `YOUR_API_KEY` with the Maps API key you have retrieved from Google.

b. In the first `<div>`, add a textbox control with plus/minus buttons to add/remove one or more locations.

c. In the second `<div>`, used a tag called `<googlemaps ... />`. This is not an HTML tag, but a custom AngularJS directive. By using this directive, we can get a reference to the container and then place our Map pointers accordingly.

d. This `<div>` also contains a call to Angular lazy loader (`oc-lazy-load`) to load the Maps API from a URL. The custom directive `<googlemaps . . . />` should go inside the `<div>` containing the `oc-lazy-load` attribute.

Styling the gadget

Style the controls before writing the business logic.

1. Navigate to **gadget.scss** file located under **WebContent > Gadget > HelloWorld > styles** of the application project.
2. Add the following code:

```
.location-publish-header{
    font-size: 13px;
    font-family: Arial;
    font-weight: normal;
    padding-left: 8px;
    padding-top: 5px;
}
.table{
    display:table;
}
.table-row{
    display:table-row;
}
.table-cell{
    display:table-cell;
    padding: 3px;
    margin-bottom: 3px;
    margin-left: 5px;
}
.location-publish-text{
    width:100%;
}
```

Writing custom maps directive in custom.js file

To create a custom directive, follow the steps below. You need to replace `<ID>` with the one generated by your gadget.

1. Open **webapp > gadgets > HelloWorld > scripts > config.js** and lookup the gadget module name in the **config.js** file. In the code below, `HelloWorld_<ID>_module` refers to the gadget module.

```
var HelloWorld_<ID>_module = angular.module(..... .
```

2. Add the code below in the `custom.js` file.

```
HelloWorld_<ID>_module.directive('googlemaps', function() {
    return {
        restrict: 'E',
//Restrict to HTML Tags only
        replace: true,
//Replace existing content with the content from this directive
        template: '<div></div>',
//Base Template for the tag
        link: function($scope, element, attrs) {
//This function is invoke when the directive get linked to the DOM
                var center = new google.maps.LatLng(50.1, 14.4);
//Taking a randon centre point in Map
                var map_options = {
// Default Options for Maps
                    zoom : 2,
                    mapTypeId : google.maps.MapTypeId.ROADMAP,
                    center : center,
                };
// Create a Map Object
                var map = new google.maps.Map(document.getElementById
               (attrs.id), map_options);
// Create a Marker Array. Each marker will correspond to a point in the map
                $scope.markers = [];
//Function to be invoke from controller. This takes a an array of location
objects and points each location on the map
// a typical locations array would be something like this
[{"location":"Bangalore"},{"location":"Seattle"},{"location":"Reston"}]
                $scope.locateInMap= function(locations){
                    $scope.hideMarkers();
// Hide existing markers and plot new ones
                    geocoder = new google.maps.Geocoder();
                    for(var key in locations){
                        if(locations.hasOwnProperty(key)){
                            geocoder.geocode({
                                'address' : locations[key].location
// Decode the location string to a specific geocode
                            }, function(results, status) {
                                if (status == google.maps.GeocoderStatus.OK)
                                {
//In this case it creates a marker, but you can get the lat and lng from
the location.LatLng
                                    map.setCenter(results[0].geometry.
                                    location);
                                    var marker = new google.maps.Marker({
                                        map : map,
                                        position : results[0].geometry.
                                        location
```

```
                                    });
                                    $scope.markers.push(marker);
                                } else {
                                    alert("Geocode was not successful for the
following reason: " + status);
                                }
                            });
                        }
                    }
                };
                $scope.hideMarkers=function() {
                    /* Remove All Markers from the Map*/
                    while($scope.markers.length){
                        $scope.markers.pop().setMap(null);
                    }
                }
                window.setTimeout(function() {
                    google.maps.event.trigger(map, 'resize');
                }, 100);
        }
    }
});
```

Coding the gadget controller

Provide appropriate logic in the gadget controller to invoke the `locateInMap` function.

Add the following code to the gadget controller file to receive location from the same gadget and then plot the map.

1. Add an empty location variable and bind it to scope to add a text box in the user interface to capture location.

```
 init:function(){
.....
 this.$scope.locations=[{location:''}];
```

2. Add the following code in the `defineScope` block of the gadget controller to add or remove text box on the user interface when the **+** or **-** icon is clicked:

```
defineScope : function() {
           var _this = this;
           this.$scope.addRow=function(){
                var row={location:''};
                _this.$scope.locations.push(row);
           }
           this.$scope.removeRow=function($index){
               _this.$scope.locations.splice($index, 1);
           }
           this.$scope.clearLocation=function(){
                _this.$scope.hideMarkers();
           }
}
```

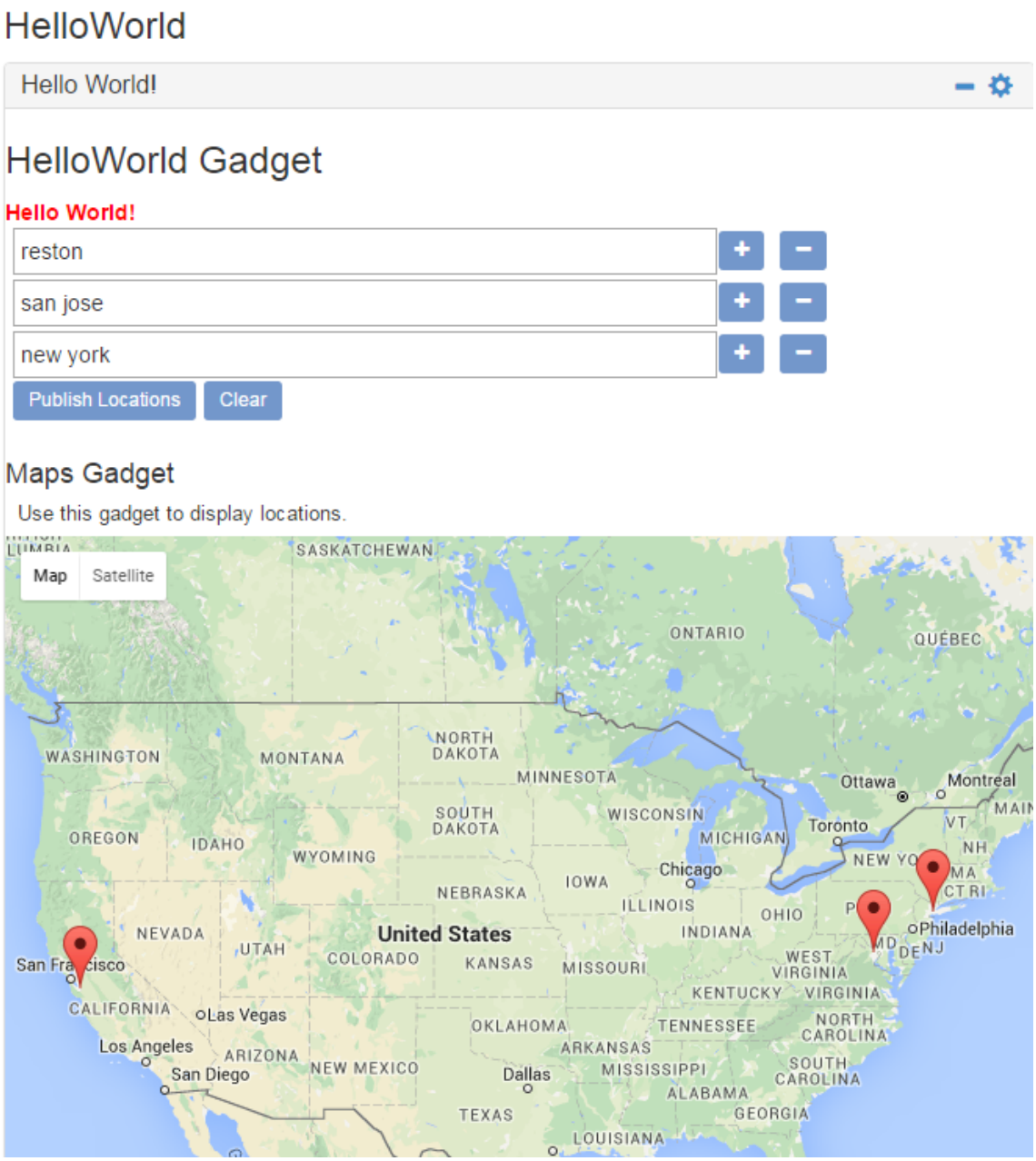
3. Add the following code in the `defineScope` block to plot the locations on the map.

```
defineScope : function() {
     .......
     this.$scope.publishLocation=function(){
             //Check to remove empty locations
             var locations=[];
             for(var key in _this.$scope.locations){
```

```
                if(_this.$scope.locations.hasOwnProperty(key)){
                    if(_this.$scope.locations[key].location!==''){
                        locations.push(_this.$scope.locations[key]);
                    }
                }
            }
            _this.$scope.locateInMap(locations);
        };
}
```

Deploying and testing maps gadgets

1. Deploy the map gadget to My webMethods Server.
2. Test the gadgets using the URL format: `http://<HOST>:8585/business.console.gadgets#/MyPortletAppProject/HelloWorld`.


HelloWorld
Hello World!
HelloWorld Gadget
Hello World!
reston
san jose
new york
Publish Locations
Clear
Maps Gadget
Use this gadget to display locations.
Map
Satellite
SASKATCHEWAN
ONTARIO
QUÉBEC
WASHINGTON
MONTANA
NORTH DAKOTA
MINNESOTA
SOUTH DAKOTA
WISCONSIN
MICHIGAN
Ottawa
Montreal
Toronto
OREGON
IDAHO
WYOMING
NEBRASKA
IOWA
Chicago
ILLINOIS
INDIANA
OHIO
NEVADA
UTAH
COLORADO
KANSAS
MISSOURI
United States
Philadelphia
San Francisco
CALIFORNIA
Las Vegas
Los Angeles
San Diego
ARIZONA
NEW MEXICO
OKLAHOMA
TEXAS
Dallas
ARKANSAS
MISSISSIPPI
TENNESSEE
KENTUCKY
WEST VIRGINIA
VIRGINIA
NORTH CAROLINA
SOUTH CAROLINA
ALABAMA
GEORGIA
LOUISIANA

3 Creating User Interface for Gadgets

Creating user interface

This section explains how to create an user interface for gadgets. You will create responsive gadgets, use form layouts, add contents to the gadgets, and style gadgets.

