Ozone Developer’s Guide

DOD GOSS

Version 8.0.0.0, 2019-07-01

Table of Contents

1. Introduction

1.1. Objectives

The purpose of this guide is to explain how to create a simple widget or integrate an existing widget into the Ozone Widget Framework (OWF).

1.2. Document Scope

This guide is written for software developers who want to change an existing application into an OWF-compatible widget(s) or understand the APIs available to them for building widgets.

1.3. Usage

1.3.1. File References

All directory and file paths in this guide are relative to the `ozone/docs/developers_guide/` directory.

Example: The full path of `node-server/` would be found at `ozone/docs/developers_guide/node-server/`.

1.3.2. Example Code

All of the code examples listed in this document can be found in the owf.war. When unpacked or unzipped, the bundle will contain a `/ozone-framework-server/owf-framework/sample-widgets` which contains .zip files of example widgets with source code built in different technology stacks. The examples included in the distribution are detailed in [example-widgets].

1.4. Additional Notes

1.4.1. Terminology Changes

Previous versions of the Ozone documentation used the terms "App Component" and "Applications" (or "Apps").

To maintain a consistent nomenclature across the documentation and code base, these terms are being deprecated in favor of the original terms "Widget" and "Dashboard".

1.4.2. Warnings

The following APIs are not currently supported by OWF v8:

- Logging API

- State API
- Chrome API
- Drag & Drop API

Additionally, some of the APIs in this document are out-of-date or in the process of being updated or deprecated.

If you run into problems or inconsistencies, please refer to the source code or contact the OZONE Development Team.

1.5. Related Documents

Table 1. Related Documents

Document	Purpose
Quick Start Guide	Walkthrough of basic OWF functions such as using widgets; unpacking the OWF bundle; setting up a local instance of OWF; installing security certificates; truststore and keystore configuration.
User's Guide	Understanding the OWF user interface; adding, deleting, modifying widgets and using intents; accessing and using the Store;using dashboards; creating, deleting, adding, switching, modifying dashboard pages; defining accessibility features such as high-contrast themes.
Administrator's Guide	Understanding administrative tools: adding, deleting, and editing users, groups, widgets, and dashboards; creating default content for users, groups and group dashboards.
Configuration Guide	Overview of basic architecture and security; OWF installation instructions; instructions for modifying default settings; database set up and logging guidance; framework and theme customization instructions; OWF upgrade instructions; directions for adding and deleting help content.

2. Creating a Widget

2.1. Overview

OWF widgets are lightweight Web applications wrapped with a metadata definition that provide a description to the framework of how the widget should load. The widget metadata definition contains a number of fields including a *URL*, *Default Name*, *Default Height* and *Default Width*.

OWF provides a suite of APIs that enable the developer to extend their Web application through the use of inter-widget communication, user preferences and internationalization. Each API is written in JavaScript so that widgets can be built in a large variety of Web technologies.

Three key factors to keep in mind when creating a Widget are:

- OWF supports and encourages a decentralized deployment model. Widgets are not required to be deployed on the same server as OWF and can be distributed throughout the enterprise.
- OWF is Web-technology agnostic. Widgets can be written in the JavaScript capable technology of the developer's choice. OWF enabled applications have been built in varied technologies such as JavaScript (React, Dojo), Java (JSPs, GWT, JSF, Groovy, Grails), .NET (ASP.NET, C# .NET), Scripting Languages (PHP, Perl, Ruby on Rails), and Rich UI Frameworks/Plugins (Flex, Silverlight, Google Earth Plugin, Java Applets).
- The location that hosts Widget descriptor files must have Cross-Origin Resource Sharing (CORS) enabled. Usually, the Web app used to create the Widget also serves as the host for the descriptor file. The method used to configure CORS support will vary based on the type of Web app platform used. For backwards compatibility, the format of the descriptor file itself is unchanged. The template is located in etc/widget/descriptor within the OWF bundle.

This document assumes that the reader has a development background and is familiar with their chosen technology stack. The tutorials found throughout this document will focus on building a simple HTML/JavaScript Web application deployed to a Java Application Server.

2.2. Tutorial

Step 1 - Create the Announcing Clock widget

The complete example files are available for this tutorial under the `ozone/docs/developers_guide/examples/` directory.

Create the following files in the `ozone/docs/developers_guide/node-server/public/` directory:

AnnouncingClock.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Announcing Clock</title>
    <script type="application/javascript" src="./AnnouncingClock.js"></script>
</head>
<body>
    The time is: <span id="clock"> </span>
</body>
</html>
```

AnnouncingClock.js

```
document.addEventListener("DOMContentLoaded", start);

function start() {
    updateClock();
    setInterval(updateClock, 1000);
}

function updateClock() {
    // Format the time portion of the current date
    const timeString = new Date().toLocaleTimeString();

    // Update the time display
    document.getElementById("clock").firstChild.nodeValue = timeString;
}
```

Step 2 - Start the web server

The examples in this guide use a simple web server using Node.js (https://nodejs.org/) and Express (https://expressjs.com/).

Using the command line, navigate to the `node-server/` directory and run the following commands:

