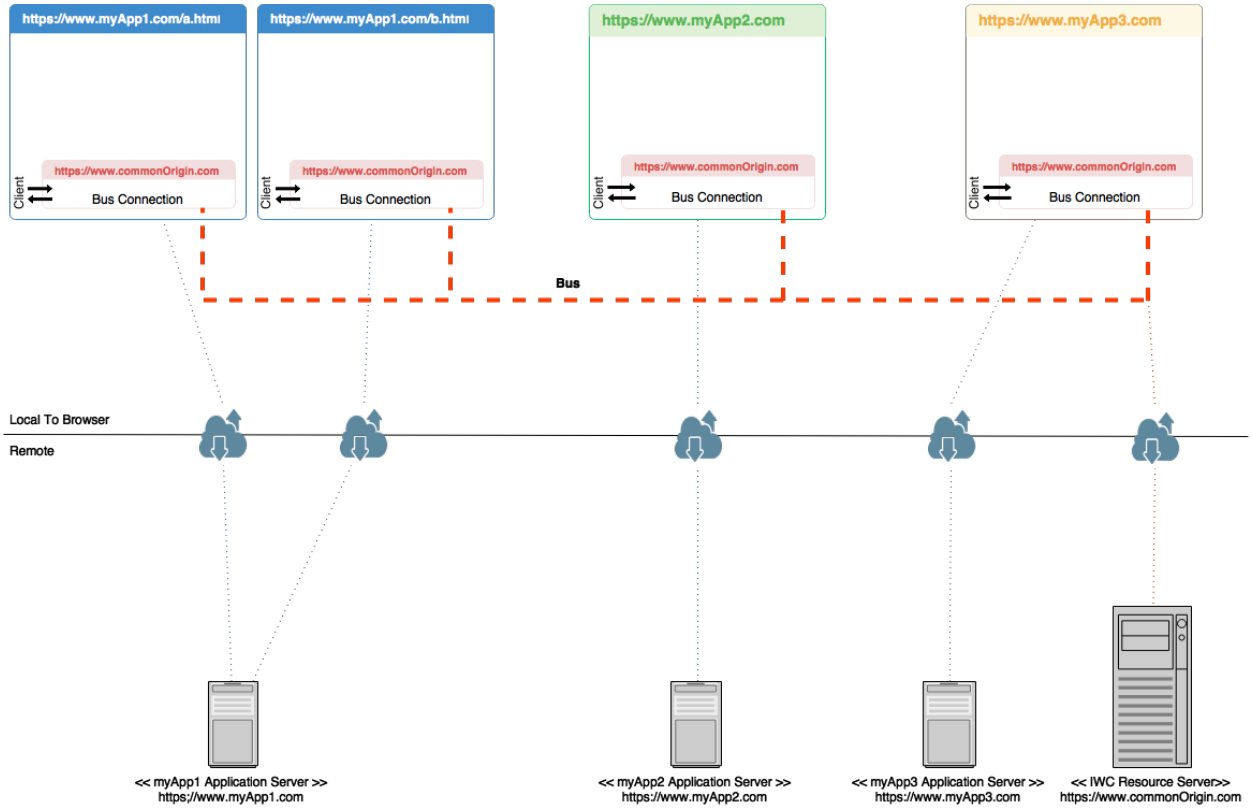
Contents

Overview

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



Client

The client component of the IWC is standard amongst all IWC instances. It is a library that developers use to have their
application connect to and make requests on any given bus. To connect to a bus, the client library opens the desired
bus in an invisible iFrame. In order to talk across the domain of the widget and the domain of the bus, the client and
bus components have a defined protocol for making cross origin requests.

The client component resides in the domain of the application, while the bus component resides in the domain hosting
the bus. In some deployment instances this may be the same domain.

Bus

The bus component of the IWC acts as a **local** network for clients to communicate through.

While each client's connection to the network opens an instance of the bus component in an iFrame, distributed consensus algorithms are used to determine that only one instance of the bus component is running. When said
bus component becomes unavailable, the consensus algorithms are used to redirect IWC traffic to a different bus
component.

A running bus component communicates with its remote server to gather information regarding available applications,
preferences, persisted data and permission level to enhance functionality of clients (widgets) connected. More information on communication with the remote server can be found in the Server communications section.

A client (widget) can only communicate with other clients(widgets) connected to the same bus, this is due to the
security measures put in place for [cross-origin resource sharing(CORS)] (http://www.w3.org/TR/cors/). Bridging two
buses together (joining two bridge domains) is possible, but not implemented at this time.