Using bootstrap components

Gadget framework provides the following files for each gadget for defining the user interface:

- view.xhtml
- settings.xhtml

view.xhtml contains information about the user interface to be displayed when the gadget is rendered on the web page.

settings.xhtml contains information about the user interface for configuring the gadget at run time, if the gadget loading requires any runtime configuration.

view.xhtml and settings.xhtml files of each gadget are individually bootstrap enabled by default. This means that the gadgets are already within a bootstrap container, and the use of Grid System within a gadget is sufficient to make the layout responsive. If you are using AngularJS version of the gadget, you can also use the AngularUI Bootstrap library that are already included as part of the application.

Creating responsive gadgets

Using bootstrap styles for your gadgets would ensure that the gadgets are internally responsive. However, there might be a situation where you need to use responsive large controls in the user interface. For example, HTML tables are not responsive by default, and if you use a big table in a gadget, the table in the gadget might overlap with other gadgets. The best solution for these scenarios would be to use percentages (%) for defining widths.

Using form layouts

If you need to use a form in a gadget, you can use the `form-horizontal` class to style the form controls.

Note: Tooltips can be specified using the `data-hint` attribute, and the orientation can be controlled using `hint--top`, `hint--bottom`, `hint--left`, and `hint--right` classes.

A typical example of a form is shown below:

```
<form class="form-horizontal">
    <div class="control-group">
        <label class="control-label hint--top" data-hint="Tooltip Message1"
for="">Control-1</label>
        <div class="controls">
            <<Add the form control>>
        </div>
    </div>
    <div class="control-group">
        <label class="control-label hint--top" data-hint="Tooltip Message2"
for="">Control-2</label>
        <div class="controls">
            <<Add the form control>>
        </div>
    </div>
</form>
```

Adding static or dynamic content

You can add your static content to the view.xhtml file.

You can build dynamic content in the view.xhtml file by using one of these methods:

- Invoking services written in controller.js
- Importing external Javascript libraries and using the function in the Javascript libraries

Note: For importing JavaScript library, use `<div oc-lazy-load="jsFileName.js"/>` tag instead of the `<script import="jsFileName.js"/>` tag. The normal `<script import="jsFileName.js"/>` tag will not work. You should not import the internally generated.js JavaScript files such as custom.js that are automatically available in the controller.js and view.xhtml files.

Styling gadgets

For each gadget, you can provide your custom styles in the gadget.scss file of the gadget. You also have the option to provide the style in-line in view.xhtml or settings.xhtml files under a `<style>` tag.

Note: From Designer 10.0 or later, Sass (Syntactically Awesome StyleSheets) is used as part of the gadget styles. Sass is an extension of CSS that adds power and elegance to the basic language. The file extension for gadgets using Sass is .scss.

Adding styles in CSS

You can add styles for a gadget within a namespace in the gadget.scss file as follows:

1. Navigate to the **gadget.scss** file under the **styles** folder of a respective gadget in the application project.
2. Include your styles in the following code:

Note: `.gadget-container-<Gadget Name>-<Short Identifier>` is the auto-generated namespace.

```
.gadget-container-<Gadget Name>-<Short Identifier> {
        .gadget-header-text{
         font-size: 20px;
         padding: 10px 10px 0px 10px;
         color: red
}
// write your styles here
}
```

Enabling CSS Editor for .scss files in Designer

To open a `.scss` file in Designer:

1. Launch the Software AG Designer in the UI perspective.
2. Navigate to the **gadget.scss** file under the **styles** folder of a respective gadget in the application project.
3. Right-click **custom.scss** and select **Open With > CSS Editor**.
4. In the **Unsupported Content Type** wizard, click **Content Types Preferences Page** link.
5. In the **Preferences** wizard, navigate to **General > Content Types** and select the **CSS** content type and click **Add**.
6. In the **Add Content Type Association** wizard, type ***.scss** in the **Content type** field.
7. Click **OK**.

Note: This is one-time configuration that you need to perform in your workspace.

Embedding a Gadget within another Gadget

You can embed a gadget inside another gadget by editing the `bc-gadget tag` as follows:

```
<bc-gadget widget-model="<GADGET_ID>" params="{ key1:value1,
key2:value2 }" embed="true"> </bc-gadget>
```

where:

- *GADGET_ID* is the unique identifier of the gadget.
- *key* is the parameter key.
- *value* is the parameter value.

4 Programming Gadgets

About programming gadgets

The business logic for programming the gadgets are defined in a base controller. The AngularJS provides a framework to manage the business logic of the gadgets. This section gives information about the functions and module dependencies defined in the controller file associated to gadget module.

Base controller for programming gadgets

AngularJS provides a MVC framework to manage the business logic of a gadget. The base logic for programming a gadget should reside in the `controller.js`. The gadget framework binds the controller to the view file dynamically at runtime so that you do not have to mention the controller explicitly in the view using the `data-ng-controller` attribute.

In the gadget development framework, we have included a base controller (`BaseController.js`) which wires some of the required services to the controller. As a best practice, all AngularJS controller for a gadget should extend base controller. Though JavaScript does not provide inheritance directly, you can use JavaScript prototype inheritance.

Extending base controller provides a way to overload some of the functions defined in it. Functions in base controller provide a structured coding approach.

Functions defined in base controller

init

The `init` function can be considered as the constructor function of the controller class. `init` function is invoked when the gadget is loading into the dashboard. Gadget loading occurs when a gadget is added to a dashboard or when an existing dashboard is launched. All invocations or business logic that are required for loading a gadget must be included in the `init` function. The `init` function arguments must match the injected components in the controller. For example, code as shown below.

```
....
 init : function($scope, restClient,eventBus,log,config)
 {
 ....
 }
....
ExampleController.$inject = [ '$scope', 'RestServiceProvider','EventBus','$log',
'config'];
```

The `ExampleController` controller is injected with `$scope`, `RestServiceProvider`, `EventBus`, `$log`, and `$config` objects. These are the services that the gadget framework exposes for various business logic. The order of the injection must match the order of the

arguments in the `init` function to receive the injected object in the `init` function. Once the objects are injected in the `init` function, specify a call as shown below.

```
this._super($scope,eventBus,restClient);
```

This call registers the `$scope`, `eventBus`, and `restClient` objects with the BaseController.

defineScope

All functions required to be defined on the controller scope is defined here. The scope functions are the main business logic of the gadget that are related to the user interface. Scope functions can be invoked in the `init` function (after `_super` call) or through any other flow (for example, on event handling).

As a best practice, all functions that are related to the user interface such as a button click should be defined under the `defineScope` block. Since scope is an `AngularJS` object accessibly locally within a gadget, any function defined the `defineScope` block makes the function available for invocation from the user interface (for example, using `ng-click=<function name>`) or from any other place that has access to the controller scope. However, generic business logic that is not tied to the user interface should not be made a part of the controller and should be added to an AngularJS service or factory. This allows the code, to be injectable as well as unit testable.

defineListeners

This function block is invoked once during controller load, and is used to define all the listeners on the `EventBus` that are related to the controller. Event listeners are fined as shown below.

```
this.eventBus.addEventListener("EVENT_TYPE_NAME", this._handleEvents.bind(this));
```

The code above will register the controller as a listener for the type of event mentioned in the first argument of `addEventListener`. For every listener added to the controller, a respective handling block should be provided under the `_handleEvents` function block.

destroy

The destroy function is synonymous to the `defineListeners` block mentioned above.

Defining module dependencies

All AngularJS services, directives, and factories defined in controller.js or custom.js are modularized for better code management as specified below.

1. Define a module for each angular directive, factory, or service as shown below.

   ```
   var module_name = angular.module('MODULE_NAME',[DEPENDENCIES]);
   ```

2. Provide a dependency for the module in the gadget. This can be done using Software AG Designer while creating the gadget or later while using the gadget definition editor.

3. Inject the module, services, or factories in the controller (in the `$inject` code) and use it appropriately.
4. In case of providers:

```
angular.module('MODULE_NAME',[DEPENDENCIES]).provider('PROVIDER_NAME',
function PROVIDER_FUNCTION(){
    this.$get =  [<INJECTABLES>,function (<INJECTED_OBJECTS>){
           this.instance={};
           this.instance.<providerFunction>= new function(){
              //INJECTED OBJECTS CAN BE SET ON THE INSTANCE OBJECT
           };
           return this.instance;
    }]
});
```

5. In case of factories:

```
angular.module('MODULE_NAME',[DEPENDENCIES]).factory(<INJECTABLES>,
[<INJECTED_OBJECTS>, function <FactoryFunction>() {
  ....
  return <OBJECT>;
}]);
```

6. In case of services:

```
angular.module('MODULE_NAME',[DEPENDENCIES]).service('<SERVICE_NAME',
[<INJECTABLES>, new ServiceFunction(<INJECTED_OJECTS>){
   //Service Code
}]);
In case of Directives,
module_name.directive('<DIRECTIVE_NAME>', function () {
    return {
         restrict: 'E, A, C',
         link: function ($scope, element, attrs, controller) {
            //Directive code goes here
         }
    };
});
```

An alternative way to create directive, services, or factories is by using modules to extend the classes function, and use the function name to link them. For example, see the directive below.

```
var ExampleDirective= Classes.extend({
    $scope : null,
    $attrs:null,
    $controller:null,
    /**
     * Initialize Directive
     *
     * @param $scope,
     *            current  scope
     */
    init : function(scope,element, attrs, controller,) {
    },
    /**
     * Initialize listeners needs to be overrided by the subclass.
       Don't forget
     * to call _super() to activate
     */
    defineListeners : function() {
        his.$scope.$on('$destroy', this.destroy.bind(this));
```

```
    },
    /**
     * Use this function to define all scope objects. Give a way to
       instantaly
     * view whats available publicly on the scope.
     */
    defineScope : function() {
    },
});
angular.module('MODULE_NAME',[DEPENDENCIES]).directive('<DIRECTIVE_NAME>',
function () {
    return {
         restrict: 'E, A, C',
         link: function ($scope, element, attrs, controller) {
            //Directive code goes here
            return new ExampleDirective();
         }
    };
});
```

Injecting services, factories, and providers

As mentioned above all AngularJS services, factories, and providers should be created as AngularJS modules, and then associated to the gadget module. Each gadget is by default created as an AngularJS module, and the dependencies should be set to allow the services to be injected in the controller. To inject any custom object use the `$inject`.

```
<CONTROLLER_FUNCTION>.$inject = [ '$scope', 'RestServiceProvider','EventBus',
'$log','config'...];
```

Defining Angular $scope object

Each controller for the gadget is injected with AngularJS `$scope` object in the `init` function and then passed over to other functions such as `defineScope` and `defineListeners`. For functions that are defined on the scope, for example, `$scope.functionName=function(){}`, the scope object is accessible within the function using `this` operator in JavaScript. However, the value of `this` changes dynamically depends on how the function is invoked.