```
npm install
npm start
```

Next, navigate to `http://localhost:5000/AnnouncingClock.html` to view the new Announcing Clock widget.

For instructions on deploying these examples as a Java .WAR to an application container, see Appendix A, *Java WAR Deployment*.

2.3. Additional Considerations

2.3.1. Utility JS API

The JavaScript Utility API is provided to allow the developer to determine whether or not the widget is running inside OWF. This is useful if the widget needs to render differently or has different defaults depending on whether or not it is running internal or external to OWF. For instance, if a widget is supposed to turn on logging when running inside OWF, the developer can use the JavaScript Utility API to determine this.

In some previous versions of OWF, a widget launched outside of OWF would spawn an error in an alert window. Now, widget can be launched and used outside OWF. And while certain features, both specific and critical to OWF, such as the launch of security alert windows, preferences and eventing will not operate outside OWF, the error alert window will not launch, provided the widget URL is NOT appended with `owf=true`. Moreover, APIs can often throw exceptions which can make a widget fail to load.

While originally defined in `js/util/widget_utils.js` (depreciated), the interface object now resides within the Ozone.util namespace. The entire namespace has been included in the OWF Widget JS bundles (both debug and min) for convenience.

Table 2. Utility API Namespaces

Namespace	Summary
`OWF.Util`	A namespace containing common utility functions to assist widget developers.

Table 3. Utility API Methods

Method	Summary
`<static> OWF.Util.cloneDashboard()`	Clones dashboard and returns a dashboard cfg object that can be used to create new dashboards.
`<static> OWF.Util.guid()`	Returns a globally unique identifier (GUID).
`<static> OWF.Util.isInContainer()`	This method informs a widget developer if their widget is running in a Container, such as OWF.
`<static> OWF.Util.isRunningInOWF()`	This method informs a widget developer if their widget is running from OWF or from a direct URL call.

2.3.2. Best Practices

Due to the complexity of the OWF APIs, a widget's ability to signal that it is ready to communicate with other widgets provides a helpful tool for developers. This ready signal is typically sent after the Web app has

subscribed to channels, registered RPC functions and Intents, etc. Starting with OWF 6, there is a standard way for widgets to signal this ready status.

To signal that it is ready, a widget calls `OWF.notifyWidgetReady()` after it is finished setting up any communication mechanisms. The OWF Development Team recommends that any widget that uses OWF APIs makes the call. However, widgets that use the Widget Intents API's receive method must make this call.

2.3.3. OWF Bundled JavaScript

All required OWF JavaScript is minified and bundled into one JavaScript file, found in `tomcat/webapps/owf/public/js/owf-widget.min.js`. This shields developers from future changes or upgrades to the underlying JavaScript files.

There are multiple ways that developers can reference the `owf-widget.min.js` file from their widgets. One way is to hard-code a link to the file residing on a specific OWF. This has the disadvantage of tying the widget to a particular OWF instance. Another option is to include a copy of `owf-widget.min.js` with the widget itself, and to use a relative URL to reference it. This makes the widget independent of any particular OWF instance, but ties the widget to a specific version of the `owf-widget.min.js` file. If there is a version mismatch between this file and the OWF version where the widget is run, then problems could arise.

The recommended way to include the `owf-widget.min.js` file into a widget is to create the script reference dynamically so that it always refers to the copy of owf-widget.min.js on the OWF server where the widget is currently running. The dynamic reference can be generated either server-side or client-side. To generate it server-side, use the "Referer" [sic] HTTP header, which will contain the URL of the main OWF page when the widget is launched. To retrieve the URL of the OWF instance for the widget's client-side code, use the following JavaScript:

```
JSON.parse(window.name).preferenceLocation.split('/prefs')[0];
```

3. Adding a Widget to OWF

3.1. Overview

OWF provides a Widget Manager, located by clicking the Administration link in the drop-down User Menu (you must be an OWF administrator to see the Administration link) and then click Widgets Administration. Administrators can use this manager to create widget descriptor files and add widget definitions to OWF. The manager allows an administrator to complete the widget definition or edit the descriptor URL in the user interface, then OWF maps the widget to users in the system. Once a widget definition has been created and mapped to a user, it will then be added to the user's Widgets menu or toolbar depending on the widget type.

Due to the fact that a widget definition actually points to the URL of a lightweight Web application, an administrator is not required to update widget definitions unless the location of the widget changes.

3.2. Tutorial

3.2.1. Creating Descriptor Files for Widgets

Developers can save the Widget information in the descriptor file and then share that file with administrators. This allows administrators to import data instead of typing entries for each field. The administrator simply enters a URL and the widget's information is automatically retrieved from a descriptor file that a developer maintains. Administrators can change properties in the widget's definition once it has been added to OWF. However, an administrator's changes will only affect their deployment of OWF, unless the administrator exports those changes to the Web-accessible location where the descriptor URLs are stored.

Descriptor URLs offer several benefits. They reduce the risk of typing errors when entering widget data into the OWF interface. They allow for several installations of OWF to easily share widget information via the descriptor file. In addition, descriptor files can contain a universal name which is a developer-generated, custom identifier that can be used to identify the widget across multiple OWF instances.

To support Internet Explorer 10 and higher, the location that hosts Widget descriptor files must be CORS enabled. Usually, the Web app used to create the Widget also serves as the host for the descriptor file. The method used to configure CORS support will vary based on the type of Web app platform used. For backwards compatibility, the format of the descriptor file itself is unchanged. The template is located in `etc/widget/descriptor` within the OWF bundle.

To create a widget descriptor URL, follow these instructions:

1. Sign in to OWF as an administrator.
2. Click the Administration link, located on the drop-down User Menu on the toolbar to open the Administration Tools. Select Widgets Administration to open the Widgets Manager.
3. Click Create to open the Widget Editor.
4. Click "Don't have a descriptor URL?"

5. Populate the mandatory fields in the definition and click Apply.

For more information about specific entry fields, please see the Administrator's Guide.

6. OPTIONAL: Add the capability to send intents. An Intent is simply an object describing an action and a data type. Sending an Intent should be tied to a user-generated action such as clicking a button or link. (If the widget does not require an intent, skip this step.) Developers should use the Widget Editor to add and edit intents.
    a. To add an intent, select the Intents tab in the Widget Editor and click Create.
    b. Populate the following fields:
        i. **Action** - The Action should be a verb describing what the user is trying to do (i.e. plot, pan, zoom, view, graph, etc.).

Intents are NOT case sensitive.

        ii. **Data Type** - The Data Type is an object containing the data that the intent is sending. It describes what type of data is being acted upon. The data type format is described in 12.4.1: Recommended Intents Data Type Conventions.

            The format of the data depends solely on how the sending or receiving widget is expecting to use the data. For example, "application/vnd.owf.sample.price" tells the NYSE Widget's how to display price.

        iii. **Send** – Checked by default, this field identifies if the widget can send intents.
        iv. **Receive** – This field identifies if the widget can receive intents.
    c. Click OK.
7. Return to the Widget Manager.
8. Select the new Widget.
9. Click the split Edit button, then select Export from the drop-down menu.
10. Enter a File Name (this will be the name of the HTML descriptor file) and click OK.
11. Save the file to a Web-accessible location like a directory where widget data is stored.
12. Return to the OWF user interface and open the Widget Manager.
13. Select the new widget, and click Edit.
14. From the Widget Editor, enter the new Descriptor URL location, click Load, then, click Apply.

From the Widget Editor, administrators can edit the widget descriptor URL. However, those changes will not change the “master” copy of the descriptor unless they replace the descriptor file stored at the Web-accessible location referenced above.

3.2.2. Sharing Descriptor Files

There are two ways to share widget descriptors:

1. **Export the file**
   To obtain an exportable HTML file, select the widget in the Widget Manager, click the split edit button and select Export.

   Exporting the widget descriptor URL only sends a copy of the file. The administrator will not receive future updates to that file that is stored on the Web-accessible location.

2. **Share the descriptor location**
   Sending a link to the widget descriptor file that is stored on a Web-accessible location will enable the administrator to receive updates to the widget descriptor URL by clicking Load, then Apply in the Widget Editor.

4. Eventing API

4.1. Overview

In order to create rich, interactive and integrated presentation-tier workflows, widgets must be able to communicate with each other; one method to do this is via the Eventing, or Publish-Subscribe, API. The Eventing API is a client-side browser communication mechanism that allows widgets to communicate with each other by using an asynchronous publish-subscribe messaging system.

Widgets have the ability to send and receive data on named channels. All widgets can be built so they can publish messages to any channel, just as all widgets can be built to subscribe to any channel at any time.

There are two main components to the Eventing API. The first component is the supporting infrastructure within OWF that routes messages; however, this should remain mostly transparent to the widget developer. The second component, of more direct interest to developers, is the infrastructure available to each widget, detailed below.

4.2. Tutorial

This tutorial will go through the process of creating a new widget called Second Tracker. The new widget will use the Eventing API to track how many seconds the Announcing Clock widget has been running. The Announcing Clock widget, created in Chapter 2, *Creating a Widget*, must be updated to broadcast an event using the Eventing API.

Example files

The source files for this tutorial are available under the `ozone/docs/developers_guide/examples/` directory.

- `SecondTracker.html`
- `SecondTracker.js`
- `AnnouncingClock_Eventing.html`
- `AnnouncingClock_Eventing.js`

Step 1 - Create the Second Tracker widget

Create the following files in the `ozone/docs/developers_guide/node-server/public/` directory:

SecondTracker.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Second Tracker</title>

    <link type="text/css" rel="stylesheet" href="./css/widget.css">

    <script type="application/javascript" src="./owf-widget.js"></script>
    <script type="application/javascript" src="./SecondTracker.js"></script>
    <script type="application/javascript">
        // If not set, defaults to
/<context>/js/eventing/rpc_relay.uncompressed.html
        // OWF.relayFile = "/owf-sample-
html/js/eventing/rpc_relay.uncompressed.html";
    </script>
</head>
<body>
    <div class="widgetContents">
        <div class="panel-header">
            Second Tracker
        </div>

        <div class="panel-body">
            <table class="messagePanel">
                <tr>
                    <td width="50%">Current Time:</td>
                    <td><span id="currentTime"></span></td>
                </tr>
                <tr>
                    <td width="50%">Connection Uptime (s):</td>
                    <td><span id="minutesOnline">0</span></td>
                </tr>
                <tr>
                    <td width="50%"> Received on channel:</td>
                    <td><span id="channelName"></span></td>
                </tr>
            </table>

            <div id="tracker-error-panel" class="error-panel">
                <span id="error"></span>
            </div>
        </div>
    </div>
</body>
</html>
```