Technology Stack

The IWC is a client-side library built without any dependencies on other javascript frameworks. Polly-fill libraries are used to add coverage for EcmaScript 6 functionality across legacy browsers. The intention is to maintain a minimal library size to not impact load times. Currently the minified IWC client library is 45kB and its minified bus counterpart is 155kB.

LocalStorage

LocalStorage is a key-value data store built into the browser that is separted by the domain of the webpage loaded in the browser.

If one browser tab has http://www.example.com/page1.html open, that website has access to write to the key-value store assigned to the http://www.example.com domain. If a second browser tab was opened with http://www.example.com/page2.html, that website would have access to the same key-value store as http://www.example.com/page1.html because they are in the same domain.

Since the two open websites are accessing the same local storage, they receive identical events when data is stored/deleted. This is the foundation of the IWC, with the ability to access the same data storage, widgets are able to communicate.

The use of LocalStorage comes with limitations, if two widgets want to communicate but are on different domains, http://www.example.com/page1.html and http://www.sample.com/page1.html for example, local storage alone wont allow these two widgets to communicate, but if each widget had reference to a connection on a common domain, http://www.iwchost.com for example, then the two widgets could communicate. To have this connection reference, HTML Iframes and PostMessage are utilized.

Iframes and PostMessage

When a website is loads another website inside of an iframe, javascript on the main website can communicate with javascript on the iframe-loaded website using the browser's built in `window.PostMessage`. PostMessage safely enables cross-origin communication, in our case between http://www.example.com/page1.html and http://www.iwchost.com, because PostMessage communication is direct between two windows, where the transmitter must specify both the window to comunicate with (the iframe) and the origin of which the window must have loaded in order to accept the message (http://www.iwchost.com).

In the instance of an widget using IWC, the IWC client library ensures that when it sends messages to its common domain that:

1. The page loaded in the iframe is indeed the common domain for IWC communication.
2. Whenever receiving a message, the origin of the message matches the expected origin of the iframe.

IWC Client

This communication through an iframe marks the separation of the two IWC libraries, `ozpIwc-client.js` and `ozpIwc-bus.js`. All code ran by the `ozpIwc-client.js` acts as a representative of the bus for a widget to communicate through. This library has no server communication. Widget developers will only use the IWC client library for their widget, so only an understanding of connecting a client and the client API calls are necessary to develop.

Widgets will show no visible trace its use of iframes and PostMessage to communicate with the IWC bus because the IWC client library generates the iframe to be hidden and have no size.

To have a common point to load the IWC bus libraries, each IWC client instance, when connecting, will load the common domain website into the iframe. For simplicity, the IWC bus library is packaged up in a `iframe_peer.html` file, such that when connecting a client, it simply sets a `peerUrl` parameter to the path that holds the iframe_peer.html file.

In other terms, if the IWC bus is available at http://www.iwchost.com/iframe_peer.html, any widget that creates an IWC client to connect as follows will have access to use that bus:

```
var client = new OzpIwc.Client({peerUrl: "http://www.iwchost.com"});
```

IWC Bus

Connection to the bus does not mean that 1 user's widget communicates with another user's widget, rather the bus runs local to the user's browser inside the `iframe_peer.html` file, allowing all other widgets open using that browser (in other tabs or windows) to communicate given they have a matching bus.

The reason these widgets must load the `iframe_peer.html` from a common domain (http://www.iwchost.com) flows back to the use of **LocalStorage**. In order to communicate locally through the key-value store, a common domain must be used. Thus, loading a resource on the common domain opens up the ability to use this communication practice.

Each IWC client loads an instance of the IWC bus. By using a distributed consensus algorithm, a single instance of the bus will act upon requests and the other instances will sit and wait incase they need to take over leadership.

The IWC bus can communicate back to a server if desired. While the IWC without any server communication is powerful all in its own it can be encorporated with a backend server for:

1. The ability to persist data for saved state and future application use.

2. Loading information to alow users to launch widgets ready for use on the bus.
3. Attribute-based access control.

Further information on server communication of the IWC can be found in the Server Communication section.

XMLHttpRequest

XMLHttpRequest (AJAX) is used by the IWC bus library to gather bus information from its corresponding server if configured to do so.