To get an instance of the `$scope` effectively inside a scope function, the `$scope` object needs to be assigned to `this` operator, and then the object can be used throughout the function.

```
this.$scope.restInvocation = function(gadgetConfig){
      var $scope = this;
//depending on how the restInvocation function is invoked,
//'this' object inside the function will have an instance of the $scope object
....
}
```

Invoking RESTful services

The gadget framework provides an AngularJS provider called RestServiceProvider, which provides several ways to invoke RESTful services.

Steps to invoke RESTful services

1. In the `URLS` object, define the URL required for invoking the RESTful service.

```
URLS:{
             MY_REST_SERVICE1:{url: '/rest/rs/myRest1',method:'GET',
             isArray:true},
             MY_REST_SERVICE2:{url: '/rest/rs/myRest2',method:'GET',
             isArray:true}
      },
```

2. Define a function (for example, `invokeMyRestFunction`) inside the `defineScope` block that makes the actual invocation.

```
this.$scope.invokeMyRestFunction= function(gadgetConfig) {
                var $scope = this;
                $scope.restClient.url($scope.URLS.MY_REST_SERVICE1)
//POINT TO THE RESTful SERVICE TO INVOKE
                 .serverAlias("IS1")
// POINT TO THE SERVER ALIAS FROM THE GADGET CONFIGURATION
                 .remote(true)
// IF LOCAL OR REMOTE CALL
                 .cors(true)
// IF CORS SUPPORTED FOR REMOTE CALLS ONLY
                 .scope($scope)
                 .gadgetConfig(gadgetConfig)
                 .success(function(response,$scope){
                     $scope.responseData =  response;
// HANDLE THE RESPONSE IN A SCOPE OBJECT
                 }).error(function(response,$scope,status, headers, config){
                     $scope.eventBus.fireEvent(NotificationConstants.ERROR,
"Unable to invoke REST ");// HANDLE ANY ERROR IN INVOCATION
                 }).invoke();
            }
```

   There are two methods to invoke a RESTful service using URLs:

Method	Description
CORS support for RESTful services	This method invokes the RESTful services using direct URL. For more information about Cross Origin Resource Sharing (CORS), see "Using CORS support for invoking RESTful services " on page 59
Business Console proxy for RESTful services	This method invokes the RESTful services using Business Console proxy, in case the remote server cannot be configured to support the

Method	Description
	CORS headers. For more information about Business Console proxy, see "Using Business Console proxy for invoking RESTful services " on page 60

3. Call the `invokeMyRestFunction` function from `init` (if the RESTful service needs to be called on Gadget load) or from any other place depending on the business logic.

```
init : function($scope, restClient,eventBus,log,config) {
            ...
            this.$scope.invokeMyRestFunction(config);
            ...
      },
```

As a part of the gadget frame work, there are two methods to invoke a RESTful service:

Method	Description
Builder Style pattern RESTful service invocation	This method provides various parameters to the `restClient` object to invoke RESTful services. For more information about the parameters for Builder Style pattern, see "Builder style pattern for invoking RESTful services" on page 55
Traditional RESTful service invocation	This method uses the `invokeREST` function to invoke RESTful services. For more information about the `invokeREST` function, see "Traditional service for invoking RESTful services" on page 57

Builder style pattern for invoking RESTful services

In this method, you can provide various parameters to the `restClient` object, and use the invoke method.

Following are the parameters for Builder Style pattern:

Parameter	Description
`url`	(REQUIRED) This is the relative URL for RESTful service invocation. The `serverAlias` parameter picks the server to be connected. In the gadget definition, you must define a list of servers (alias) that this gadget must invoke.

Parameter	Description
`method`	(OPTIONAL) This can be `GET`, `POST`, `PUT`, `DELETE`, or any other HTTP Request Method. Default method is `GET`.
`requestData`	(OPTIONAL) This can be `GET`, `POST`, `PUT`, `DELETE`, or any other HTTP Request Method. Default method is `GET`.
`serverAlias`	(REQUIRED) This is the alias of the server the gadget must connect to. The list of servers and their alias must be defined in the gadget creation phase or by editing the gadget definition xml file.
`remote`	(REQUIRED) If this parameter is set to `false`, then the invocation will always go to the local MWS server. If it is set to `true`, then the remote server will be evaluated based on the `serverAlias` provided.
`cors`	(OPTIONAL) If this parameter is set to `true`, then a Cross Origin Request will be send to the remote server considering that the CORS headers are already set to allow the request to execute successfully. If it is set to `false`, then the request will be routed through a Business Console proxy URL to avoid Cross Origin Requests. The default value is `true`.
`scope`	(REQUIRED) This is the scope of a gadget under AngularJS context. This will be passed back as part of the `success callback` function so that further actions can be taken.
`gadgetConfig`	(REQUIRED) The configuration of the gadget that is passed to the controller is set here.
`success`	(REQUIRED) The response for the successful invocations will be send to the `success callback` function. The arguments passed are: ■ `response`: The `response` object (JSON) ■ `$scope`: The `$scope` object associated with the gadget controller

Parameter	Description
`failure`	(REQUIRED) The response for the failed invocations will be send to `error callback` function. The arguments passed are: ■ `response`: The `response` object from the invocation ■ `$scope`: The `$scope` object associated with the gadget controller ■ `status`: HTTP status code of the response ■ `headers`: `{function([headerName])}` function retrieves the `header` object ■ `config`: The configuration object used to generate the request

`invoke`	This function must be invoked at the end after passing all required parameters.
	`$scope.restClient.url($scope.URLS.MY_REST_ SERVICE1)`
	`//POINT TO THE RESTful SERVICE TO INVOKE`
	`.serverAlias("IS1")`
	`// POINT TO THE SERVER ALIAS FROM THE GADGET CONFIGURATION`
	`.remote(true)`
	`// IF LOCAL OR REMOTE CALL`
	`.cors(true)`
	`// IF CORS SUPPORTED FOR REMOTE CALLS ONLY`
	`.scope($scope)`
	`.gadgetConfig(gadgetConfig)`
	`.success(function(response, $scope){`
	`$scope.responseData = response;`
	`// HANDLE THE RESPONSE IN A SCOPE OBJECT`
	`}).error(function(response, $scope,status, headers, config)`
	`{`
	`$scope.eventBus.fireEvent(NotificationConstants.ERROR,`
	`"Unable to invoke REST ");`
	`// HANDLE ANY ERROR IN INVOCATION`
	`}).invoke();`

Traditional service for invoking RESTful services

You can use the `invokeREST` function defined under the `RestService` Angular service to invoke RESTful services.

invokeREST function signature:

```
invokeREST:function(URLobj,successCallback,errorCallback,parameters,data,
scope, pathParams, gadgetConfig, isCrossOriginRequest, serverAlias)
```

invokeREST function arguments:

- `URL job`: The URL object should point to a local URL object created under your controller. This can be defined under controller.

```
REST_URLS_OBJECT = {
          REST_SVC_1: {url: '/rest/svc1',method:'GET', isArray:true},
          REST_SVC_2: {url: '/rest/svc2',method:'GET', isArray:true},
}
```

  For example, to use `REST_SVC_1, use REST_URLS_OBJECT.REST_SVC_1` as your `URLobj`.

  Note: All URLs defined here must be relative URLs without the `host:port` information. The `host:port` information will be fetched based on the `serverAlias` argument of the function call.

- `successCallback`:

◦ **Success call back function signature:**

```
var successFunc = function(response, status, headers, config,scope,
gadgetConfig){
// YOU SUCCESS HANDLER CODE GOES HERE
}
```

  - Success call back function arguments:
    - `response`: Function invocation success response
    - `status`: HTTP status code of the response
    - `headers`: `{function([headerName])}` to retrieve the header object
    - `config`: The configuration object used to generate the request
    - `scope`: The `$scope` object associated with the gadget controller
    - `gadgetConfig`: The gadget configuration object. It contains the server list and any optional parameters that the gadget has been configured with

- `errorCallback`: This is the error callback function where the response will be passed when the `invokeREST` invocation fails. Arguments passed arguments to the `error callback` function:
  - `response`: Error response from the invocation
  - `status`: HTTP status code of the response
  - `headers`: `{function([headerName])}`. This can be a function to retrieve the header object
  - `config`: The configuration object used to generate the request

- `scope`: The `$scope` object associated with the gadget controller
- `gadgetConfig`: The gadget configuration object. It contains the server list and any optional parameters that the gadget has been configured with

- `Parameters`: JavaScript object to build the query parameters. Build the query as follows:

```
var parameters = new Array();
var param1 = new Object();
param1.name = "key1";
param1.value = value1;
parameters.push(param1);
param2.name = "key2";
param2.value = value2;
parameters.push(param2);
```

`Query built: ?key1=value1&key2=value2`

- `Data`: Required in case of POST and PUT calls. The request data object that can be passed to the server. It can be a String or a JSON object.
- `Scope`: The scope of the gadget in the AngularJS context. This will be passed back as part of the `success callback` function so that further actions can be taken
- `pathParams`: The path parameters that are appended to the URL string. If `pathParams` is a string, then it is directly appended to the end of the URL prior to the query parameters. If it is an Array, then the params string is constructed as follows:

```
var pathParams= new Array();
pathParams.push("param1");
pathParams.push("param2");
pathParams.push("param3");
```

URL constructed: `URL/param1/param2/param2?<Query Param>`

- `gadgetConfig`: The configuration of the gadget contains the server list and any optional parameters that the gadget has been configured with.
- `isCrossOriginRequest`: Set it to true in case of a Cross Origin Request to a server supporting CORS headers. Otherwise, set it to false to use a proxy invocation to remote server.
- `serverAlias`: The alias of the server to make the call to. The list of servers should be defined in the gadget configuration file and the selected alias should be passed here.

