Developer Guide

Published: April 16, 2015
Version: 10.0.0.1

Table of Contents

1. Introduction

1.1. Overview

This document explains specific configurations and APIs that will enhance the development and use of OZONE.

1.1.1. Organizational Benefits

Organizations can use OZONE to share applications; this partnership can reduce costs and redundancies for partnering organizations. OZONE gives users the tools to create and share applications. Application owners and stewards decide who can see and use those tools. When users search for applications, the search results can include applications from other organizations. This allows OZONE users to save and use other organization's applications. This negates the need for multiple organizations to allot valuable time and resources developing redundant tools. This same principle holds true within organizations that segment information to sub-organizations.

2. Build Instructions

This section explains the steps a developer must complete to start an OZONE environment using vagrant. It explains how to clone, compile, configure and launch all OZP projects on a virtual machine. The product uses an independent Elasticsearch instance (It does not use an H2 in-memory database or in-memory Elasticsearch). The front-end resources are hosted at nginx which is also used as a reverse proxy so that all front- and back-end resources are served by the same domain. When the virtual box initially starts, it executes `bootstrap.sh` . This runs 4 scripts to align the development VM with the production version of the software (though, some differences must be addressed during deployment). The scripts run by bootstrap are:

- `initial_provisioning.sh`: Installs dependencies (Java, Grails, MySQL, etc.), configures dependencies, and copies insecure SSL keys for use by nginx and Tomcat. (Unlike the other three scripts, this one should only run once.)
- `build.sh`: Clones the ozp applications from GitHub and builds them.
- `package.sh`: Tars and compresses the compiled output from `build.sh`.
- `deploy.sh`: Moves the artifacts to the right places and perform any configuration required for deployment.

To begin, review the main project on Github at https://github.com/ozone-development.

2.1. Build and Deploy on a Virtual Machine

Prerequisites

Install Vagrant and VirtualBox on your local machine.

- Vagrant (Use version 1.6 or newer; Vagrant documentation)
    - *Optional* — Install `vagrant-vbguest` to keep your Guest Additions up-to-date:

```
vagrant plugin install vagrant-vbguest
```

- VirtualBox (VirtualBox documentation)

> **NOTE**
> Add or confirm that an SSH client (like Cygwin, MinGW or Git) is on your PATH.

Instructions

1. Clone the dev-tools repo:

```
git clone https://github.com/ozone-development/dev-tools
```

> **NOTE**
> If you're using Cygwin, cd to `cygdrive/c/yourFolder` before cloning.

2. Change directories:

```
cd to dev-tools/vagrant/centos6.6/build-and-deploy-box
```

3. Next, you have to customize some files.
    - Open `Vagrantfile`, change line 52 and change it to match your path. The original line is:

```
config.vm.synced_folder "~/ozp/", "/home/vagrant/ozp", type: "rsync",
rsync__exclude: "**/*.git/"
```

    - Open `deploy.sh`, change line 7 to your IP address:

```
HOST_IP="10.10.XX.XX"
```

4. Open a command line window and enter:

```
vagrant up
```

> NOTE
> This may take 30 minutes to an hour to complete.
> To capture errors on Linux and OS X systems, run `vagrant up 2>&1 | tee vagrant.log`

   a. OPTIONAL: If any of the apps do not install properly, run this after `vagrant up`:

```
vagrant ssh -c "/vagrant/initial_provisioning.sh; /vagrant/build.sh;
/vagrant/package.sh; /vagrant/deploy.sh;"
```

5. If you plan to only run OZP locally, this step is unnecessary. If you commit code changes to https://github.com/ozone-development, you must update ~/ozp. By default, the contents of ~/ozp on your host rsync with the vagrant box. This is useful for development, where the code you edit needs to get from your host machine to the VM for execution. Run `vagrant rsync` to update the contents of `/home/vagrant/ozp` in the VM.
6. The Vagrant box should have built, packaged and deployed the following OZP applications. In a browser window, change "localhost" to your IP address and try to connect to:
   a. https://localhost:7799/center
   b. https:// localhost:7799/hud
   c. https:// localhost:7799/webtop
   d. https:// localhost:7799/iwc/debugger.html
   e. https:// localhost:7799/marketplace/api
   f. https:// localhost:7799/demo_apps
   g. https://localhost:7799/manager/html/ (username: tomcat, password: password)