Server Communication

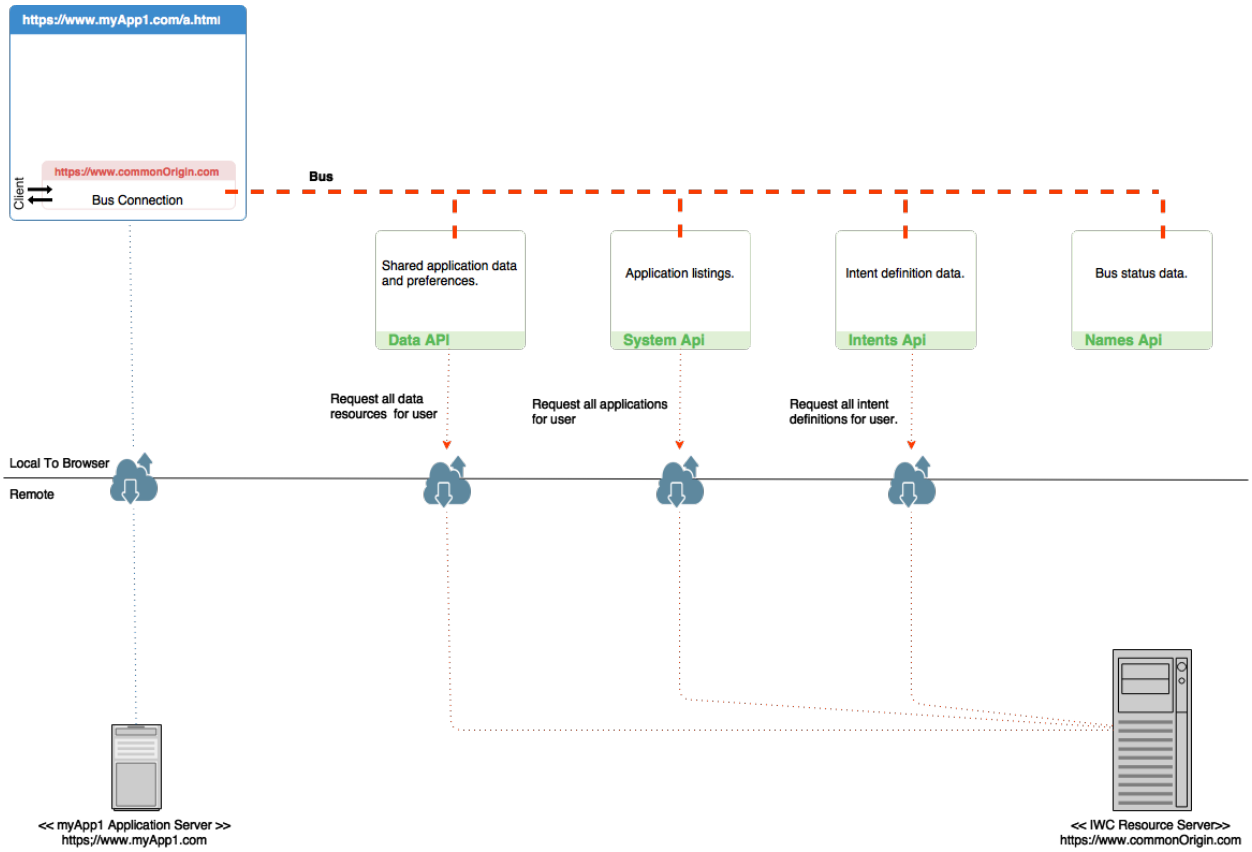
The Bus component of the IWC communicates with its remote server at the creation of the bus to request persistent
resources.