Using CORS support for invoking RESTful services

Business Console gadget supports direct invocation of URLs to a remote server, if the remote server supports Cross-Origin Resource Sharing (CORS). To make a cross-origin request, ensure that the remote server is configured with all the CORS settings.

For example, In Integration Server, provide the following in `Extended Settings` to support CORS headers.

```
watt.server.cors.allowedOrigins=http://localhost:8585, (Please specify the URLs
from where the invocation is happening)
watt.server.cors.enabled=true
```

```
watt.server.cors.exposedHeaders=Set-Cookie,X-Frame-Options,Access-Control
-Allow-Origin
watt.server.cors.maxAge=1000000
watt.server.cors.supportedHeaders=samlassertion,accept,
withcredentials,content-type
watt.server.cors.supportedMethods=GET,POST,PUT,DELETE,OPTIONS,HEAD
watt.server.cors.supportsCredentials=true
```

After configuring CORS settings, you can invoke a RESTful service with `cors` (true) and `remote` (true) options.

```
$scope.restClient.url($scope.URLS.MY_REST_SERVICE1)
// POINT TO THE RESTful SERVICE TO INVOKE
                .serverAlias("IS1")
// POINT TO THE SERVER ALIAS FROM THE GADGET CONFIGURATION
                .remote(true)
// IF LOCAL OR REMOTE CALL
                .cors(true)
// IF CORS SUPPORTED FOR REMOTE CALLS ONLY
                .scope($scope)
                .gadgetConfig(gadgetConfig)
                .success(function(response,$scope){
                    $scope.responseData =  response;
// HANDLE THE RESPONSE IN A SCOPE OBJECT
                }).error(function(response,$scope,status, headers, config){
                 $scope.eventBus.fireEvent(NotificationConstants.ERROR,
"Unable to invoke REST ");
// HANDLE ANY ERROR IN INVOCATION
                }).invoke();
```

Using Business Console proxy for invoking RESTful services

In case the remote server cannot be configured to support CORS headers, Business Console provides a proxy service to route the request through the host My webMethods Server.

To use this option, you need to make a local RESTful POST call to MWS using `/rest/bc/proxy` URL. Provide the server options for fetching the data in the POST body. Provide data in a JSON format with escaped quotes.

```
$scope.restClient.url($scope.URLS.BC_PROXY)
//POINT TO THE RESTful SERVICE TO INVOKE
                .serverAlias("MWS1")
// POINT TO THE SERVER ALIAS FROM THE GADGET CONFIGURATION
                .remote(false)
// IF LOCAL OR REMOTE CALL
                .method("POST")
                .scope($scope)
                .requestData(data)
//JSON Structure of the Request data. See below
                .gadgetConfig(gadgetConfig)
                .success(function(response,$scope){
                    $scope.responseData =  response;
// HANDLE THE RESPONSE IN A SCOPE OBJECT
                }).error(function(response,$scope,status, headers, config){
                    $scope.eventBus.fireEvent(NotificationConstants.ERROR,
"Unable to invoke RESTful service");
// HANDLE ANY ERROR IN INVOCATION
                }).invoke();
Following are the example of different invocations
GET CALL TO IS
```

```
var data = { "serverType":"IS",
"url":"/rest/rs/monitor/process/model",
"requestMethod":"GET"
, "requestHeaders:{'key1':'value1','key2':'value2'}"
}
```

POST call to IS

```
var data =  {"serverType":"IS",
"url":"/rest/rs/monitor/process/instanceSearch",
"host":"localhost",
"port":"5555",
"protocol":"http",
"requestMethod":"POST",
"data":"{\"instanceSearchQuery\":{\"processKey\":\"FeatureProject/Feature\",
\"pageNumber\":1,\"pageSize\":10,\"status\":\"2\",\"instanceId\":null,
\"customId\":null,\"businessConsoleRequest\":true}}"
"requestHeaders:{'key1':'value1','key2':'value2'}"
}
```

GET call to remote MWS

```
var data =  { "serverType":"MWS",
"url":"/rest/bc/userpreferences",
"host":"localhost",
"port":"8585",
"requestMethod":"GET",
"protocol":"http" (optional)
"requestHeaders:{'key1':'value1','key2':'value2'}"
}
```

Generating REST Connector Code for REST Services

You can generate UI elements by dragging and dropping the Rest API Descriptor to a gadget project. This capability automatically generates the REST connector code for the associated REST services. You can create a REST client UI which is a Business Console gadget. Currently, this use case supports only the REST APIs from Integration Service that have the REST API Descriptors (RAD) defined for them. The REST Connector codes are generated based on the signatures defined by the REST API Descriptors. See Working with REST API Descriptors in the *webMethods Service Development Help* for more information.

Drag and Drop Existing REST Resources to Generate the Gadget UI

The UI elements are generated by reading Integration Server REST API Descriptors. After the REST API Descriptor code is generated in the gadget, you need to deploy it on My webMethods Server to make it available on Business Console.

To drag and drop existing REST resources to generate the gadget UI

1. Launch Software AG Designer in the UI perspective.
2. In the **Solutions** tab, create a new portlet or web application.
3. Under the newly created portlet project, create a new Business Console Gadget.

4. In the **Package Navigator** tab, create or identify the REST resource services in the project.
5. Create a REST API Descriptor and associate the identified REST resource services with the REST API Descriptor.
6. Specify the REST API details.
7. Drag and drop the REST API Descriptor you created, into the `view.xhtml` page of the newly created portlet project.

   The **New REST Connector** wizard opens.
8. In the **New REST Connector** wizard, verify the gadget project details, and click **Next**.
9. Select the REST resource service and a method such as GET, POST, PUT, or DELETE and click **Finish**.
10. Type a name in the **Page Name** field.

    This name appears as form name on the web UI.
11. Select the services and fields for which you want to create the REST UI connector code. Subsequently, configure the fields and click **Next**.

REST resource type	Actions you can perform
GET	Specify the following information: ■ **Input Parameters**. You can modify the **Display Label**, **Mandatory**, **Validation**, and **UI Control** fields. The **Input Type** field cannot be modified. Only flat fields are accepted for GET; complex types are not supported. ■ **Output Parameters**. You can modify the **Display Label**, **Type**, and **UI Control** fields. The **Input Type** field can be modified only for String Array type. It can be set either as **StringArray** or **StringTable**. If the type is a StringTable type on the Integration Server, this has to be selected as StringTable if it has to work properly. Else it is treated as StringArray by default. ■ Select the column count for IN and OUT parameters. ■ Set pagination details by selecting **Page Number**, **Page Size**, **Start Index**, **End Index**, and **Total Records**. Pagination is applicable only for array types. The parameters shown in the list includes only path and query parameters. The parameter list for **Total Records** column includes all the output parameters of type text and numeric values.

REST resource type	Actions you can perform
POST	Specify the following information: ■ **Input Parameters**. You can modify the **Display Label**, **Mandatory**, **Validation**, and **UI Control** fields. Input type cannot be modified. ■ **Output Parameters**. You can modify the **Display Label**, **Type**, and **UI Control** fields. **Type** can be modified only for String Array type. It can be set either as **StringArray** or **StringTable**. ■ Select the column count for IN and OUT parameters. ■ Set pagination details by selecting **Parameter Details**, **Page Size**, and **Total Records**.
PUT	Specify the following information: • **Input Parameters**. You can modify the **Display Label**, **Mandatory**, **Validation**, and **UI Control** fields. The **Type** field cannot be modified. ■ **Output Parameters**. You can modify the **Display Label**, **Type**, and **UI Control** fields. Type can be modified only for String Array type. It can be set either as **StringArray** or **StringTable**. ■ Select the column count for IN and OUT parameters. ■ Set pagination details by selecting **Parameter Details**, **Page Size**, and **Total Records**.
DELETE	Specify the following information: ■ **Input Parameters**. You can modify the **Display Label**, **Mandatory**, **Validation**, and **UI Control** fields. ■ Select the column count for IN and OUT parameters. ■ Set pagination details by selecting **Parameter Details**, **Type**, **Page Number**, **Page Size**, and **Total Records**.

12. Select the number of input and output parameters for the selected REST service and click **Finish**.

    Two new files namely `partial.xhtml` and `directive.js` are created. The files `view.xhtml`, `config.js`, and `controller.js` are updated after the drag and drop operation.

    The partial xhtml file name is of the format `<operation>-<servicename>-<shorted>.xhtml`. The operation is the REST operation type (GET, POST, PUT or DELETE) that is selected. Service name is the complete namespace of the REST

service. Short ID is a random alpha-numeric ID of 6 character length. The directive.js is of the format `<operation>_<servicename>_<shorted>.js`.

13. Make the desired changes to the gadget using Software AG Designer and deploy it to My webMethods Server.

    See "Deploy gadgets to My webMethods Server" on page 14 for instructions on deploying the modified gadget in the My webMethods Server.

14. Alternatively to deploy the modified project in My webMethods Server, in the **Server** tab of Designer, right-click on the My webMethods Server instance and select **Add and Remove**. The **Add and Remove** dialog box appears.

15. Move the project from the available list to the configured list and click **Finish**.

    This ensures that the gadget is deployed on My webMethods Server. See "Add/View Gadgets" on page 61 for more information on adding the modified AgileApps form as a gadget in Business Console.