   Except for the tomcat manager, use the following basic authentication to access OZP:
   - testUser1, password (role: User)
   - testAdmin1, password (role: Marketplace Steward)
   - testOrgSteward1, password (role: Organizational Steward)

2.1.1. Potential Issues

This section troubleshoots some issues you may experience:

- Firewalls may cause problems. If the pages do not load in the browser, run the following in the command window: `sudo service iptables stop`
- `deploy.sh` waits two minutes for ozp-rest to start. On slow or resource-starved machines, this may not be enough time.
- After changing the rsync directory in `Vagrantfile`, do NOT rsync a directory that includes rsync.
- If your host machine runs Windows, the `vagrant ssh –c…` command may not work the first time. Try running it a second time if the pages do not load in the browser, it is:

```
 vagrant ssh -c "/vagrant/initial_provisioning.sh; /vagrant/build.sh;
/vagrant/package.sh; /vagrant/deploy.sh;"
```

- On a Windows machine, you may run into errors if your SSH client automatically converts new lines in the `sh` files to Windows format. To fix the problem; install and run `dos2unix` (command *.sh) or a similar utility after you cd to the …/build-and-deploy-box directory in step 2.

2.2. Helpful Hints

- nginx error log: `/var/log/nginx/error.log`
- nginx access log: `/var/log/nginx/access.log`

- restart nginx: `nginx -s reload` or `sudo service nginx restart`
- Logs (backend): /var/log/tomcat/, /usr/share/tomcat/logs/
- verify that elasticsearch is running: `sudo netstat -lnp | grep 9300`
- sudo multitail -i /usr/share/tomcat/logs/marketplace.log -i /usr/share/tomcat/logs/stacktrace.log -i /var/log/tomcat/localhost_access_log.2015-04-04.txt -i /var/log/tomcat/catalina.out -i /var/log/tomcat/localhost.2015-04-04.log -i /var/log/nginx/error.log -i /var/log/nginx/access.log -s 2 -sn 4
- # multitail: F1 for help, O to clear all windows, o to clear specific window, 0..9 to set a marker in the correspond window to easily see what's changed, B to merge all into one
- from Modern IE VMs, http://10.0.2.2 to get to host box. This means we need to adjust the paths in OzoneConfig.js to use 10.0.2.2 instead of localhost
- IE testing: Always enable Cache->Always Refresh From Server option in F12 dev tools
- clear the elasticsearch data: curl -XDELETE 'http://localhost:9200/marketplace' (done on local box, not from host (would need port fwd))
- # elasticsearch data at /var/lib/elasticsearch

3. Marketplace Configurations (Center)

3.1. MarketplaceConfig.groovy

3.1.1. Change the Public URI

You can make the store publicly accessible using the `marketplace.publicURI` configuration. The URI (uniform resource identifier) can be useful if the ozp-rest backend is behind a load balancer or reverse proxy, and the base URI as calculated by the ozp-rest backend itself is not correct.

To change the Public URI:

1. Open `\apache-tomcat-7.0.55\lib\MarketplaceConfig.groovy`
2. Change the value for `marketplace.publicURI`. This value will be used as the base URI in the construction of all URIs within the resources representation returned by the server. If the value is null or not set, the automatically-calculated base URI for each request will be used.

3.2. Image Servers

3.2.1. Configure Image Server backend location and file types

This section explains how to change the location where images are stored and change the types of images that are accepted on the Listing Create/Edit Form.

Listings include several icons and screenshots as part of their metadata. Listing owners upload these images on the Listing Create/Edit Form. By default, the images are stored on the OZONE REST server backend file system, this setting is configured in the following file:

```
\apache-tomcat-7.0.55\lib\MarketplaceConfig.groovy
```

To change the location where images are stored, use the following property which is commented out by default in `MarketplaceConfig.groovy`:

```
marketplace.imageStoragePath = "${System.properties['catalina.home']}/images"
```

- This property should contain an absolute path to a directory where the images are stored.
- The OZONE REST server must have read and write permissions in this directory, including the ability to create sub-directories.
- To use a path relative to the running server's working directory, the appropriate system properties can be inserted into the path.
  For example:
  The default value of the `marketplace.imageStoragePath` property is `${System.properties['catalina.home']}/images`.
  For Tomcat servers, this will result in images being placed in an `images` directory within the main Tomcat directory.
  Note: To cluster, you must configure an external mechanism (like a shared network drive) to allow all servers in the cluster to use the image storage directory.