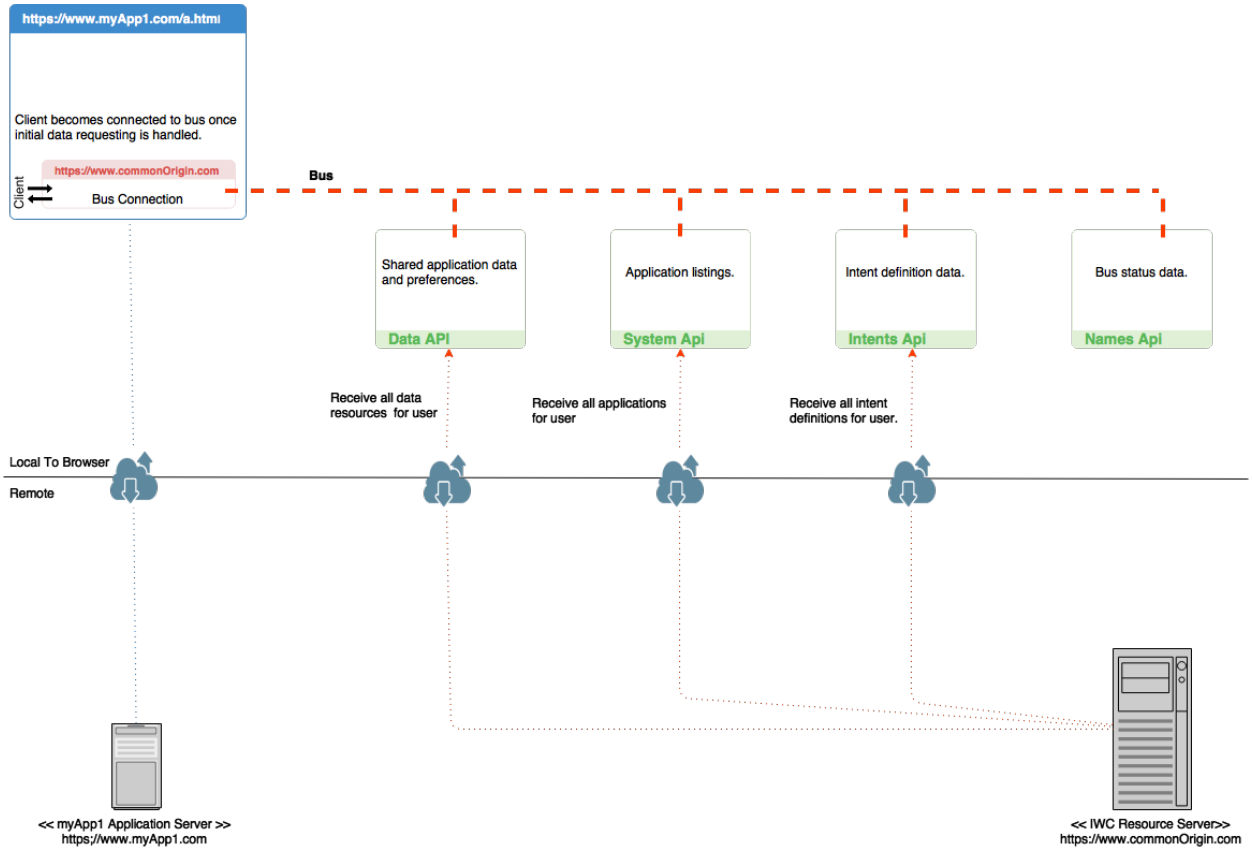
When a client performs an action against an API they are performing it in the local network that is the IWC bus.
Individual APIs that are inside the bus component may choose to communicate with the remote server additionally.

Bus initialization

When the bus is created (first client connects), individual APIs in the bus component will reach out to the remote
server for initial data.

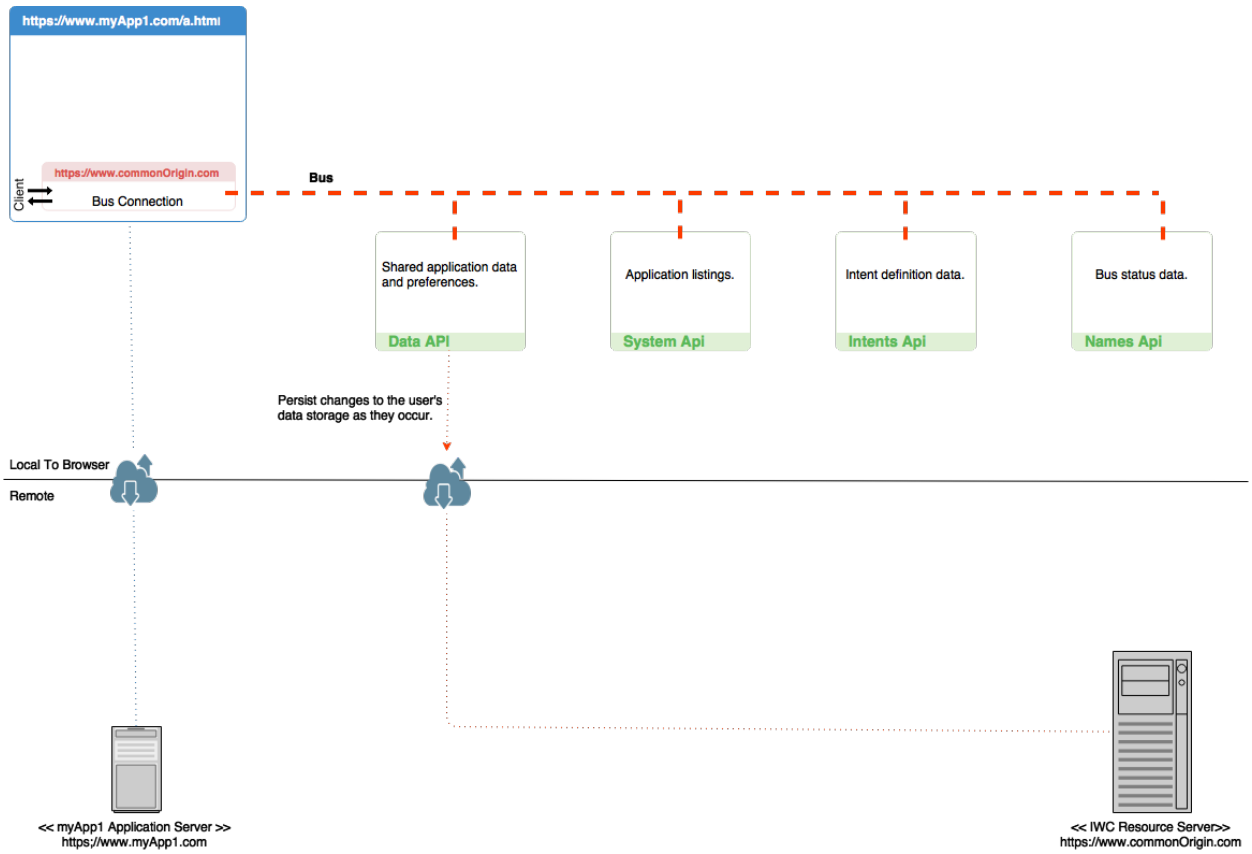
When the initial data requests are resolved, the bus is initialized and the client is connected.