The relay file is configured by setting `OWF.relayFile` to the location of the file. In the above example, `/owf-sample-html/` is assumed to be the root context. The developer must replace `/owf-sample-html/js/eventing/` with the correct relative location of the `rpc_relay.uncompressed.html file` (see the note below for more information).

Pay attention to the OWF relay file argument. In order to work correctly, the relay file must be specified with full location details, but without a fully qualified path. In the case where the relay is residing at `http://server/path/relay.html`, the path used must be from the context root of the local widget. In this case, it would be `/path/relay.html`. Do not include the protocol.

SecondTracker.js

```
document.addEventListener("DOMContentLoaded", function() {
    OWF.ready(start);
});

function start() {
    document.getElementById("currentTime").innerHTML = new Date();

    let channelToUse = "ClockChannel";

    const launchConfig = OWF.Launcher.getLaunchData();

    // No launch configuration; use the default channel
    if (!launchConfig) {
        document.getElementById("error").innerHTML = "Widget was launched
manually";
        document.getElementById("tracker-error-panel").style.display =
"block";
    }

    // Got the launch configuration; use the dynamically specified channel
    else {
        // Parse the configuration
        const launchConfigJson = JSON.parse(launchConfig);
        channelToUse = launchConfigJson.channel;

        // Hide the error panel
        document.getElementById("tracker-error-panel").style.display = "none";
    }

    // Update the displayed channel
    document.getElementById("channelName").innerHTML = channelToUse;

    // Subscribe to the channel using the Eventing API
    OWF.Eventing.subscribe(channelToUse, this.update);
}

/**
 * The function called every time a message is received on the eventing
channel
 */
function update(sender, msg) {
    let count = parseInt(document.getElementById("minutesOnline").innerHTML);
    count = count + 1;
    document.getElementById("minutesOnline").innerHTML = count;
    document.getElementById("currentTime").innerHTML = msg;
}
```

The code above performs several functions:

The initial code waits for the page to be loaded, using the standard `DOMContentLoaded` event. Once the page is ready, the widget must also wait for OWF to become ready. The `OWF.ready` function takes a callback that executes when the widget has connected to OWF and the widget APIs are ready to use.

SecondTracker.js

```
document.addEventListener("DOMContentLoaded", function() {
    OWF.ready(start);
});
```

The `start` function initializes the widget. It subscribes to the channel `ClockChannel`, passing in its `update` function. Additionally, if the widget has been launched using the Launcher API, it uses the launch data to select a different channel name to subscribe to. Refer to the [launcher-api] for additional details.

The `update` function serves as a callback for the Eventing API. Whenever a message is broadcast on the channel that the update function was subscribed to (in this case, `ClockChannel`), the function will be invoked. All Eventing API callback functions should take two arguments: `sender` and `message`. When the update function is fired, the count is incremented, and the `innerHTML` of the `currentTime` span is updated to reflect the message sent by the clock.

Step 2 - Include the Widget API JavaScript library

The Announcing Clock must be updated to publish messages on the expected channel.

Replace the code in the `AnnouncingClock.html` file with the following:

AnnouncingClock.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Announcing Clock (Eventing)</title>

    <link type="text/css" rel="stylesheet" href="./css/widget.css">

    <script type="application/javascript" src="./owf-widget.js"></script>
    <script type="application/javascript">
        // If not set, defaults to
/<context>/js/eventing/rpc_relay.uncompressed.html
        // OWF.relayFile = "/owf-sample-
html/js/eventing/rpc_relay.uncompressed.html";
    </script>

    <script type="application/javascript"
src="./AnnouncingClock_Eventing.js"></script>
</head>
<body>
    <div class="widgetContents">
        <div class="panel-header">
            Announcing Clock
        </div>

        <div id="error-panel" class="error-panel">
        </div>

        <div class="panel-body">
            <div class="clock-frame">
                <span id="clock"> </span>
            </div>
        </div>

        <div class="button-panel">
        </div>

        <div id="message-panel" class="message-panel">
        </div>
    </div>
</body>
</html>
```

Notice that the following `<script>` tags have been added:

```
<script type="application/javascript" src="./owf-widget.js"></script>
<script type="application/javascript">
    // If not set, defaults to
/<context>/js/eventing/rpc_relay.uncompressed.html
    // OWF.relayFile = "/owf-sample-
html/js/eventing/rpc_relay.uncompressed.html";
</script>
```

Step 3 - Broadcast a message using the Eventing API

Replace the code in the `AnnouncingClock.js` file with the following:

AnnouncingClock.js