3.2.2. Purging the system of unnecessary images

When you create or edit a listing, the system uploads its images to the server in separate calls. If someone removes images from a listing or deletes a listing, the unused images will be deleted by a daily OZONE REST server job. The job deletes unused images that are at least one day old.

3.2.3. Acceptable types of images

To configure the types of images that the server accepts:

1. Open `\apache-tomcat-7.0.55\lib\MarketplaceConfig.groovy`
2. Use the `marketplace.acceptableImageTypes` configuration to add or delete file types. The values must be strings that are used as the file extension for the stored image files.

> NOTE
> This option must be a map object where the keys are Internet Media Type strings that the server can accept. Internet Media Types that do not start with "image/" are rejected in any setting.

3. Save the file and restart `\apache-tomcat-7.0.55\bin\startup.bat` (or `startup.sh`)

Example

As an example, the default value of `marketplace.acceptableImageTypes` is as follows:

```
marketplace.acceptableImageTypes = [
    'image/png': 'png',
    'image/jpeg': 'jpg',
    'image/webp': 'webp',
    'image/svg+xml': 'svg'
]
```

This configuration accepts PNG, JPEG, WebP and SVG images that will be saved with the file extensions .png, .jpg, .webp and .svg, respectfully.

3.3. Add users and stewards to a test implementation

In an implementation, the system's security determines users' roles. In a development environment, you may want to create various users and roles in order to test the software. Never use these instructions to create users in a production environment because the passwords are not stored safely or encrypted. That said, use the following instructions to create test users in a development system:

1. From your `Apache-tomcat directory`, open the `lib` directory.
2. In the `lib` directory, open the `users.properties` file.
3. Add users and roles based on the following examples:

```
testUser1=password,ROLE_USER,Test User 1,Test 1
Organization,testUser1@nowhere.com,[group1;I am a sample Group 1 from
users.properties;test@gmail.com;active]

testSteward1=password,ROLE_USER:ROLE_ORG_STEWARD,Test Steward 1,Test 1
Organization,testSteward1@nowhere.com

testAdmin1=password,ROLE_ADMIN:ROLE_USER,Test Admin 1,Test Admin
Organization,testAdmin1@nowhere.com,[group1;I am a sample Group 1 from
users.properties;test@email.com;active]
```

> NOTE
> See the Steward section of the User and Steward Guide for further explanation of roles.

4. Save changes to the `users.properties` file.

5. Restart `startup.bat` or `startup.sh` in the `Apache-tomcat-7.0.55\bin` directory. Each new user will appear in the system after you log in with it.

4. OzoneConfig.js

From `OzoneConfig.js` you can configure the URLs for API, Help, Metrics, Center, HUD, Webtop, feedback, developer resources, etc. Webtop, Center and HUD each have their own `OzoneConfig.js`. While the files exist independently, they should reference many of the same URLs because users will expect to go to the same addresses from the Global Toolbar no matter where they are in the application. For example, clicking Help in HUD should take them to the same Help location that Help takes them to in Webtop. To do this, you will need to configure all three `OzoneConfig.js` files.

`OzoneConfig.js` locations

- Marketplace/Center – `/apache-tomcat-7.0.55/webapps/center`
- HUD – `/apache-tomcat-7.0.55/webapps/hud`
- Webtop – `/apache-tomcat-7.0.55/webapps/webtop`

5. Inter-Widget Communications (IWC)

5.1. Overview

The Inter-Widget Communications (IWC) is the communications bus within the browser. It allows listings in OZONE to communicate solely within a browser window, which reduces network latency, improves reaction time and addresses some server security concerns.

The IWC is composed of two components, a client and a bus:

- **Client**

  The communication module used in application code to interface with other applications.
- **Bus**

  The connection module that client's communicate through. The bus is made up of deployment-configurable modules. The bus is configured and deployed to all clients using a common URL. This does not mean the bus is remote. The bus runs locally to the end-user and is gathered via a common location to allow clients to communicate through a common origin.