https://www.myApp1.com/a.html
https://www.commonOrigin.com
Client
Bus Connection
Bus
Shared application data and preferences.
Data API
Application listings.
System Api
Intent definition data.
Intents Api
Bus status data.
Names Api
Request all data resources for user
Request all applications for user
Request all intent definitions for user.
Local To Browser
Remote
<< myApp1 Application Server >>
https://www.myApp1.com
<< IWC Resource Server>>
https://www.commonOrigin.com


https://www.myApp1.com/a.html
Client becomes connected to bus once initial data requesting is handled.
https://www.commonOrigin.com
Client
Bus Connection
Bus
Shared application data and preferences.
Data API
Application listings.
System Api
Intent definition data.
Intents Api
Bus status data.
Names Api
Receive all data resources for user
Receive all applications for user
Receive all intent definitions for user.
Local To Browser
Remote
<< myApp1 Application Server >>
https://www.myApp1.com
<< IWC Resource Server>>
https://www.commonOrigin.com

Bus lifespan

As long as the user has at least one client connected to a given bus, the resources for the buses APIs are retained.
The Data API is the only api that persists resource changes to the remote server. Whenever a resource changes a post
request is made to the server with the changed data.



Server Unavailability

Documentation pertaining to remote server unavailability mitigation to be completed.

Setup

Using bower, the IWC client can be gathered with the following:

```
bower install ozone-development/ozp-iwc
```

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>
```

A pre-configured example bus is hosted at `http://ozone-development.github.io/iwc` for demonstrative purposes.
See connecting for connecting a client to the bus.

To use a configured IWC bus, refer to the IWC deployment guide.
An example configured bus is provided with the distributed IWC via github.

Connecting

To use IWC between applications, IWC client connections must be added to each application.
These clients connect to an IWC bus that is bound by the browser as well as the domain it is obtained from.

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

client.connect().then(function(){
   /* client use goes here */
});
```

In this example, an IWC connection is made to the bus on domain `http://ozone-development.github.io/iwc`.
The actual javascript that makes up the bus is gathered from that url and ran locally enclosed in the same domain.

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

Disconnecting

Disconnecting an application from the IWC bus is as simple as calling `disconnect()`.

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

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

Quick Start

The purpose of this guide is to get a base hosting of the IWC running locally.

Building and Running the Bus (includes a static mock-backend)

1. Install node.js.
2. sudo npm install -g grunt-cli bower
3. git clone git@github.com:ozone-development/ozp-iwc.git
4. cd ozp-iwc
5. git submodule init && git submodule update
6. npm install && bower install
7. grunt connect-all
8. Browse to http://localhost:14000 for an index of samples and tests.

Connecting a Javascript entity to the Bus

1. Include the IWC's client library to your html.

   - **If using bower**: Include the ozp-iwc library to your bower dependencies. Then reference the bower_component
     file.

     ```
     bower install ozp-iwc --save-dev
     ```

     ```
     <script src="<relative pathing for project>/bower_components/ozp-iwc-/dist/js/ozpIwc-client.mi
     ```

   - **Not using bower**: Copy the library (dist/js/ozpIwc-client.min.js) into your widget's project and refer to it
     with the script tag.