```
document.addEventListener("DOMContentLoaded", function() {
    OWF.ready(start);
});

function start() {
    updateClock();
    setInterval(updateClock, 1000);

    const msg = "Running in OWF: " + (OWF.Util.isRunningInOWF() ? "Yes" :
"No");

    document.getElementById("message-panel").innerHTML = msg;
    document.getElementById("message-panel").style.display = "block";
}

function updateClock() {
    // Format the time portion of the current date
    const timeString = new Date().toLocaleTimeString();

    // Update the time display
    document.getElementById("clock").firstChild.nodeValue = timeString;

    OWF.Eventing.publish("ClockChannel", timeString);
}
```

The `updateClock` function has been modified to publish the current time. See the code snippet below:

```
OWF.Eventing.publish("ClockChannel", timeString);
```

Once complete, any widget that subscribes to `ClockChannel` will receive messages broadcast from this widget. Once this widget is closed, the broadcast will stop.

Step 4 - Start the web server

Using the command line, navigate to the `node-server/` directory and run the following commands:

```
npm install
npm start
```

This will start the sample web server to statically host the widget files at http://localhost:5000/.

Step 5 - Add the Second Tracker and Announcing Clock widgets to OWF

To test the added Eventing API functionality, the widgets must be run from within OWF. Add the Second Tracker and modified Announcing Clock widgets to OWF via the OWF Admin page. For details on how to do this, see Chapter 3, *Adding a Widget to OWF*.

4.3. Additional Considerations

4.3.1. Channel Conventions

It is important to use a unique channel name so widgets are not accidentally published or subscribed to a pre-existing channel.

One approach is to use a hierarchical naming pattern with the levels of the hierarchy separated by a dot (`.`). To form a unique channel, prefix the channel name with a customer domain name reversing the component order. For example, if developing a widget for a company with the domain name of `mycompany.com`, the channel name's prefix would be `com.mycompany`. From that point, naming conventions for an individual's organization can be used to complete the channel name.

4.3.2. Payload Conventions: JSON Encoding

It is acceptable to directly encode the data broadcasted on Eventing channels as a simple string. This approach works when sending only a single variable. While a flat string would require the least amount of overhead, it leads to rigid code in the data, especially if the complexity of the sent data increases, because the code that parses the string may not be flexible. In that case, refactoring the message payload may break contract with established listening widgets.

Sending JSON objects with the data directly embedded is an approach that leads to considerably more flexible code. This process allows for the adding of additional data without having to re-code widgets that may not have been updated to communicate with the most current version of the broadcasting widget.

4.3.3. Payload Conventions: RESTful Data URIs

While simple strings and JSON objects will work well for many use cases, there are two situations in which widget developers can run into issues:

1. The information that is being sent has potential security concerns.
2. The size of information to be passed is large (such as a data set with hundreds of rows). Sending large quantities of information across the client browser can cause memory and performance issues.

The solution in both cases is to send a reference to the information rather than the information itself. The standardized best practice for sending said information is to send a REST URI encoded as a JSON object that contains the correct way to look up this information. The JSON object would then be parsed by the receiving widget and acted upon appropriately.

Currently, the standardized JSON object has only one field, dataURI. Later versions of this standard may contain additional fields. Adhering to this standard will ensure that other OWF-compliant widgets will be able to communicate effectively.

Sample JSON object containing a REST data URI

```
{
   "dataURI": "https://server/restful/path/to/object"
}
```

For a widget to make information available to other OWF widgets by exposing a REST API, it is important to guarantee that REST information will be accessible via cross-domain through AJAX calls. By default, many browsers will prevent such a call from succeeding and therefore developers must take explicit steps to make their application function correctly.

The recommended approach is to use the Cross-Origin Resource Sharing (CORS) standard, which allows the browser and the server to cooperate in a way which safely allows cross-origin calls to succeed. See the official CORS specification for details: http://www.w3.org/TR/cors/.

4.3.4. Eventing API Enhancements

OWF automatically enables the Eventing API and the Drag & Drop API. By automatically enabling the Eventing API, a widget activates when a user clicks inside it. Without it, users would have to click once to focus the widget and then click a second time to activate it. The Eventing API also activates Drag & Drop indicators. For example, a widget will activate when a user drags the mouse over it.

5. Preferences API

5.1. Overview

The Preferences API provides a convenient mechanism for the developer to store user specific data to the OWF database. A user preference is a `string` value that is uniquely mapped to a user, identified by name and namespace combination. The namespace should use a hierarchical naming pattern to avoid naming collision with other widgets. The value can be any string value, including serialized JSON.

In the tutorial below, a military time checkbox will be added to the Announcing Clock widget developed in Chapter 2, *Creating a Widget*. The state of this checkbox, whether it has been checked or not, is stored in a user preference.

5.2. Tutorial

Step 1 - Update the Announcing Clock HTML file

Replace the code in the `AnnouncingClock.html` file with the following:

AnnouncingClock.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Announcing Clock (Preferences)</title>

    <link type="text/css" rel="stylesheet" href="./css/widget.css">

    <script type="application/javascript" src="./owf-widget.js"></script>
    <script type="application/javascript">
        // If not set, defaults to
/<context>/js/eventing/rpc_relay.uncompressed.html
        // OWF.relayFile = "/owf-sample-
html/js/eventing/rpc_relay.uncompressed.html";
    </script>

    <script type="application/javascript"
src="./AnnouncingClock_Preferences.js"></script>
</head>
<body>
    <div class="widgetContents">
        <div class="panel-header">
            Announcing Clock
        </div>

        <div id="error-panel" class="error-panel">
        </div>

        <div class="panel-body">
            <div class="clock-frame">
                <span id="clock"> </span>
            </div>
        </div>

        <div id="button-panel" class="button-panel">
            <label for="checkboxMilitaryTime">Use Military Time:</label>
            <input id="checkboxMilitaryTime" type="checkbox"
onClick="setMilitaryTimePreference(this.checked);"/>
        </div>

        <div id="message-panel" class="message-panel">
        </div>
    </div>
</body>
</html>
```

Notice that the following `<script>` tags have been added:

```
<script type="application/javascript" src="./owf-widget.js"></script>
<script type="application/javascript">
    // If not set, defaults to
/<context>/js/eventing/rpc_relay.uncompressed.html
    // OWF.relayFile = "/owf-sample-
html/js/eventing/rpc_relay.uncompressed.html";
</script>
```