5.2. Add an IWC Client to an Application

An IWC client is an application's connection to an IWC bus. This section demonstrates how to add an IWC bus to a JavaScript application. It assumes you already built IWC. It includes instructions for connecting to an externally hosted bus (i.e. a remote server hosts the IWC components and database connections).

1. Use Bower to gather the IWC client with the following command:

   `bower install ozone-development/ozp-iwc`
2. To add the client module to an application, include the `ozpIwc-client.min.js` script from the `/dist` directory:

   `<script src="../bower_components/ozp-iwc-/dist/js/ozpIwc-client.js"></script>`

> NOTE
>
> To use a configured IWC bus, refer to the IWC deployment guide. It provides an example configured bus with the distributed IWC via GitHub.

Example

An IWC bus is always local to the user, but bound by its host domain. For this example we will be using a predefined IWC bus hosted on the ozp-iwc github page ozone-development.github.io/ozp-demo/

Our example application for this guide is a simple journal application. This application, when connected to an IWC bus, can store/share/edit entries with other applications.

1. Load the IWC-client library
   a. To use HTML: Load in the IWC-client library first to expose it to our application JavaScript `app.js`.

```
<html>
    <head>
        <script src="../bower_components/ozp-
iwc/dist/js/ozpIwc-client.min.js"></script>
        <script src="js/app.js"></script>
```

```
    </head>
```

b. In JavaScript: The IWC library is encapsulated in the `ozpIwc` namespace.

2. Connect your IWC client to an IWC bus
   a. To use HTML: Enter `client = new ozpIwc.Client(...)`.
   b. In JavaScript: In the application code, an `ozpIwc` client is instantiated and connects to the IWC bus hosted on the GitHub pages

```
var client = new ozpIwc.Client({
    peerUrl: http://ozone-development.github.io/iwc
});
```

On instantiation, the client will asynchronously connect. Any client calls made prior to the client's connection will be queued and run once connected.

To perform operations bound by the client connection, the `connect` promise can be called.

```
var client = new ozpIwc.Client({
    peerUrl: http://ozone-development.github.io/iwc
});

client.connect().then(function(){
   ... // connection dependent code
});
```

Once connected (asynchronously), you can obtain a client address. This is indication that the application has connected to the bus.

```
var client = new ozpIwc.Client({
    peerUrl: http://ozone-development.github.io/iwc
});

client.connect().then(function(){
    console.log("Client connected with an address of: ",
client.address);
});
```

3. Install IWC Client Files
You may install the client file using Bower or manually. However, for the remainder of this section, the IWC client will be referenced at the Bower installation location.
   a. Using bower From your project's directory, run `bower install ozone-development/ozp-iwc`
   The IWC Client file that you will reference from your application is at:
   `bower_components/ozp-iwc/dist/js/ozpIwc-client.min.js`
   b. Manually
   Download the latest distribution of the IWC client from the IWC GitHub repository at: https://raw.githubusercontent.com/ozone-development/ozp-iwc/master/dist/js/ozpIwc-client.min.js and place it in your project's directory. You may put the file in any location you want as long as it can be located from within the HTML.

5.3. Connecting to the Bus

5.3.1. Connecting

To use IWC between applications, you must add the IWC client connections to each listing or "client." The clients connect to an IWC bus that is bound by the browser and the domain where it is obtained.

An application or client that includes this example code will connect via IWC to the bus on domain `http://ozone-development.github.io/iwc`. The system gathers the JavaScript for the bus from the URL. From there, it runs locally enclosed in the same domain.

All aspects of the client use promise to simplify integration with asynchronous applications.

```
var client = new ozpIwc.Client({
    peerUrl: "http://ozone-development.github.io/iwc"
});
client.connect().then(function(){
   /* client use goes here */
});
```

5.3.2. Disconnecting

To disconnect an application or "client" from the IWC bus, call `disconnect()` as shown in the following example:

```
var client = new ozpIwc.Client({
    peerUrl: "http://ozone-development.github.io/iwc"
});

client.connect().then(function(){
    client.disconnect();
});
```

5.4. Making IWC API calls

Each deployed version of the IWC bus is capable of having its own specified API's integrated with it. For the purpose of this guide, the IWC bus used by the journal application has a `data.api` component.

The `data.api` is a key/value storage component on the IWC bus. Here, common resources can be shared among applications via the `set` and `get` actions.

API's on the IWC bus inherit from a common API base and all share a common set of `actions` of which they can expand on.