     ```
     <script src="<relative pathing for project>/ozpIwc-client.min.js"></script>
     ```

2. Create a IWC client

   ```
   var client = new ozpIwc.Client({ peerUrl: "http://localhost:13000"});
   ```

3. Test that the client can connect

   ```
   client.connect().then(function(){
     console.log("IWC Client connected with address:", client.address);
   }).catch(function(err){
     console.log("IWC Client failed to connect:", err);
   });
   ```

Key-Value capabilities

1. Storing

   ```
   var value = {
       size: "medium",
       quantity: 1,
       color: "red"
   };

   client.data().set("/t-shirt/Style01",{entity: value});
   ```

2. Retrieving

   ```
   var value;

   client.data().get("/t-shirt/Style01").then(function(resp){
       value = resp.entity;
   });
   ```

3. Watching

   ```
   var onChange = function(resp){
       console.log("Old value:", resp.entity.oldValue);
       console.log("New value:", resp.entity.newValue);
   };

   client.data().watch("/t-shirt/Style01",onChange);
   ```

Application capabilities

1. Listing applications registered to the bus

```
var applications;

client.system().get("/application").then(function(resp){
    applications = resp.entity;
});
```

2. Launching an application registered to the bus

```
var application = "/application/1234";

client.system().launch(application);
```

Intent capabilities

1. Registering an intent

```
var intent = "/application/json/print";

var config = {
    entity: {
        type: "/application/json",
        action: "print",
        icon: "printIcon.png",
        label: "Console logs data received from the invoker"
    }
};

var onInvoke = function(resp){
        console.log("Intent invoked with the following data:", resp.entity);
};

client.intents().register(intent, config, onInvoke);
```

2. Invoking an intent

```
var intent = "/application/json/print";
var value = {
    size: "medium",
    quantity: 1,
    color: "red"
};

client.intents().invoke(intent,value);
```

3. Broadcasting an intent (all registrants will handle)

```
var intent = "/application/json/print";
var value = {
```

```
            size: "medium",
            quantity: 1,
            color: "red"
        };

        client.intents().broadcast(intent,value);
```

Available APIs

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

This code demonstrates showing a console print out of what APIs are available, the actions they can handle, and the client's function name assigned to it.

```
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]
    };
});
```

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");`

The purpose of this is to inform the developer of what functionality they have to work with given a particular bus.

To learn more about api use, see API Requests.

API Requests

Making an api request uses the following structure:

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

Api: {Function} The api the client is calling. ex) `data(...)`

Action: {Promise} The action the client is performing. ex) `set(...)`

Path: {String} The path of the stored resource pertaining to the action. ex) `"/foo"`

Value(Optional): {Object} The object sent for the Api to operate on. ex: `{ 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**.

Since there are instances where callbacks may be used but no value needed, action calls can be also made using the
following additional structures:

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

The one restriction is the following Api call structure is not allowed:

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

API Responses

Since the IWC operates asynchronously, all requests receive a response sent back to a client that resolves the actions
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 is to 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.
If they are not demonstrated in an example that does not mean they are not useable. All actions will resolve/reject with
the response received from the Bus.

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' is indication 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 response was generated and sent from the Data API module.

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"}`

pattern: A string pattern set to the resource to compile its collection of other relevant resources when watched.

collection: An array of resources matching this resource's pattern , this only updated if the resource is being watched.

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 identifier of the request that this response was sent because of.

Response Types

The following table breaks down the `response` property of the object passed back to the promises then, catch, or
registered callback:

Response	Occurs When	Reason
ok	Promise resolves or registered callback called.	Action was performed as expected
changed	Registered callback for a watch action called.	A resource has changed after a watch request was issued. The entity contains fields “newValue” and “oldValue” fields that of the indicated contentType. If the resource was deleted, “newValue” will be undefined. If the resource was created, “oldValue” will be null.
badResource	Promise rejects.	The resource was not semantically valid for this API.
badAction	Promise rejects.	The action property of the request is not valid in this API.
badPermission	Promise rejects.	The permission property of the request was not valid.
noPermission	Promise rejects.	Sender does not have permission to perform the requested action.
noMatch	Promise rejects.	A conditional action failed.

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 API Responses).

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 `response` field of **errRes**.

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 snippit below:

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

Example

This snippit 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 change of value of `/foo` 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.

Core APIs

The IWC Bus defaults to having the following 4 APIs:

- Data API:
  A simple key/value JSON store for sharing common resources amongst applications. Creating, updating, and deleting Data
  API resources persists to the Data APIs endpoint.