An `<input>` checkbox element has been added. Note the `onClick` event listener, which calls the method to save the preference when the user toggles the checkbox.

```
<input id="checkboxMilitaryTime" type="checkbox"
onClick="setMilitaryTimePreference(this.checked);"/>
```

Step 2 - Update the Announcing Clock JavaScript file

Replace the initial code in the `AnnouncingClock.js` file with the following:

AnnouncingClock.js

```
let militaryTime = false;

document.addEventListener("DOMContentLoaded", function() {
    OWF.ready(start);
});

function start() {
    updateClock();
    setInterval(updateClock, 1000);

    const msg = "Running in OWF: " + (OWF.Util.isRunningInOWF() ? "Yes" :
"No");
    document.getElementById("message-panel").innerHTML = msg;
    document.getElementById("message-panel").style.display = "block";

    document.getElementById("button-panel").style.display = "block";

    getPreference();
}
```

Since the widget is using the Eventing API, it must first wait until OWF has connected and the APIs are ready to use. This initial code waits for the page to be loaded, using the standard `DOMContentLoaded` event. Once the page is ready, the `OWF.ready` function is given a callback to execute when the widget is connected and ready.

Step 3 - Retrieve the Military Time preference

Add the following functions to the `AnnouncingClock.js` file:

```
function getPreference() {
    OWF.Preferences.getUserPreference({
        namespace: "com.mycompany.AnnouncingClock",
        name: "militaryTime",
        onSuccess: onGetPreferenceSuccess,
        onFailure: onGetPreferenceFailure
    });
}

function onGetPreferenceSuccess(pref) {
    militaryTime = pref.value === "true";

    document.getElementById("checkboxMilitaryTime").checked = militaryTime;
}

function onGetPreferenceFailure(error, status) {
    if (status !== 404) {
        OWF.Util.ErrorDlg.show("Got an error getting preferences! Status Code:
" + status + " . Error message: " + error);
    }
}
```

The `getPreference` function will asynchronously fetch the user's `com.mycompany.AnnouncingClock.militaryTime` preference.

After successfully retrieving the user preference, the `onSuccess` callback is called, passing the resulting preference object to the specified `onGetPreferenceSuccess` function. This uses the preference value to update the global `militaryTime` setting and update the checkbox.

The returned preference `value` is always a `string` and must be handled accordingly.

If a failure occurs, the `onFailure` callback will be called, passing an error message and status code to the new `onGetPreferenceFailure` function.

If the preference does not exist, a failure status code of `404` (Not Found), will be returned. This is a likely occurrence, as the user may not have saved their preference yet.

Step 4 - Update the Military Time preference

Add the following functions to the `AnnouncingClock.js` file:

```
function setMilitaryTimePreference(checkedState) {
    militaryTime = checkedState;

    OWF.Preferences.setUserPreference({
        namespace: "com.mycompany.AnnouncingClock",
        name: "militaryTime",
        value: checkedState,
        onSuccess: function() {},
        onFailure: onSetPreferenceFailure
    });
}

function onSetPreferenceFailure(error, status) {
    OWF.Util.ErrorDlg.show("Got an error updating preferences! Status Code: "
+ status + " . Error message: " + error);
}
```

When the user clicks the checkbox, the `setMilitaryTimePreference` function is called. The `OWF.Preferences.setUserPreference` API method is used to asynchronously create or update the preference with the following values:

- A `namespace` of `com.mycompany.AnnouncingClock`.
- A `name` of `militaryTime`.
- A `value` of either `true` or `false`, depending on whether or not the military time checkbox is checked.
- The `onSuccess` callback to execute if the user preference is successfully stored in the database.

Since no action is required under a successful completion in this tutorial, we are passing a no-op function.

- The `onFailure` callback to execute, `onSetPreferenceFailure`, that will show an error message dialog if a failure occurs.

Step 5 - Update the display to accommodate the Military Time preference

Replace the `updateClock` function in the `AnnouncingClock.js` with the following:

```
function updateClock() {
    // Format the time portion of the current date
    const timeString = new Date().toLocaleTimeString(undefined, { hour12:
!militaryTime });

    // Update the time display
    document.getElementById("clock").firstChild.nodeValue = timeString;
}
```

The current time will now be displayed in either regular or military time, depending on the state of the military time checkbox.

Step 6 - Start the web server

Using the command line, navigate to the `node-server/` directory and run the following commands:

```
npm install
npm start
```

This will start the sample web server to statically host the widget files at http://localhost:5000/.

Step 7 - Add the Announcing Clock widget to OWF

To test the added Preferences API functionality, the widget must be run from within OWF. Add the modified Announcing Clock widget to OWF via the OWF Admin page. For details on how to do this, see Chapter 3, *Adding a Widget to OWF*.

5.3. Preferences API Reference

The Preferences API comprises the following:

```
getUserPreference({
  namespace: "namespace",
  name: "name",
  onSuccess: onSuccessCallback,
  onFailure: onFailureCallback
});
```

```
setUserPreference({
  namespace: "namespace",
  name: "name",
  value: "value",
  onSuccess: onSuccessCallback,
  onFailure: onFailureCallback
});
```

```
deleteUserPreference({
  namespace: "namespace",
  name: "name",
  onSuccess: onSuccessCallback,
  onFailure: onFailureCallback
});
```

Each of these methods communicates with the server asynchronously and therefore requires the use of callback functions to provide the results of the requested operation.

For all three methods, the `onSuccess` callback should be a callback function that accepts one argument, a JSON object of the following structure:

```
{
  value: "true",
  path: "militaryTime",
  user: {
    userId: "testAdmin1"
  },
  namespace: "com.mycompany.AnnouncingClock"
}
```

In `getUserPreference`, this is the preference retrieved. In `setUserPreference`, this is the preference that has been created. And in `deleteUserPreference`, this is the preference that was deleted.

If a preference is not found in a call to `getUserPreference`, a different JSON object is returned to the `onSuccess` function, which looks like this:

```
{
  preference: null,
  success: true
}
```

If an error occurs, such as a `500` (Internal Server Error), the `onFailure` callback is executed. It is passes two arguments, as follows:

```
function onFailure(errorMessage, statusCode) {
  alert("Error (" + statusCode + "): " + errorMessage);
}
```

The `errorMessage` argument is a `string` describing the error, while `errorCode` is a `number` code indicating the HTTP error code returned by the server.

5.4. Additional Considerations

5.4.1. Preference Naming Conventions

In order to avoid name collisions with user preferences defined by other widgets, always use a hierarchical naming pattern with the levels of the hierarchy separated by a dot (`.`). To form a unique namespace, prefix the internet domain name, reversing the component order. For example, if developing a widget for a company with the domain name of `mycompany.com` then the namespace prefix would be `com.mycompany`. From that point, organizational naming conventions can be applied to the rest of the namespace.