5.5. Available APIs

The IWC client code does not depend on the IWC bus. When the client connects to a bus it becomes aware of the APIs available on that bus and configures function calls for those APIs.

This code demonstrates a console print out of:

- available APIs

- the actions those APIs handle
- the client's assigned function name

```
var client = new ozpIwc.Client({
    peerUrl: "http://ozone-development.github.io/iwc"
});
client.connect().then(function(){
var apis = client.apiMap;
    for(var api in apis) {
        console.log("Api: ", apis[api].address, "Actions: ",
apis[api].actions,
                    "Function: ", apis[api].functionName);
    };
```

As an example, an output of this code may yield:

```
Api: data.api
Actions: "get", "set", "delete", "watch", "unwatch", "addChild",
"removeChild",
         "list"
Function: data
```

This means the client can `get` a resource from the `data.api` by calling:

`client.data().get("/some/resource");`

This informs the developer of a specific bus's available functionality.

5.6. API Requests

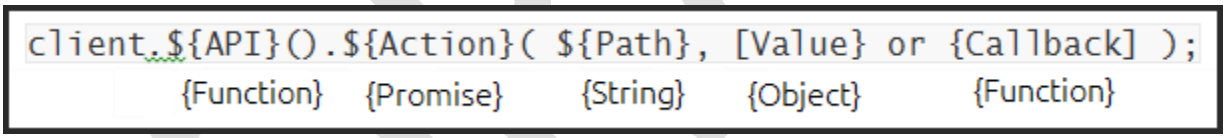

Figure 1: API Request Example

Use the following structure to make an API request:

`client.${API}().${Action}( ${Path}, [Value} or {Callback] );`

- **API** {Function}
  The API that the client calls, i.e. `data(...)`
- **Action** {Promise}
  The action that the client performs, i.e. `set(...)`
- **Path** {String}
- The path of the stored resource pertaining to the action, i.e. `"/foo"`
- **Value (Optional):** {Object}
- The object sent for the API to operate on, i.e. `{ bar: "buz"}`
- **Callback (Optional):** {Function}
- A callback function used for recurring actions. This applies to actions like watch and register.

Since some actions do not require values passed (read actions) and some actions do not require callbacks, the Value and Callback parameters are optional.

5.6.1. Valid Structures

Use the following structures to make action calls:

```
client.${API}().${Action}( ${Path} );
client.${API}().${Action}( ${Path}, [Value] );
client.${API}().${Action}( ${Path}, [Callback] );
client.${API}().${Action}( ${Path}, [Value], [Callback] );
```

The following is **not** a valid structure:

```
client.${API}().${Action}( ${Path}, [Callback], [Value] );
```

5.7. API Responses

Since the IWC operates asynchronously, all requests receive a response sent back to a client that resolves the action's promise.

The promise structure is as follows:

```
client.${API}().${Action}(...).then(function(res){...});
```

The `then` of the promise receives a formatted response object. The contents of the response **res** are as follows:

- **response** {String}
  The result of the request; "ok" means a successful operation.
- **src** {String}
  The sender of the message.
- **dst** {String}
  The receiver of the message.
- **entity** {Object | String}
  The payload of the response. If an action should return data, it will be in this object.

> NOTE
> This guide uses the `then` and `catch` resolution/rejection's of promises only as needed to demonstrate use. Others may be usable. All actions will resolve/reject with the response received from the bus.

5.7.1. Response Example

The variable `foo` contains the value stored at `/foo` once the API request receives its response.

```
var dataApi = client.data();
var foo;

dataApi.get('/foo').then(function(res){
    foo = res.entity;
});
```

The value of **res**, the resolved object of the list request, is formatted as follows:

```
{
```

```
    "response":"ok",
    "src":"data.api",
    "dst":"4e31a811.31de4ddb",
    "entity":{
        'bar': "buzz"
    },
    "ver":1,
    "time":1424897169456,
    "msgId":"i:1397"
    "replyTo":"p:704",
}
```

- **response**
  Seeing a response of 'ok' indicates that the request was handled without error.
- **src**
  The origin of the response. With the request to get the resource `/foo` sent to the Data API, the Data API module generated and sent a response.
- **dst**
  The destination of the response. Each IWC client is assigned a unique address local to the IWC Bus. This address designates who should receive the data being transmitted.
- **entity**
  The value of the resource. In this case, `/foo` holds `{'bar': "buzz"}`
- **ver**
  The version of the resource. Whenever the value of `/foo` changes, **ver** will increment.
- **time**
  Epoch time representation of when the response was generated.
- **msgId**
  Each data transmission through the IWC is labeled with a unique message identifier. The bus keeps track of message identifiers so that components know to whom they should reply.
- **replyTo**
  The message identifying the request that the response was sent.