- 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 presents the user with a
  dialog to choose what application should handle their request.
- Names API:
  Status of the bus. This api exposes information regarding applications connected to the bus. This is a read-only API.
- System API:
  Application registrations of the bus. This api gives connections to the bus awareness of what applications the bus
  has knowledge of. Different then the names api, these application's 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. This is a read-only API.

Common Functionality

All APIs have access to the following actions. Some APIs may choose to prevent or modify the behavior of an action.

get: Requests the API to return the resource stored with a specific key. (see Retrieving Resources)

bulkGet: Requests the API to return multiple resources stored with a matching key. (see Retrieving Resources)

set: Requests the API to store the given resource with the given key. (see Storing Resources)

list: Requests the API to return a list of children keys pertaining to a specific key. (see Listing Resources)

watch: Requests the API to respond any time the resource of the given key changes. (see Watching Resources)

unwatch: Requests the API to stop responding to a specified watch request. (see Watching Resources)

delete: Requests the API to remove a given resource. (see Removing Resources)

Storing a Resource in an API

Storing an object in an API makes it available to other connected clients. This is useful for resources that are desired among multiple applications but do not need communication between said applications.

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 APIs 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. This is an optional parameter, in some APIs resources
have refined control based on the content they hold.

pattern: A string matching a partial path of resources. This is used to allow resources to include updates when
about other relevant resources when watched. For example a pattern of "/foo/" on resource `/foo` would return
all resources below "/foo/" if watched for changes ("/foo/1", "/foo/2", ...). This is an optional parameter, it will
only update on the resource if you include it in the request. See Watching Resources
for further details.

Since the set is asynchronous, the client does not need to wait for the action to occur. Although, if an implementation
desires waiting it can be done through the `then` of the set call.

Note: If the API does not allow storing of resources (read-only APIs or read-only resources), the promise will reject with a response of "noPermission".

```
var systemApi = client.system();

systemApi.set('/application/1234-1234-1234-1234').catch(function(err){
    //err.response === "noPermission"
});
```

Retrieving a Resource from an API

To retrieve a resource stored in an 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;
});
```

The value of `res`, the resolved object of the get request, is formatted as follows:

```
{
    "response":"ok",
    "src":"data.api",
    "dst":"4e31a811.31de4ddb",
    "entity":{
        "bar": 'buz'
    },
    "ver":1,
    "time":1424897872164,
    "msgId":"i:33"
    "replyTo":"p:7",
}
```

Requesting a resource that does not exist is not an valid action, this will result in an promise rejection with a `'noResource'` response and an entity containing the request packet.

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

dataApi.get('/a/nonexistant/resource').catch(function(err){
        //err.response === "noResource"
});
```

The value of `err`, the rejected object of the get request, is formatted as follows:

```
{
    "ver": 1,
    "src": "data.api",
    "msgId": "p:686",
    "time": 1435674200150,
    "response": "noResource",
    "entity": {
      "ver": 1,
      "src": "c1f6b99e.21851da2",
      "msgId": "p:692",
      "time": 1435674200148,
      "dst": "data.api",
      "action": "get",
      "resource": "/a/nonexistant/resource",
      "entity": {}
    },
    "replyTo": "p:692",
    "dst": "c1f6b99e.21851da2"
}
```

Retrieving a collection of resources

In some cases, gathering multiple resources at once is desired. The `bulkGet` action takes a resource path, and returns
the resource values for all of the API's resources that match.

For example, the Names API has a collection of resources on the various APIs connected to the IWC. These resources are
labeled as `/api/{address}` (/api/data.api, /api/names.api, ...).

```
var namesApi = client.names();
var apiMap = {}
names.bulkGet("/api").then(function(res){
    for(var i in res.entity){
        var resource = res.entity[i].resource;
        var apiEntity = res.entity[i].entity;

        apiMap[resource] = apiEntity;
    }
});
```