16. Ensure that the same version of the modified gadget is deployed on the My webMethods Server and imported as a gadget in an AppSpace on Business Console.

Including independent AngularJS modules in gadgets

All business console gadgets are defined as independent AngularJS modules.

To customise designs for services, factories, and directories in gadgets as independent modules, do the following:

1. Define services, factories, and directives as independent modules in `custom.js`.
2. Set the dependency for the independent modules.

   For example, in `custom.js`, define a directive as a module.

   ```
   angular.module("MY_GADGET_DIRECTIVE_MODULE", []).directive("myDirective",
   function(){
   // ADD DIRECTIVE CODE HERE
   })
   ```

   The `myDirective` directive above is defined in an independent `MY_GADGET_DIRECTIVE_MODULE` module.

3. To use `myDirective` directive in your gadget or in any other gadget, in the `gadget-definition.xml` file, set the dependency of the gadget to `MY_GADGET_DIRECTIVE_MODULE`. The dependency of the gadget is shown below.

   ```
   var myGadget = angular.module('myGadget-<ID>', ['adf.provider',
   'MY_GADGET_DIRECTIVE_MODULE'])
   ```

   This makes the gadget dependent on the `MY_GADGET_DIRECTIVE_MODULE` module, and the gadget will be able to use all the services, factories, and directives from the new module.

4. If third party AngularJS libraries are used:

- Save the library `js` files under the script directory of your gadget.
- Include modules in your gadget by editing gadget definition file.
- Specify modules as per `Dependencies` section.

Invoking JavaScript functions with same name in different libraries

For AngularJS gadget, JavaScript functions must be specifically defined in either `controller.js` or as AngularJS services, which are singleton objects or single use classes. Duplicate functions inside different services can be easily invoked by using the service injections. For non-AngularJS based gadgets, because the functions can be directly defined on the Window object, functions with same might result in conflicts.

To resolve conflict between similarly named functions, it is recommended that you encapsulate functions inside a binding function, and then use the binding functions to invoke the functions inside.

For example,

```
var GadgetOne_Controller = function($scope) {
    this.userDefinedFunctionOne = function() {
        console.log("from userDefinedFunctionOne function of mygadget");
    }
    this.userDefinedFunctionTwo = function() {
        console.log("from userDefinedFunctionTwo function of mygadget"); }
    }
}
var gadgetOne_Controller= new GadgetOne_Controller(); // NOTE THE
DIFFERENT CASES
```

In this case, you can invoke the `gadgetOne_Controller.userDefinedFunctionOne()` from your view file.

Using third party libraries in gadgets

Including third party libraries in gadgets

To include third party libraries in gadgets:

1. Save the external libraries (js files) under the scripts directory in the gadget.
2. Update the `gadget-defintion.xml` file of the gadget to include the library.
   a. Open the **gadget-defintion.xml** file under the **Scripts and Styles** section.
   b. Click **Add** to add the scripts to the gadget.
3. Deploy the gadget to My webMethods Server.

The external scripts will be loaded as part of your gadget. You can now add the required behavior to the gadget controller using the external scripts.

Loading external libraries

Use `OC Lazy Loader` to load external libraries.

Some external JavaScript libraries cannot be used by saving a copy of the JavaScript files in a local system. For example, you cannot store libraries of Google Maps in a local file system and add reference to these libraries in the local folder. Such external libraries must be accessed directly in the `view.xhtml` file. Referring to these libraries using the `<script>` tag inside the `view.xhtml` will not work because the gadgets are AngularJS based, and for the controller to work, the external JavaScript files must be completely loaded. Hence, use the `ocLazyLoader` provided by the gadget framework to import the external JavaScript files in your `view.xhtml` file.

Following code snippet shows how to use `ocLazyLoader` to load external JS libraries.

```
<div oc-lazy-load="['..js/testModule1.js', '..js/testModule2.js']">
//YOU CAN PROVIDE DIRECT URL TO THE JAVASCRIPT SERVED THROUGH A CDN
//YOUR HTML CODE GOES HERE
</div>
```

If the link to the JavaScript file does not end with a `.js` extension, use a `js!` prefix to your URL as shown below.

```
<div oc-lazy-load="['js!https://maps.googleapis.com/maps/api/js?key=YOUR
_API_KEY']">
<div class="row">
    <map id="map_canvas" style="height: 760px;"></map>
</div>
```

You can get the API key from Google site: "https://developers.google.com/maps/documentation/javascript/get-api-key"

You can also use external AngularJS libraries by directly including the module through `ocLazyLoad`.

Defining success and error notification in gadgets

Use `EventBus` to trigger success and error notifications from a gadget.

To send success notifications, use the code below.

```
$scope.eventBus.fireEvent(NotificationConstants.SUCCESS,"PROVIDE YOUR
SUCCESS MESSAGE HERE");
```

To send error notifications, use the code below.

```
$scope.eventBus.fireEvent(NotificationConstants.ERROR,"PROVIDE YOUR
ERROR MESSAGE HERE");
```

You can send notifications to any place that is within the scope of the `EventBus` object. In the controller `init` block, add the `EventBus` in the `$scope` object so that the `EventBus` is easily accessed through the `$scope` object (such as directives).

Using forms in gadgets

Using gadgets, you can capture HTML form values and pass it to other gadgets by submitting the form.

Use the following code in your gadget `view.xhtml` file, if you want to submit a form with three fields: `First Name`, `Last Name`, and `Phone`.

```
<form role="form" name="myForm">
    <div class="form-group row">
        <label for="fname" class="col-md-4">First Name:</label>
        <input type="text" class="col-md-8 remove-paddings" name="fname"
id="fname" data-ng-model="config.params.fname"></input>
    </div>
    <div class="form-group row">
        <label for="lname" class="col-md-4">Last Name:</label>
        <input type="text" class="col-md-8 remove-paddings" name="lname"
id="lname" data-ng-model="config.params.lname"></input>
    </div>
    <div class="form-group row">
        <label for="lname" class="col-md-4">Phone:</label>
        <input type="text" class="col-md-8 remove-paddings" name="phone"
id="phone" data-ng-model="config.params.phone"></input>
    </div>
    <input class="btn bc-button row" type="button" value="Submit Form"
onclick="submitMyForm()"></input>
</form>
```

The code above provides a form with three fields. Each field can be data-bound to the `config` parameters if required.

On clicking **SUBMIT** button on a form, a JavaScript function, `submitForm` is called. The code for `submitForm` function is as shown below:

```
function submitForm(){
        var fname= document.getElementById("fname").value;
        var lname= document.getElementById("lname").value;
        var phone= document.getElementById("phone").value;
        var href ="";
        if(window.location.href.indexOf("?")>0){
            href = window.location.href.substring(0,
            window.location.href.indexOf("?"));
        }else{
            href=  window.location.href;
        }
        var actionUrl = href+"?fname="+fname;
        actionUrl =actionUrl+"&lname="+lname;
        actionUrl +="&phone="+phone;
        window.location.href=actionUrl;
        window.location.reload();
    }
```

The code above will append the values to the URL as URL parameters that get bound to the receiving gadget through the `config` object.

Note: For parameters to work, add the parameter names to the `gadget-definition.xml` under the parameters section. Only the parameters defined in the `gadget-defintion.xml` will be received or bound to other gadgets.

To display the form parameters in another receiving gadget, use the code below:

```
<div class="form-group">
    <label for="fname">First Name:</label>
    <label>{{config.params.fname}}</label>
</div>
<div class="form-group">
    <label for="lname">Last Name:</label>
    <label>{{config.params.lname}}</label>
</div>
<div class="form-group">
    <label for="lname">Phone:</label>
    <label>{{config.params.phone}}</label>
</div>
```

Note: To send multiple values (array of values) as part of a form field, you can send coma separated values. An array of values can be parsed on the receiving gadget and rendered in a drop-down list.

Accessing services and functions in XHTML files and controller

Accessing services and functions in AngularJS gadgets

You can define your services, factories, and providers in `custom.js`.

1. Attach the services, factories, and providers to the gadget module or define them within your own module.
2. Set the dependency of the gadget module to the custom module. If you look up `config.js` of your gadget, you would find the gadget module defined as below:

   ```
   var myGadget = angular.module('<MY_GADGET_MODULE>', ['adf.provider',
   'MY_GADGET_DIRECTIVE_MODULE'])
   .config(){.....
   ```

   Here `MY_GADGET_MODULE` refers to the gadget module.
3. Define services, factories, and providers:

   - Attach services, factories, and providers to `MY_GADGET_MODULE`:

     ```
     MY_GADGET_MODULE.service('<SERVICE_NAME', [<INJECTABLES>,
     new ServiceFunction(<INJECTED_OJECTS>){
        //Service Code
     }]);
     ```

   - Define customised modules:

     ```
     angular.module("MY_CUSTOM_MODULE",[]).service('<SERVICE_NAME',
     [<INJECTABLES>, new ServiceFunction(<INJECTED_OJECTS>){
        //Service Code
     ```

```
}]);
```

Note: Set the dependency of the gadget to `MY_CUSTOM_MODULE` in the `gadgetDefintion.xml` to use the service in your gadget. service can be injected into the Controller function using the `$inject` method. Inject objects of services, factories, and providers in the `init` method of the Controller in order.

Accessing services and functions in non AngularJS gadgets

Non AngularJS Gadgets will not have access to AngularJS services, factories, and directives. If you have custom functions defined in the `custom.js`, use them directly in your controller code or in `view.xhtml`. It is always safer to make them unique to avoid potential conflicts.

For example:

```
var SomeUniqueFunction1 = function() {
         this.userDefinedFunctionOne = function() {
            console.log("from userDefinedFunctionOne function of mygadget");
        }
        this.userDefinedFunctionTwo = function() {
            console.log("from userDefinedFunctionTwo function of mygadget");
        }
}
var SomeUniqueFunctionId= new SomeUniqueFunction1();
```

In `view.xhtml`, call `SomeUniqueFunctionId.userDefinedFunctionOne()`. This will print from `userDefinedFunctionOne` function of `mygadget` in the console.