5.8. Error Handling

Not all requests are valid. In the event that a request cannot be handled, or should not be allowed, the promise will reject. This allows for a clean separation for error handling. The value of `errRes` follows the same format of a valid API response (see 5.7: API Responses).

5.8.1. Error Example

In this example a get request is sent, but no resource was specified in the **get** action call (`dataApi.get()`)

```
var dataApi = client.data();
var foo;
```

```
dataApi.get().then(function(res){
    foo = res.entity;
}).catch(function(errRes){
    // handle the error here.
});
```

Because of this, the response will be in the `catch` because it was a failed request.
Further information about why the request failed can be found in the **errRes** `response` field.

NOTE
If supporting IE 8, `catch` is a keyword and cannot be used. In this situation, replacing `catch` with `['catch']` will prevent IE 8 from failing. See snippet below:

```
dataApi.get().then(function(res){
    foo = res.entity;
})['catch'](function(errRes){
    // handle the error here.
});
```

5.9. API Usage Example

This snippet is an example of using the `Data API` through an already connected client.

```
var dataApi = client.data();
var foo = { 'bar': 'buz' };

dataApi.set('/foo',{ entity: foo});           //(1)

dataApi.get('/foo').then(function(res){       //(2)
    //res has /foo's value
});

dataApi.watch('/foo',function(response,done){ //(3)
    //when I'm done watching I call done
    done();                                   //(4)
});
```

1. A **set** action is performed on the resource `/foo` of the Data API. After this action completes, the value stored at `/foo` will equal `{'bar': 'buz'}`.
2. A **get** action is performed on the resource `/foo` of the Data API. In the **resolution** of this action, the value of `/foo` will be available in the variable `res` in the following format:

```
{
    'entity': {
        'bar' : 'buz'
```

```
    }
}
```

3. A **watch** action is performed on the resource `/foo` of the Data API. In the `callback` of this action, a report of the value change of `/foo` will be available in the variable `response` in the following format:

```
{
    'entity': {
        'newValue': ${New value of /foo},
        'oldValue': ${Previous value of /foo},
    }
}
```

4. Based on the changed `/foo` value, the watch may no longer be necessary. Because of this a `done` parameter is available to the callback. Calling `done()` will cause the callback to be unregistered.

6. IWC Core APIs

The IWC Bus defaults to having the following four APIs:

Data API

A simple key/value JSON store for sharing common resources among applications; It can use backend connections for persisting storage in listings.

Intents API

Handling for application intents; Follows the idea of android intents; Allows applications to register to handle certain intents (ex. graphing data, displaying HTML). Like android, the IWC Intents API give the user a dialog to choose what application should handle the request.

Names API

Status of the bus. This API exposes information regarding applications connected to the bus.

System API

Application registrations of the bus. This API makes bus connections aware of applications the bus can identify. Different then the names API, these applications are not the current running applications, rather these are registrations of where applications are hosted and default configurations for launching them. This gives IWC clients the capability to launch other applications.

6.1. Data API

Data API

A simple key/value JSON store for sharing common resources among applications; It has backend connections that allow applications/listings to persist storage.

6.1.1. Actions

Available actions for the Data API:

get

Requests the Data API to return the resource stored with a specific key.

set

Requests the Data API to store the given resource with the given key.

list

Requests the Data API to return a list of children keys affiliated with a specific key.

watch

Requests the Data API to respond any time the resource of the given key change.

unwatch

Requests the Data API to stop responding to a specified watch request.

addChild

Requests the Data API to store a given resource and link it as a child to another resource.

removeChild

Requests the Data API to remove a given resource and unlink it as a child of another resource.

delete

Requests the Data API to remove a given resource.