5.4.2. Payload Conventions: JSON Encoding

To store several pieces of information, multiple user preferences can be created. As an alternative, they can be aggregated into one logical object, converted into a JSON string, and stored into one user preference. For example, consider storing a user's first, middle and last name. Using the first option would require the use of the following three user preferences:

```
com.mycompany.widget.firstName
com.mycompany.widget.middleName
com.mycompany.widget.lastName
```

Using the second option would require just one user preference using the following JSON string:

```
{
  "firstName": "John",
  "middleName": "Quincy",
  "lastName": "Adams"
}
```

Sending JSON objects with the data directly embedded is an approach that leads to considerably more flexible code. This process allows for the adding of additional data without having to re-code widgets that may not have been updated to communicate with the most current version of the broadcasting widget.

5.4.3. Payload Conventions: RESTful Data URIs

While simple strings and JSON objects will work well for many use cases, there are two situations in which widget developers can run into issues:

1. The information that is being sent has potential security concerns.
2. The size of information to be passed is large (such as a data set with hundreds of rows). Sending large quantities of information across the client browser can cause memory and performance issues.

The solution in both cases is to send a reference to the information rather than the information itself. The standardized best practice for sending said information is to send a REST URI encoded as a JSON object that contains the correct way to look up this information. The JSON object would then be parsed by the receiving widget and acted upon appropriately.

Currently, the standardized JSON object has only one field, dataURI. Later versions of this standard may contain additional fields. Adhering to this standard will ensure that other OWF-compliant widgets will be able to communicate effectively.

Sample JSON object containing a REST data URI

```
{
   "dataURI": "https://server/restful/path/to/object"
}
```

For a widget to make information available to other OWF widgets by exposing a REST API, it is important to guarantee that REST information will be accessible via cross-domain through AJAX calls. By default, many browsers will prevent such a call from succeeding and therefore developers must take explicit steps to make their application function correctly.

The recommended approach is to use the Cross-Origin Resource Sharing (CORS) standard, which allows the browser and the server to cooperate in a way which safely allows cross-origin calls to succeed. See the official CORS specification for details: http://www.w3.org/TR/cors/.

6. Launcher API

6.1. Overview

The Launcher API allows widget to send data to another widget. It is possible that one of the widgets sending or receiving the data does not have a user interface. Those widgets can be configured as background widgets (explained in the *OWF Administrator's Guide*).

This tutorial will go through the process of using the Launcher API by describing the behavior of the Channel Listener and Channel Shouter widgets. Channel Listener and Channel Shouter are example widgets included with OWF. In the tutorial below, the Channel Listener and Channel Shouter widgets work together to demonstrate both the Eventing and the Launcher APIs. The widget launching functionality is commented out by default. For this tutorial, uncomment that section of the code.

Channel Listener allows the user to subscribe to channels by entering the channel name into a text box and pressing the button on the widget. The widget will then display any messages which are broadcast on the channels to which it is subscribed. Channel Shouter allows the user to publish messages to specific channels by entering both the name of a specific channel as well as the message text into text boxes and pressing the button.

To demonstrate the Launcher API, if a user only has a Channel Shouter on their application, any message sent by the Channel Shouter will launch the Channel Listener. Once launched, the Channel Listener will be listening to the same channel and display the messages which get broadcast by the Channel Shouter.

6.2. Tutorial

Step 1 - Import the correct JavaScript Files

The example `ChannelListener.gsp` and `ChannelShouter.gsp` files use the maximum JavaScript import list.

The minimum required include list needed to use the Launcher API is described in the [required-includes] section.

Step 2 - Uncomment the Launcher API functionality

The Launcher API functionality is commented out by default. To use the sample, include the code in `ChannelShouter.gsp`.

Step 3 - Wrap the JavaScript that requires the Launcher API in the OWF.ready function

The Launcher API requires the use of Eventing. See `ChannelShouter.gsp` for example:

Example

```
OWF.ready(shoutInit);
```

Step 4 - Find the GUID for the widget to be launched

To launch a widget, the developer must know the widget's GUID. This tutorial determines the widget's GUID by querying the Preferences API using the findWidgets function shown below. This function retrieves a list of all widgets that a user has access to, including widget names and GUIDs. If the name of the widget is known, it is therefore easy to find the appropriate GUID, which can then be saved as a preference. The following code is searching for a widget named Channel Listener.

ChannelShouter.gsp

```
…
var scope = this;
shoutInit = owfdojo.hitch(this, function() {
  OWF.Preferences.findWidgets({
    searchParams: {
      widgetName: "Channel Listener"
    },
    onSuccess: function(result) {
      scope.guid = result[0].id;
    },
    onFailure: function(err) { /* No op */ }
  });
});
…
```

For additional ways to find the widget's GUID, see the below section Alternative Ways to Find a Widgets's GUID.

Step 5 - Launch the widget

To launch the widget, use the `launch` function on the OWF.Launcher object that was created as a result of `OWF.ready`. The `launch` function takes a configuration object which has four attributes:

`guid`

a `string` containing the unique ID of the widget to be opened.

`launchOnlyIfClosed`

a `boolean` flag which decides if a new widget should always be opened (`false`), or if the widget is already present to restore it (`true`).

`title`

a `string` that will replace the widget's title when launched.

`data`

a `string` representing an initial set of data to be sent only if a new widget is opened.
The data which is going to be sent must be passed as a string. In the example below, the data to be sent is a JavaScript object with two attributes – channel and message. Next, this object must be converted into a JSON string. This is accomplished by using the `OWF.Util.toString` utility function.

If the widget to be launched is already in an OWF application, the data value will not be sent.

In the code shown below, the widget to be launched is Channel Listener.

Example

```
shout = owfdojo.hitch(this, function () {
  var channel = document.getElementById("InputChannel").value;
  var message = document.getElementById("InputMessage").value;

  if (channel != null && channel != "") {
…
    if (scope.guid != null && typeopf scope.guid == "string") {
      var data = {
        channel: channel,
        message: message
      };
      var dataString = OWF.Util.toString(data);
      OWF.Launcher.launch({
         guid: scope.guid,
         launchOnlyIfClosed: true,
         title: "Channel Listener Launched",
         data: dataString
      }, function(response) {
…
    }
  }
}
```

Step 6 - Retrieve the initial data inside the launched widget

Once a widget has been launched, the widget may need to retrieve the initial set of data from the previous step. This is accomplished by using the `OWF.Launcher.getLaunchData()` function. This function will return the initial data from the previous step. In the `ChannelListenerPanel.js` code sample below, the data retrieved is a JSON string. This string is then parsed into a JavaScript object by using the `OWF.Util.parseJson` function. In `ChannelListenerPanel.js` the initial data sent is a channel to start listening on (`data.channel`), and an initial message to display on that channel (`data.message`).