Using custom JS or CSS files in gadgets

To include your own JS or CSS files in gadgets, perform the following steps.

1. Copy the custom JS or CSS files to the respective directories under the gadget directory.
   - For JS files, copy to **YOUR_PROJECT > WebContent > gadgetName > scripts**.
   - For CSS files, copy to **YOUR_PROJECT > WebContent > gadgetName > styles**.
2. Navigate to the **gadgetDefinition.xml** located under **YOUR_PROJECT > WebContent > WEB-INF > gadgets > <Gadget_ID>**.
3. Edit the `gadget-definition.xml` file to include the scripts or CSS.
4. Select the **Gadget Defintion Editor** tab.
5. Expand **scripts and styles**, and add the JS files or CSS files by navigating to the respective directory. If your JS files contains one or more AngularJS modules you might want to add to the gadget, specify the module names in the **Dependencies** section.

6. Directly invoke the custom functions from your own JS files directly in the controller. For CSS files, the styles will be automatically applied to your gadget.

   Note: Styles under CSS files are applied universally to all the gadgets. Make sure you have applied selectors in your CSS to apply it selectively to your gadgets. For example, if you use a style shown below, this style will get applied to all elements having `myStyle` as class.

   ```
   .myStyle{
      border:solid 1px #FFF;
   }
   ```

   To ensure proper encapsulation, use style as shown below.

   ```
   .myGadget .myStyle{
      border:solid 1px #FFF;
   }
   ```

7. Use `myGagdet` style in the class of the root element in your gadget's view file.

   ```
   <div class="myGadget">
      <div class="myStyle">
         //STYLE GETS APPLIED HERE
      </div>
   </div>
   ```

   Note: Style will not be applied in `myGadget2`.

   ```
   <div class="myGadget2">
      <div class="myStyle">
         //STYLE DOES NOT GET APPLIED HERE
      </div>
   </div>
   ```

Reusing JS files and CSS files across gadgets

The JS files and CSS files added to the `gadgetDefintion.xml` are also globally available for use in other gadgets. All JS files are included in the common `gadget-framework.js` file, and the CSS files are included in `gadget-framework.scss` file.

For AngularJS modules, to use modules of other gadgets, set the dependencies to the modules of other gadgets.

```
var GadgetOne_Controller = function($scope) {
    this.userDefinedFunctionOne = function() {
        console.log("from userDefinedFunctionOne function of mygadget");
    }
    this.userDefinedFunctionTwo = function() {
        console.log("from userDefinedFunctionTwo function of mygadget"); }
    }
}
var gadgetOne_Controller= new GadgetOne_Controller(); // NOTE THE
DIFFERENT CASES
```

5 Communicating Between Gadgets

About communication between gadgets

This section provides information about various methods to allow communication between gadgets. It also provides information about the services and settings required to enable communication between the gadgets.

Communicating between gadgets using events

Communication between gadgets is necessary to allow information to be shared between one or more gadgets. Gadget communication is possible only between AngularJS gadgets.

There are two ways to allow communication between gadgets:

- JavaScript based EventBus
- AngularJS events

AngularJS defined events are sometimes not favorable for communication between gadgets as we need to decide the event flow (upwards or downwards) based on the logic. A more appropriate way for communication is to provide publish-subscribe mechanism for communication.

Gadget framework provides another communication mechanism using a JavaScript based `EventBus` that registers all the events from the controller when the controller is getting loaded.

Using EventBus

The gadget framework provides an AngularJS service called as `EventBus` to allow communication between gadgets. The `EventBus` can effectively provide gadget communication using publish-subscribe mechanism.

To use `EventBus` in your gadgets:

1. Inject the `EventBus` provider in the controller if not already included.

```
gadget_controller.$inject = [ '$scope', 'RestServiceProvider','EventBus',
'$log','config'];
//INJECTING EVENTBUS IN CONTROLLER
```

2. Add the listener logic to the subscriber controller(s). In the `defineListener` block, invoke `addEventListener` on the `eventBus` with the event type name.

```
this.eventBus.addEventListener("SOME_EVENT_NAME",this.
_handleEvents.bind(this));
```

3. Provide the logic for handling the event in the `_handeEvents` block as shown below. Here, the `exampleHandleEventAction` function is invoked after receiving the event.

Define the `exampleHandleEventAction` function on `$scope` inside the `defineScope` block.

```
_handleEvents:function(eventType,payload,context){
            /* Logic to handle events
             */
             switch(eventType){
             case "SOME_EVENT_NAME":
                    /* Add Event Handling Logic for GLOBAL_EVENT */
                    this.$scope.exampleHandleEventAction(payload);
//ONCE EVENT IS RECEIVED, INVOKE THE exampleHandleEventAction
function on $scope.
                    break;
             }
      },
```

4. Clean up the listener on Controller unload. This is required to eliminate unnecessary event calls when no controller is available on the view.

```
this.eventBus.removeEventListener("SOME_EVENT_NAME",this.
_handleEvents.bind(this));
```

5. Trigger the event from the publisher controller.

```
this.eventBus.fireEvent("SOME_EVENT_NAME", "Some Event!");
```

Using Angular events

AngularJS provides three services to allow communication between gadgets:

- `$broadcast`
- `$emit`
- `$on`

`$broadcast`

`$broadcast` service dispatches an event name downwards to all child scopes (and their children) and notify to the registered `$scope` listeners. The event life cycle starts at the scope on which `$broadcast` was called. All listeners for the event on this scope get notified. Afterwards, the event traverses downwards toward the child scopes and calls all registered listeners along the way. The event cannot be canceled.

```
$scope.$broadcast('eventName', { message: msg });
```

`$emit`

`$emit` service dispatches an event name upwards through the scope hierarchy, and notifies the registered `$scope` listeners. The event life cycle starts at the scope on which `$emit` was called. The event traverses upwards towards the root scope, and calls all the registered listeners along the way. The event will stop propagating if one of the listeners cancels it.

```
$scope.$emit('eventName', { message: msg });
```

$on

`$on` service listens to the events of an event type. `$on` can catch the event dispatched by `$broadcast` and `$emit` and handle the event accordingly.

```
$scope.$on('eventName', function (event, args) {
 $scope.message = args.message;
 console.log($scope.message);
});
```

Adding gadget settings

The `settings.xhtml` file of a gadget provides specific configurations at run time to a gadget.

For example, in a gadget for charting, you might want to specify the chart type and other parameters for charting. The communication between the gadget controller and the `settings.xhtml` settings file is done through a `config` object injected to the controller as shown below.

```
//INJECTING CONFIG TO CONTROLLER
ExampleGadgetController.$inject = ['$scope', 'RestServiceProvider',
'EventBus','config'];
```

The `config` object injected in the code above is the gadget configuration service, which contains the configuration information and is available at the `init` function in the controller, and is tied to the controller scope so that the values can be two-way bound in the `settings.xhtml` file.

For example, the code to bind an input box to a `config` object, and handle the input box in the controller is shown below.

```
settings.xhtml
<form class="form-horizontal">
    <div class="control-group">
        <label class="control-label hint--top">Gadget Configuration 1</label>
        <div class="controls">
            <input type="text" data-ng-model="config.params.config1">
        </div>
    </div>
    <input type="button" data-ng-click="applySettings()"></input>
</form>
controller.js
....
init : function(scope, restClient,EventBus,config) {
        ...
//ADDING THE CONFIG OBJECT TO SCOPE TO BIND TO UI
        scope.config = config;
        ....
}
...
defineScope : function() {
        var _this=this;
        this.$scope.applySettings:function(){
            console.log(_this.config.params.config1)
// This would print the value from the settings.xhtml file
            if(_this.config.params.config1==="SOME VALUE"){
            //Handle Accordingly
```

```
            }
        }
}
```

Connecting multiple views with controller

All AngularJS gadgets generated using Designer have a controller that is automatically bound to `view.xhtml`. If you have multiple XHTML files to create the view, specify the sub-views in `view.xhtml` file.

Defining a view with sub-views in multiple XHTML files

1. Create the `<file_name>.xhtml` under the views directory.
2. Use `data-ng-include` attribute to tie up with the parent view.

   If you have created `view1.xhtml` and `view2.xhtml`, edit `view.xhtml` as shown below.

   ```
   <div data-ng-include="/<CONTEXT_ROOT_OF_APPLICATION/<GADGETS_DIRECTORY>
   /views/view1.xhtml">
   </div>
   <div data-ng-include="/<CONTEXT_ROOT_OF_APPLICATION/<GADGETS_DIRECTORY>
   /views/view2.xhtml">
   </div>
   ```

   Replace `<CONTEXT_ROOT_OF_APPLICATION>` and `<GADGETS_DIRECTORY>` accordingly. All views specified under the `view.xhtml` are automatically bound to the controller.

   Note: You can use the `data-ng-model` or `data-ng-bind` attributes to setup two-way binding to the controller scope variables.

Invoking a function on a controller

1. Define a scope function under the `defineScope` block in the controller.

   ```
   defineScope : function() {
        ...
        this.$scope.myFunction= function() {
           //DO SOMETHING
        }
       ....
   },
   ```

2. Invoke the function on the scope directly on user action or based on some business logic.

   ```
   <button data-ng-click="myFunction()"> value="CLICK ME!"
   // INVOKES THE FUNCTION myFunction defined on $scope on user click
   </button>
   ```

6 Improving Gadget Performance

Gadget Performance

Business Console gadgets are built on top of AngularJS. Hence, improving gadget performance would mean improving the way the gadgets are coded.

This section explains various techniques to improve the gadget performance.

Techniques for improving gadget performance

Paginating

Ensure that all RESTful services that return huge data sets are well paginated on the server side. For example, if you are trying to load a grid with 1000 records, paginate 10-20 records at a time. You can have grid control, and lazy load data sets as and when users scroll through the grid. This will ensure that less data is processed by the user interface, and will improve performance.