6.1.2. Accessing the API

To call functions on the Data API, reference `client.data()` on the connected client.

```
var client = new ozpIwc.Client({
    peerUrl: "http://ozone-development.github.io/iwc"
});

client.connect().then(function(){
    var dataApi = client.data();
});
```

6.1.3. Listing Data API Resources

To obtain a list of all resources in the Data API, use the **list** action.

The list action without any resource specified returns an array of resource keys in **entity**:

```
var dataApi = client.data();

dataApi.list().then(function(res){
    var dataResources = res.entity;
});
```

The value of `res`, the resolved object of the list request, uses the following format:

```
{
    "response":"ok",
    "src":"data.api",
    "dst":"4e31a811.31de4ddb",
    "entity":[
        "/someResource",
        "/pizza",
        "/theme",
        "/parent",
        "/parent/1234",
        "/parent/5678",
    ],
    "ver":1,
    "time":1424897169456,
    "msgId":"i:1397"
    "replyTo":"p:704",
}
```

entity

The value of the resource. In this case, the resource is dynamically generated as an array of all resource names in the API. This means there is no Data API resource that is providing this information; rather it is gathered when called.

ver

The version of the resource. In this case, as a dynamically generated resource, the resource does not have a name and is not stored in the Data API. Because of this, history of the version does not exist. Thus, its history will always be `1`.

The list action can also be used for gathering children of a resource (see 4.9.2. Getting a List of a Data API Resource's Children).

6.1.4. Storing a Resource in the Data API

Storing an object in the Data API makes it available to other connected clients. This allows multiple applications to use the resource without communicating with each other.

To store a resource, the **set** action is used.

```
var dataApi = client.data();
      var foo = { 'bar': 'buz' };
      dataApi.set('/foo',{ entity: foo }).;
```

In this example, the resource `/foo` in the Data API is called to store `{ 'bar': 'buz' }` with the **set** action.

The value of `foo` is wrapped in an object's entity field because there are multiple properties that can drive the API's handling of the request:

entity

The value of the object that will be set.

contentType

The content type of the object that will be set.

Since the set is asynchronous, the client does not need to wait for the action to occur. However, the client can be told to wait through the `then` of the set call.

6.1.5. Storing a Resource in the Data API with Persistence

Without adding `persist: true`, updating a resource in the Data API would only last as long as the bus is open. If an action needs to persist longer than the user's current session, it is necessary to apply persistence.

Persisting a Data API action to the backend is as simple as including `persist: true` in the entity of the request.

```
var dataApi = client.data();
var foo = { 'bar': 'buz' };
foo.persist = true;
dataApi.set('/foo',{ entity: foo });
```

Since the set is asynchronous, the client does not need to wait for the action to occur. However, the client can be told to wait through the `then` of the set call.

6.1.6. Retrieving a Resource from the Data API

To retrieve a resource stored in the Data API the get action is used. The retrieval is asynchronous, and the response is passed through the resolution of the action's promise.

```
var dataApi = client.data();
var foo;

dataApi.get('/foo').then(function(res){
        foo = res.entity;
});
```

You can request a resource that does not exist. This action is valid. It will result in an `'ok'` response, an entity of `{}` and the resource created in the Data API (without persistence).

6.1.7. Watching a resource in the Data API

To watch a resource stored in the Data API for changes, the **watch** action is used.
Whenever a resource changes, it calls the onChange callback.
The promise resolves when it handles a response to the request, not when a change occurs.

```
var dataApi = client.data();
var onChange = function(reply,done){
    var newVal = reply.entity.newValue;
    var oldVal = reply.entity.oldValue;
    var doneCondition = { 'foo': 1 };
    if(newVal === doneCondition){
        done();
    }
};
dataApi.watch('/foo',onChange);
```

The value of the `onChange` callback's `reply` argument varies from a normal response (see 5.7: API Responses).

```
{
    "dst":"20c0f063.b80a890a",
    "src":"data.api",
    "replyTo":"p:1856",
    "response":"changed",
    "resource":"/foo",
    "entity":{
        "newValue":{"bar":"bark"},
        "oldValue":{"bar":"buz"},
        "removedChildren":[],
        "addedChildren":[]
    },
    "ver":1,
    "msgId":"i:3761",
    "time":1424901227419
```

```
}
```

response

The reply's response will not be `ok` like a request's response, rather a `changed` will denote the value has changed.

entity

The entity of the reply is not value stored in the resource, rather a report of the change in state of the resource.

entity.newValue

The new value of the resource.

entity.oldValue

The old value of the resource.

removedChildren

Children resources that were removed (see 4.9.5. Removing a Child Resource from the Data API)

addedChildren

Children resources that were added (see 4.9.3. Storing a Child Resource in the Data API)

6.1.8. Remove Watch

To stop watching the resource based on its state, calling the `done()` function passed to the callback will remove the watch.