Example

```
…
  render: function() {
    var launchConfig = OWF.Launcher.getLaunchData();
    if (launchConfig != null) {
      var data = OWF.Util.parseJson(launchConfig);
      if (data != null) {
        scope.subscribeToChannel(data.channel);
        scope.addToGrid(null,data.message,data.channel);
      }
    }
  },
…
```

6.3. Additional Considerations

6.3.1. Alternative Ways to Find a Widget GUID

6.3.1.1. Storing a Widget GUID as a Preference

An alternative way to determine which widget to launch is to store the GUID as a preference in the database using the Preferences API. The OWF Administration tools can be used to find the GUID of any widget. For the Channel Shouter/Channel Listener example, Channel Listener's GUID can be found by editing the Channel Listener widget using the Widget Editor. This will bring up a dialog that displays the GUID. The GUID should be saved under a newly created preference. The widget can then retrieve that GUID and used accordingly.

Example

```
OWF.Preferences.getUserPreference({
    namespace: "owf.widget.ChannelShouter",
    name: "guid_to_launch",
    onSuccess: function(result) {
        const guid = result.value;
        // Do something with guid...
    },
    onFailure: function(err) { /* No op */ }
});
```

6.3.1.2. Find a Widget by its Universal Name

Another way to determine which widget to launch is to search using its *universal name*. This can be done by querying the Preferences API using the `getWidget` function and including the `universalName` field in the parameters. This retrieves the specified widget's configuration details, including its GUID.

Example

```
OWF.Preferences.getWidget({
    universalName: "org.owfwebsite.owf.examples.NYSE",
    onSuccess: function(result) {
        const guid = result.value;
        // Do something with guid...
    },
    onFailure: function(err) { /* No op */ }
});
```

A widget's *universal name* is defined in its descriptor file. See Section 3.2.1, "Creating Descriptor Files for Widgets" for details on descriptor files.

6.3.2. Using Regular Expression to Change a Widget Title

The `launchWidget` function also accepts a `titleRegex` property. This property will be used as a replacement regular expression to alter the title. This allows the current widget title to be changed in complex ways. The example below appends text to the widget's title when it is launched.

Example

```
OWF.Launcher.launch({
  guid: someGuidVariable,
  title: "$1 - (Launched)",  // $1 represents the existing title
  titleRegex: /(.*)/,        // Matches and captures all text in the existing
title
  launchOnlyIfClosed: false,
  data: someDataString
}, onLaunchFailure);
```

7. Intents API

8. Remote Procedure Call (RPC) API

8.1. Overview

The Remote Procedure Call (RPC) API provides a method for direct, or point-to-point, interaction between widgets. This is different from the Eventing API, which allows widgets to globally publish messages or subscribe to a subset of globally published messages from any widget. In contrast, RPC follows a client-server model and the calls cannot be received by any widget other than the intended target. The "server" widget registers a list of functions with OWF which may then be called by "client" widget. This process is facilitated by a widget proxy object, which exposes the registered functions on the server widget to the client widget.

OWF ships with two example widgets, the Color Server and Color Client, which demonstrate the functionality of the RPC API. In the tutorial below, simple versions of the Color Server and Color Client widgets will be recreated.

8.2. Tutorial

Step 1 - Create the Color Server widget HTML file

Create the `ColorServer.html` file and add the following code:

ColorServer.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Color Server</title>
    <script type="application/javascript" src="./owf-widget.js"></script>
    <script type="application/javascript" src="ColorServer.js"></script>
</head>
<body id="body" style="background-color: white;">
    <h1>Color Server</h1>
</body>
</html>
```

Step 2 - Register the Color Server RPC functions

Create the `ColorServer.js` file and add the following code:

ColorServer.js

```
document.addEventListener("DOMContentLoaded", function() {
    OWF.ready(start);
});

function start() {
    OWF.RPC.registerFunctions([
        {
            name: "getColors",
            fn: getColors
        }, {
            name: "changeColor",
            fn: changeColor
        }]
    );
}
```

The initial code waits for the page to load using the standard `DOMContentLoaded` event. Once the page is ready, the `OWF.ready` API method is used to wait for the widget to connect to the framework. When the widget is ready, the `start` callback function is executed.

The `start` function uses the `OWF.RPC.registerFunctions` API method to register a list of functions which will be made available to other widgets.

Step 3 - Add the Color Server functionality

Add the following code into the `ColorServer.js` file:

ColorServer.js

```
function getColors() {
    return ["Red", "Blue", "Yellow"];
}

function changeColor(color) {
    document.getElementById("body").style.backgroundColor = color;
    return true;
}
```

The `getColors` function returns a list of supported colors.

The `changeColor` function updates the background color of the widget, using the color `string` argument.

Using the `OWF.RPC.registerFunctions` API method described in the previous step, these functions will now be available for the Color Client to call.

Step 4 - Create the Color Client widget HTML file

Create the `ColorClient.html` file and add the following code:

ColorClient.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Color Client</title>
    <script type="application/javascript" src="./owf-widget.js"></script>

    <script type="application/javascript" src="ColorClient.js"></script>
</head>
<body>
    <h1>Color Client</h1>
    <button type="button" onclick="getColors()">List Colors</button>
    <select id="colors" onclick="setColor(this.value)"></select>
    <div></div>
    <div id="errors" style="color:red; font-weight:bold; display:none;"></div>
</body>
</html>
```

Step 5 - Find the widget proxy

Create the `ColorClient.js` file and add the following code:

ColorClient.js

```
function getColorServerProxy(callback) {
    setError();

    OWF.getOpenedWidgets(function(widgets) {
        if (!widgets) {
            setError("Failed to get opened widgets");
            return;
        }

        const colorServerId = findWidgetIdByName(widgets, /^.*Color
Server.*$/);
        if (!colorServerId) {
            setError("Color Server widget not found");
            return;
        }

        OWF.RPC.getWidgetProxy(colorServerId, function(proxy) {
            if (!proxy) {
                setError("Failed to get widget proxy for Color Server");
                return;
            }
            callback(proxy);
        });
    });
}
```

In order to call the registered RPC functions on the Color Server, the `OWF.RPC.getWidgetProxy` API method is be used to get a widget proxy instance. Since the widget ID is required, `OWF.getOpenedWidgets` is used to get the list of open widgets which is then searched using the below helper functions to find the Color Server.