Minimizing the use of watchers

AngularJS scans and keeps track of all the changes in the application. This means that every watcher is monitored for update requests (digest cycle). If one of the watchers relies on another watcher, AngularJS re-runs the digest cycle to make sure that all of the changes are propagated. Digest cycle runs continuously until all the watchers are updated and the application is stabilized. Even though JavaScript execution is really fast in modern browsers, if you add too many watchers in AngularJS, your gadget might slow down. Although it is impossible to avoid watchers, minimizing watchers will definitely help performance.

For example, when `use bind once where possible` watchers are set, AngularJS adds the `::` notation to allow one time binding. AngularJS will wait for a value to stabilize after the first series of digest cycles, and will use that value to render the DOM element. After that, AngularJS will remove the watcher and forget about that binding. You can use this to bind constant values which do not change throughout the application.

```
$scope.$watch
{{ }} type bindings
Most directives (i.e. ng-show)
Scope variables scope: { bar: '='}
Filters {{ value | myFilter }}
ng-repeat
```

Using ng-if

Use `ng-if` instead of `ng-show` wherever possible. `ng-show` will add the `display:none` style to your HTML code depending on the condition. So your HTML will always be part of the DOM even if it is hidden. `ng-if` will not add the HTML code to your DOM if the condition is not satisfied, thus minimizing the size of the DOM object. Make sure that your use case is satisfied by `ng-if`.

7 Importing and Enhancing AgileApps Forms

Enhancing AgileApps Forms

AgileApps forms are built either with database objects or case objects. In AgileApps, these forms have limited capabilities to allow you to modify the UI controls, business logic, and styles. A new capability allows you to import the AgileApps forms as plain HTML and JavaScript into Software AG Designer and enhance the imported forms. In conjunction with this, you can use the UI generated by the REST connector code along with imported AgileApps form. See " Drag and Drop Existing REST Resources to Generate the Gadget UI" on page 61.

Limitations

The following limitations are observed while importing the AgileApps forms:

- The layout rules are not imported from the AgileApps form.
- Components such as attachments and time are imported but do not function as intended.
- On the AgileApps form, after business logic and new controls are added in the Software AG Designer and deployed on My webMethods Server, these new controls do not reflect in the AgileApps form on AgileApps. In the interim, if you add new controls in the AgileApps form, those controls do not reflect in the gadget generated in Software AG Designer automatically. But you can manually add the new controls and business logic in the imported AgileApps form gadget. See "Example: Use Case to Add New Business Logic" on page 86 for more information on how to add new business logic in Software AG Designer.

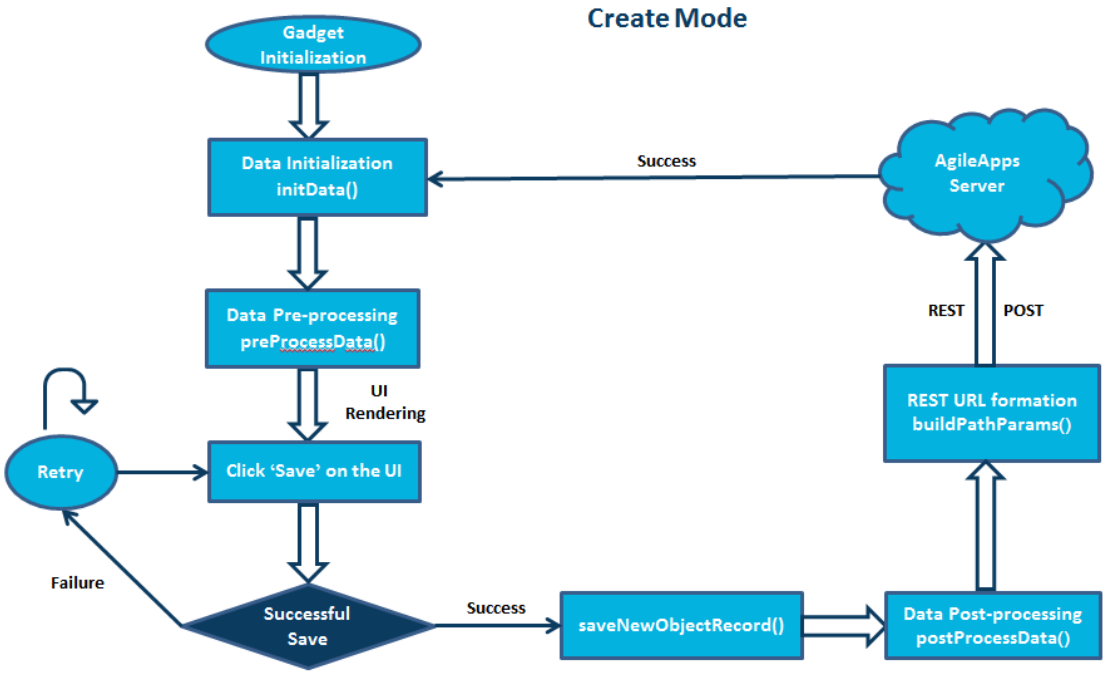
Lifecycle of an AgileApps Form Gadget

The lifecycle of an AgileApps form gadget is depicted in the flow diagrams and the details of each stage is available in the following table:

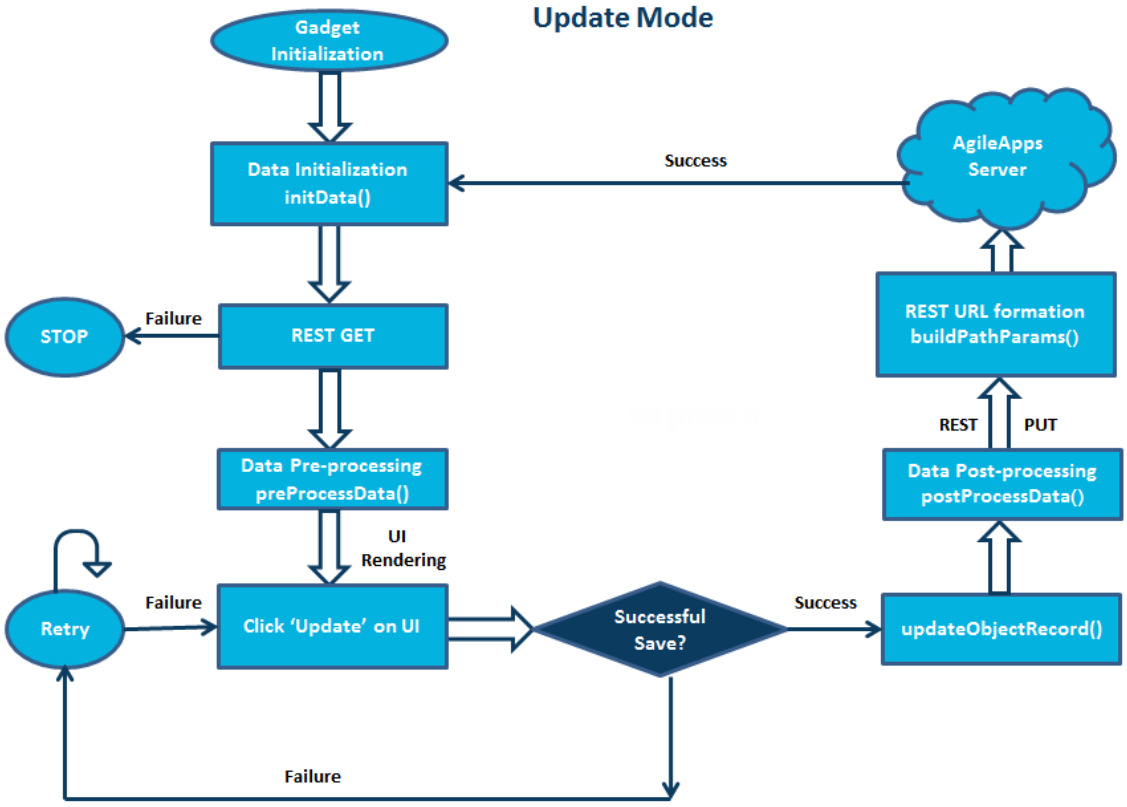
Stages	Create Mode	Update Mode
initData()	Initializes the UI model with default values.	Initializes the UI model by retrieving the record from the AgileApps server using either the AgileApps object name or object type Id and record ID.

Stages	Create Mode	Update Mode
saveNewObjectRecord()	A new object is created with the specified values for the object name or object type.	This stage does not apply to update mode.
updateObjectRecord()	This stage does not apply to create mode.	Updates an existing record with the specified values.
preProcessData()	Converts the data from the back end service response to the UI model format. For example, time field on the UI requires a complete dataTime value but the back end service returns only the timestamp.	Converts the data from the back end service response to the UI model format. For example, time field on the UI requires a complete dataTime value but the back end service returns only the timestamp.
postProcessData()	Converts the data from UI model format to match the back end service requirements. For example, multi check box on the UI needs an Array, but the back end service requires a string with commas as a separator.	Converts the data from UI model format to match back end service requirements. For example, multi check box on the UI needs an Array, but the back end service requires a string with commas as a separator.
buildPathParams ()	Forms the REST URL with the required path and query parameters.	Forms the REST URL with the required path and query parameters.

The following flow diagram depicts the control flow in the create mode:



The following flow diagram depicts the control flow in the update mode:



Inherited Event Subscriptions

On importing an AgileApps form into Software AG Designer, a few event subscriptions are inherited along with all the required parameters. The `gadgetdefinition.xml` contains this information.

The gadget generated with the AgileApps form subscribes to `AA_RECORD_CHANGE_EVENT` and `AA_MODE_CHANGE_EVENT` by default. A few events such as `GEO_LOCATION` and `LOOKUP` are published by the generated gadget based on the actions you perform.

Payload for both the events are expected in a certain format. The payload for `AA_RECORD_CHANGE_EVENT` needs to be in the following form:

```
"AA_OBJECT_RECORD_CHANGE":
                        payload = {AAObjectName:"",
                AAObjectID:"",
                AAObjectRecordID:""
                }
```

The payload for `AA_MODE_CHANGE_EVENT` needs to be in the following form:

```
"AA_GADGET_MODE_CHANGE":
          payload = {AAObjectName:"",
                  isNewMode:""
                  }
```

When the Workstream gadget is placed along with the generated AgileApps form gadget in a single AppSpace, the gadgets interact seamlessly. In the Workstream gadget,

if you select a case instance, the AgileApps form is automatically refreshed with the details of the selected case instance.