6.1.9. Removing a resource from the Data API

To remove a resource stored in the Data API the `delete` action is used.

This only removes the resource from the Data API and not the persistent backend.

```
var dataApi = client.data();
dataApi.delete('/foo');
```

6.1.10. Removing a resource from the Data API with persistence

To remove a resource from the backend, passing an entity with `persist: true` will trigger a delete request to the backend.

```
var dataApi = client.data();
var foo = {persist: true};
dataApi.delete('/foo',{ entity: foo });
```

6.1.11. Data API Children Resources

Children resources are JSON objects that relate to a common parent resource. The resource name of the child resource is arbitrary as it is created at run time, but a link to child is created in the parent resource so actions like *"Run some function on all children of resource A"* are possible.

6.1.11.1. Example of using Children Resources

An example of when children resources are beneficial is a shopping cart implementation

`/shoppingCart`: a resource that holds information pertaining to a user's purchase. This is the parent resource and it holds a field total, the total of all items in the shopping cart.

`/shoppingCart/1234`: a runtime generated resource that holds information about an entry in the shopping cart. Since the entry may contain unique information like quantity, size, and color, it does not make sense to make a hardcoded resource. Rather, letting the IWC create a resource name and link it to `/shoppingCart` creates better versatility. This, if added as a child, will be a child resource of `/shoppingCart`

This structure lets it calculate and keep an updated total in `/shoppingCart` with minimal coding needed.

6.1.12. Getting a List of a Data API Resource's Children

Use the `list` action to get a list of all resources that are children of a resource.

Unlike getting a list of all resources in the Data API, passing a resource key into the list function returns an array of resource keys that are direct children of the resource.

```
var dataApi = client.data();
var cartItems = [];
dataApi.list('/shoppingCart').then(function(res){
    cartItems = res.entity;
});
```

6.1.13. Storing a Child Resource in the Data API

To add a child resource, use the `addChild` action on the desired parent resource.

Since the resource key of the child is runtime generated, it will be returned in the promise resolution's `entity.resource` as a string.

```
var dataApi = client.data();
var cartEntry = {
    'price': 10,
    'size': 'M',
    'color': 'red',
    'quantity': 1
};
var cartEntryResource = "";
dataApi.addChild('/shoppingCart',{ entity:
cartEntry}).then(function(res){
        cartEntryResource = res.entity.resource;
});
```

In this example, the added child will have a resource similar to `/shoppingCart/1234` with `1234` being the random runtime child key.

6.1.13.1. Storing a Child Resource in the Data API with Persistence

Just like with other persistence actions, adding `persist: true` to the resource's `entity` will cause the API to persist the resource to the backend.

```
var dataApi = client.data();
var cartEntry = {
    'price': 10,
    'size': 'M',
    'color': 'red',
    'quantity': 1
};
cartEntry.persist = true;
var cartEntryResource = "";
dataApi.addChild('/shoppingCart',{ entity:
cartEntry}).then(function(res){
    cartEntryResource = res.entity.resource;
});
```

6.1.13.2. Removing a Child Resource from the Data API

To remove a child resource from the Data API, use the `removeChild` action to remove the resource via its parent resource.

To know what child to remove, the entity's resource field passes in the resource key returned in the `addChild` resolution.

```
var dataApi = client.data();
var removeEntry = { resource: "/shoppingCart/1234" };
dataApi.removeChild('/shoppingCart',{ entity: removeEntry});
```

6.1.13.3. Removing a Child Resource from the Data API with Persistence

Just like with other persistence actions, adding `persist: true` to the resource entity will cause the API to persist the child resource removal on the backend.

To know what child to remove, the entity's resource field passes in the resource key returned in the `addChild` resolution.

```
var dataApi = client.data();

var removeEntry = {
    resource: "/shoppingCart/1234",
    persist: true,
};

dataApi.removeChild('/shoppingCart',{ entity: removeEntry});
```