Add the following code for the `findWidgetIdByName` and `setError` helper functions:

ColorClient.js

```
function findWidgetIdByName(widgets, regexp) {
    for (let i = 0; i < widgets.length; i++) {
        let widget = widgets[i];
        if (widget.id != null && widget.name.match(regexp) != null) {
            return widget.id;
        }
    }
    return null;
}

function setError(message) {
    const errors = document.getElementById("errors");
    errors.style.display = message ? "block" : "none";
    errors.innerText = message || "";
}
```

Step 5 - Call the Color Server RPC functions

Add the following code into the `ColorClient.js` file:

ColorClient.js

```
function getColors() {
    getColorServerProxy(function(colorServer) {
        colorServer.getColors(updateColorChoices);
    });
}

function setColor(color) {
    getColorServerProxy(function(colorServer) {
        colorServer.changeColor(color);
    });
}
```

Using the previously created `getColorServerProxy` function, the two registered functions are called to get the list of colors and to set a new selected color.

This example code purposely does not save the widget proxy between calls. If the widget is closed or otherwise becomes unavailable, the proxy would become invalid. Finding the widget and proxy "on-demand" ensures that the target widget is available to receive the RPC calls.

Finally, add the following to update the combo box with the new color options:

ColorClient.js

```
function updateColorChoices(colors) {
    const selectBox = document.getElementById("colors");

    // Remove previous `option` elements
    for (let i = selectBox.options.length - 1; i >= 0; i--) {
        selectBox.remove(i);
    }

    // Create an `option` element for each color
    for (let i = 0; i < colors.length; i++) {
        const color = colors[i];
        const option = document.createElement("option");
        option.text = color;
        option.value = color;
        selectBox.add(option);
    }
}
```

Step 6 - Start the web server

Using the command line, navigate to the `node-server/` directory and run the following commands:

```
npm install
npm start
```

This will start the sample web server to statically host the widget files at http://localhost:5000/.

Step 7 - Add the Color Server and Color Client widgets to OWF

To test the RPC API functionality, the widgets must be run from within OWF. Add the new Color Server and modified Color Client widgets to OWF via the OWF Admin page. For details on how to do this, see Chapter 3, *Adding a Widget to OWF*.

9. Chrome API (Unsupported)

The Chrome API is not currently supported in OWF v8.

10. Drag & Drop API (Unsupported)

The Drag & Drop API is not currently supported in OWF v8.

11. Logging API (Unsupported)

The Logging API is not currently supported in OWF v8.

It is recommended that widget developers use the native Web browser logging capabilities, such as `console.log`, `console.warn`, and `console.error`.

12. State API (Unsupported)

The State API is not currently supported in OWF v8.

Glossary

Accordion (layout)

Display widgets in equal, horizontal panes that do not scroll (each individual widget may scroll using its own scroll bar).

Affiliated Store

A store that another organization uses for their system. When a local store is connected to an affiliated store, users in the local store can search for and add listings from the affiliated store (assuming the user has proper authentication for the affiliated store).

App

Deprecated term for a Stack.

App Component

Deprecated term for a widget.

Dashboard

An organized collection of widgets with a customizable layout.

Filters

A feature used to reduce the number of search results by type or category.

Fit (layout)

Allows a user to place a single widget on the screen.

Help

Repository of instructional guides and video tutorials.

Intent

Instructions for carrying out a widget’s intentions.

Listing

Any software dashboard or widget that a user enters into the Store is called a "Listing." Listings can be a various types of Web content.

Marketplace

A searchable catalog of shared listings of widgets and dashboards (also referred to as the Store).

OWF

Abbreviation for Ozone Widget Framework.

Pages

Deprecated term for a dashboard.

Portal (layout)

A column-oriented layout that organizes widgets of varying heights. Each new widget loads above the first one on the screen. The user drags a dividing bar to specify widget's height. The widgets and the Ozone window scroll.

Required Listings

An association between Listings. *Example: if Listing A needs Listing B to function, Listing B is a Required Listing.*

Stack

A collection of Dashboards (pages). Allows administrators and users to group Dashboards into folder-like collections that allow for easy transition from one to another.

Store

Commonly used term for the Ozone Marketplace.

Tabbed (layout)

Display one widget per screen, with tabs the top of the screen to switch from one widget to another.

Toolbar

The navigation bar at the top of the application. It links to a user's stacks, widgets, the Store, online Help and options from the drop-down User Menu.

User

A person signed into the Ozone application, usually referring to a person without administrative privileges.

Widget

A light-weight, single-purpose Web application that offers a summary or limited view of a larger Web application and may be configured by the user and displayed within a Dashboard.

Widget Menu

The Widgets Menu displays all available widgets. Use this feature to start or add widgets to a dashboard.

Appendix A: Java WAR Deployment

A.1. Walkthrough

Step 1 - Create the Servlet directory structure

1. Create a project directory. (e.g. `webapp/`).

   Web applications using Java and the Servlet specification follow a layout of directories and files. The root of the hierarchy defines the document root of the Web Application. In this walk-through, the root directory will be the `webapp/` directory.

2. Create the `WEB-INF/` directory under the `webapp/` directory.

   All files under the project `webapp/` directory can be served to the client, except for files under the special directory `WEB-INF/`. The `WEB-INF/` directory is not part of the public document tree of the application, and may contain resources required for running the Web application.

3. Create the `web.xml` file in the `WEB-INF/` directory, and add the following:

   Example: webapp/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
         <display-name>Announcing Clock Widget</display-name>
</web-app>
```

Step 2 - Create the Announcing Clock widget

The complete example files are available under the `ozone/docs/developers_guide/examples/` directory.

Create the following files in the `webapp/` directory:

AnnouncingClock.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Announcing Clock</title>
    <script type="application/javascript" src="./AnnouncingClock.js"></script>
</head>
<body>
    The time is: <span id="clock"> </span>
</body>
</html>
```

AnnouncingClock.js

```
document.addEventListener("DOMContentLoaded", start);

function start() {
    updateClock();
    setInterval(updateClock, 1000);
}

function updateClock() {
    // Format the time portion of the current date
    const timeString = new Date().toLocaleTimeString();

    // Update the time display
    document.getElementById("clock").firstChild.nodeValue = timeString;
}
```

Step 3 - Create a .WAR file

A `.war` file is a Web application compressed into a single file. While directories and files can be copied directly onto the Web server, it is easier and more common to use a WAR file.

To create the WAR file for the Announcing Clock Widget, open a command prompt and navigate to the previously created `webapp/` directory. Then, run the following command:

```
jar cvf announcing-clock.war
```

Step 4 - Deploy the .WAR file to the server

The deployment method used depends on the Web application server. For the prepackaged OWF Tomcat server, the process is as simple as copying the WAR file into the `tomcat/webapps/` directory on the Web

application server. In the event that a particular application server has different requirements, the appropriate Java application server documentation should be consulted for information on how the WAR file should be deployed.