If the generated gadget is placed along with the Business Console maps gadget, then the AgileApps locations are identified and mapped automatically in the Business Console maps gadget.

Switching Between the Create and Update Modes

On clicking the Settings  icon, the gadget settings model dialog appears. Select **True** in **isNewMode** to make the form appear with create controls. Select **False** for the form to appear with editing controls.

New Files Generated on Importing AgileApps Forms

A few files are generated anew when you import an AgileApps form. The following table contains the list:

Folder Names	File Names	File Description
Views		
	`view.xhtml`	This file contains the parent view that is visible on the UI. The `aa_view_partial.xhtml` file is linked to the `view.xhtml` page as an angular partial directive.
	`aa_view_partial.xhtml`	This file contains the complete HTML content of the AgileApps form.
	`settings.xhtml`	Object name and object type fields selected while importing the AgileApps form are non-editable fields. The **RecordID** field is used to retrieve the AgileApps object instance details and it is populated with a randomly chosen object instance by default. This is an editable field. Mode field controls the behavior of the imported AgileApps form.

Folder Names	File Names	File Description
		It can be used to either create or update a record.
Scripts		
	`controller.js`	This file provides the model layer for the `view.xhtml` page.
	`config.js`	This file contains internal configurations used by the gadgets framework. Any modification to this file may result in unexpected behavior of the generated gadget.
	`directive.js`	This file is the angular directive. It contains all the business logic.
	`custom.js`	This file remains blank on creation. You can add custom business logic in this file.

Importing an AgileApps Form into Software AG Designer

You can import an AgileApps form into Software AG Designer and make enhancements such as change the theme, add business logic, and so on. Importing a form from AgileApps is unidirectional. Ensure that you configure the AgileApps host in the **Administer Business Console** menu of Business Console. After the required changes are made to the AgileApps forms in Software AG Designer, you need to deploy it on My webMethods Server to make it available on Business Console.

To import AgileApps into Software AG Designer

1. Launch the Software AG Designer in the UI perspective.
2. Create a portlet or web application in the UI Development perspective in Software AG Designer.
3. In the **Solutions** tab, right-click on the UI project and select **Import AgileApps Forms From Gadget**. The **New AgileApps Form** wizard appears.
4. Type the AgileApps credentials and click **Next**.
5. Select the application, objects types, and forms you want to import in Software AG Designer and click **Finish**. The imported AgileApps form appears as a UI Development project.

6. Configure the AgileApps host in the **Administer Business Console** menu of Business Console.

   Ensure that you configure the same host you used for importing the AgileApps form or ensure that the application is available in the configured AgileApps server.

7. Make the desired changes to the AgileApps form using Software AG Designer.

See "Deploying gadgets to My webMethods Server " on page 14 for instructions deploying the imported AgileApps form in the My webMethods Server and consequently see "Viewing gadgets " on page 16

Modifying an AgileApps Form in Software AG Designer

You can modify a form imported from AgileApps in the UI perspective of Software AG Designer.

To modify AgileApps into Software AG Designer

1. Launch the Software AG Designer in UI perspective.
2. In the **Solutions** tab, navigate to the imported **<gadget_name_project>** > **Views** > **aa_view_partial.xhtml** file. This file contains all the UI fields pertaining to AgileApps.

   The **view.xhtml** file contains the code that renders the UI.

3. Open the **aa_view_partial.xhtml** file to edit it.

   You can add or remove the fields using the HTML code.

4. Click **Save**.

See "Deploy gadgets to My webMethods Server" on page 14 for instructions on how to deploy the modified AgileApps form in the My webMethods Server and consequently see "Viewing gadgets " on page 16 for information on adding the modified AgileApps form as a gadget in Business Console.

Ensure that the same version of the modified AgileApps form is deployed on the My webMethods Server and imported as an AppSpace on Business Console.

Example: Use Case to Add New Business Logic

The example use case lists instructions on how to add a new JavaScript function in the **aa_view_partial.xhtml** file to enhance the business logic.

To add a new JavaScript function AgileApps using Software AG Designer

1. Launch the Software AG Designer.
2. In the **Solutions** tab, navigate to the imported **<gadget_name_project>** > **Scripts** > **aa_view_directive.js** file. This file contains all the business logic.

3. Open the **aa_view_directive.js** file to edit it.
4. In the `defineScope()` function, add a custom function as follows:

```
$scope.custom=function(){
 $scope._Data.customMessage = "hello world"
 }
```

5. Navigate to the **<gadget_name_project>** > **Views** > **aa_view_partial.xhtml** file. This file contains all the UI fields pertaining to AgileApps.
6. Add the new UI fields and controls to bind the function(custom) and `model(_Data)` variable `(customMessage)` created previously.

   The example code snippet is as follows:

```
<div class="form-group" style="margin-top: 10px;">
<button data-ng-click='custonFunc()'>Click Here</button>
 <div class="col-sm-6 col-md-6">
  <input type='text' class='bc-newci-text form-control'
  data-ng-model='_Data.customMessage'/>
 </div>
</div>
```

8 Troubleshooting Gadgets

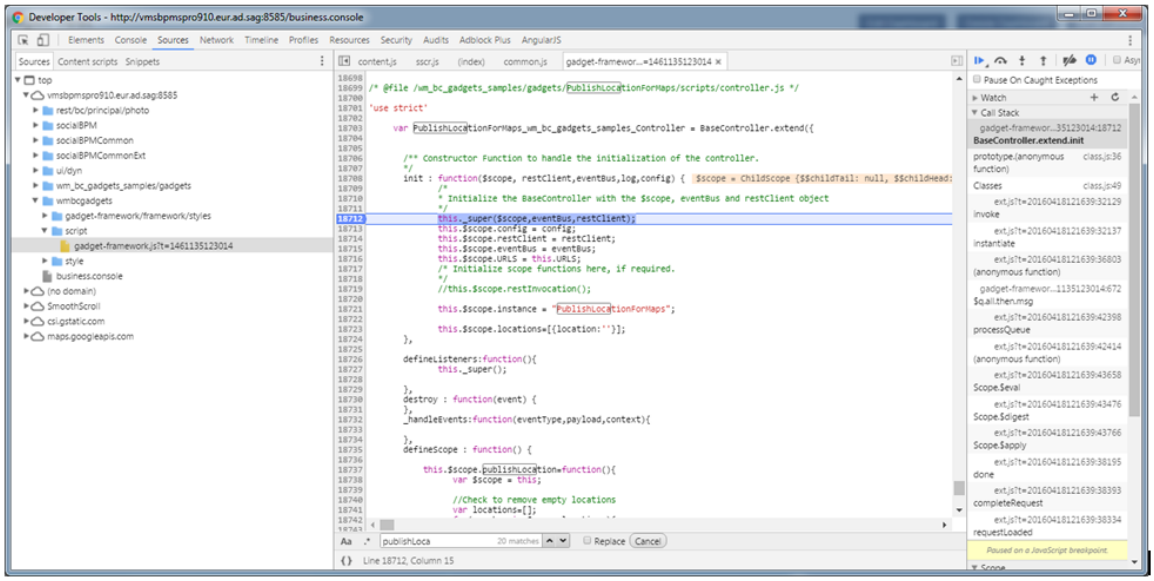
About troubleshooting gadgets

This section explains how to test gadgets and how to troubleshoot the gadgets in case of exceptions. The gadget code is included in a JavaScript file called `gadget-framework.js`. The `gadget-framework.js` file of a gadget is used to test the gadget code.

Testing gadget in a browser

Testing a gadget in chrome browser

1. Open the **Developer Tools** (Press F12).
2. In **Sources** tab, from the left hand side menu, expand **wmbcgadgets > script**.
3. Double-click on **gadget-framework.js**. This file contains all your gadget code.
4. Search for your controller function (Use Ctrl+F).
5. Set breakpoints accordingly



Handling exceptions

The gadget framework in Designer handles all the compile-time errors in your gadget code. To ensure that your gadget is free of syntax error, check the errors and warnings sections, and ensure that no errors are listed. If there are any warnings, program the gadget to handle the warnings to ensure that your gadget works properly. To handle

runtime exceptions, you can encapsulate your scope functions in a `try/catch` block, and log the errors appropriately.

Using a CSS URL data type in the CSS file of a gadget

You can use a CSS `URL` data type in the CSS file of the gadget.

To use a CSS URL data type in the CSS file of a gadget

- Prefix the relative path with `CONTEXT___ROOT` followed by the relative path of the gadget.

  `CONTEXT___ROOT` automatically resolves to the application context root value during runtime by the gadget framework. For example:

```
@font-face {
    font-family: 'My Glyphicons Halflings';
    src: url(/CONTEXT___ROOT/FontCssGadget/styles/fonts/
         glyphicons-halflings-regular.eot);
    src:
         url(/CONTEXT___ROOT/FontCssGadget/styles/fonts/
         glyphicons-halflings-regular.eot?#iefix)
         format('embedded-opentype'),
         url(/CONTEXT___ROOT/FontCssGadget/styles/fonts/
         glyphicons-halflings-regular.woff2)
         format('woff2'),
         url(/CONTEXT___ROOT/FontCssGadget/styles/fonts/
         glyphicons-halflings-regular.woff)
         format('woff'),
         url(/CONTEXT___ROOT/FontCssGadget/styles/fonts/
         glyphicons-halflings-regular.ttf)
         format('truetype'),
         url(/CONTEXT___ROOT/FontCssGadget/styles/fonts/
         glyphicons-halflings-regular.svg#glyphicons_halflingsregular)
         format('svg');
}
.imagecss-sample-gadget {
    background-image: url('/CONTEXT___ROOT/FontCssGadget/images/logo.png');
    height: 100px;
    width: 150px;
}
```

  Here `CONTEXT___ROOT` is replaced by the `<application_name>` that you provided while creating the CAF portlet or web application.