The value of `res`, the resolved object of the bulkGet request is formatted as follows (omitted entries due to length):

```
{
    "ver": 1,
    "src": "names.api",
    "msgId": "p:119470",
    "time": 1435701118804,
    "contentType": "application/json",
    "entity": [
        {
            "entity": {
                "actions": [
                    "get",
                    "set",
                    "delete",
                    "watch",
                    "unwatch",
                    "list",
                    "bulkGet",
                    "addChild",
                    "removeChild"
                ]
            },
            "lifespan": {
                "type": "Ephemeral"
            },
            "contentType": "application/vnd.ozp-iwc-api-v1+json",
            "permissions": {},
            "eTag": 1,
            "resource": "/api/data.api",
            "collection": []
        },
        {
            "entity": {
                "actions": [
                    "get",
                    "set",
                    "delete",
                    "watch",
                    "unwatch",
                    "list",
                    "bulkGet"
                ]
            },
            "lifespan": {
                "type": "Ephemeral"
            },
            "contentType": "application/vnd.ozp-iwc-api-v1+json",
            "permissions": {},
            "eTag": 1,
            "resource": "/api/names.api",
            "collection": []
        },
        // Omitted additional entries due to size...
    ],
    "response": "ok",
```

```
    "replyTo": "p:3370",
    "dst": "abc57c00.e3b19f7f"
}
```

The resulting `apiMap` variable would look as so:

```
{
    "/api/data.api": {
        "actions": [
            "get",
            "set",
            "delete",
            "watch",
            "unwatch",
            "list",
            "bulkGet",
            "addChild",
            "removeChild"
        ]
    },
    "/api/names.api": {
        "actions": [
            "get",
            "set",
            "delete",
            "watch",
            "unwatch",
            "list",
            "bulkGet"
        ]
    },
    // Omitted additional entries due to size...
}
```

Removing a resource from an API

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

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

dataApi.delete('/foo');
```

Deleting a resource that does not exist **is a valid action**. Thus, so long as the API allows deletion of resources
resolved response would be as follows:

```
{
    "ver": 1,
    "src": "data.api",
    "msgId": "p:3684",
    "time": 1435674349620,
    "response": "ok",
```

```
    "replyTo": "p:3660",
    "dst": "c1f6b99e.21851da2"
}
```

Note: If the API does not allow deletion of resources, the promise will reject with a response of "noPermission".

```
var namesApi = client.names();

namesApi.delete('/api/names.api').catch(function(err){
    //err.response === "noPermission"
});
```

The value of `err`, the rejected object of the delete request, is formatted as follows:

```
{
  "ver": 1,
  "src": "system.api",
  "msgId": "p:71",
  "time": 1435674505220,
  "response": "noPermission",
  "entity": {
    "ver": 1,
    "src": "c1f6b99e.21851da2",
    "msgId": "p:6748",
    "time": 1435674505218,
    "dst": "system.api",
    "action": "delete",
    "resource": "/a/nonexistant/resource",
    "entity": {}
  },
  "replyTo": "p:6748",
  "dst": "c1f6b99e.21851da2"
}
```

Listing API Resources

To obtain a list of all resources in the API, the **list** action is used.

The list action on the root path returns an array of all of the API's resource keys in it's entity. This is because
the list action matches any resource that **begins with** the resource provided.

The list action does not create or update a resource, rather finds resources within the API that match the resource
string provided.

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

dataApi.list("/").then(function(res){
    var dataResources = 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":[
        "/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 the case of a list action no individual resource is returned, rather a list of matches. Because of this, history of the version does not exist so it will always be `1`.

Watching a resource in an API

To watch a resource stored in an API for changes, the **watch** action is used.

the promise resolves when a response to the request is handled, **not when a change has occurred**.

The onChange callback is called whenever there is a change in the resource.

```
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 Receiving Responses):

```
{
    "dst":"20c0f063.b80a890a",
    "src":"data.api",
    "replyTo":"p:1856",
    "response":"changed",
    "resource":"/foo",
    "entity":{
        "newValue":{"bar":"bark"},
        "oldValue":{"bar":"buz"},
        "newCollection":[],
        "oldCollection":[]
    },
    "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.

entity.newCollection: new collection resources that start with the watched resources pattern. This will be an empty
array if the resource does not have a pattern. (see Storing Resources for information on
setting the resources pattern).

entity.oldCollection: old collection resources that start with the watched resources pattern. This will be an empty
array if the resource does not have a pattern.

Collection

The collection field of a resource defaults to only updating if the following are true:

1. The resource has a `pattern` property.
2. The resource is being watched by some connection on the bus.

The collection field will continue to update so long as both conditions are true. Some APIs may choose to update
certain resources based on their own criteria. In these cases, sufficient documentation should be provided to denote
when and how a resources collection updates.

Stoping a watch

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

Stopping a watch external to its callback

In some cases, stopping a watch may be desired outside of the onChange callback. For this reason, there is the `unwatch`
action. When registering a watch, 2 important properties are returned in the resolution:

1. **dst**: the participant that sent the request
2. **replyTo**: the packet that was sent to create the watch

With these 2 properties, the watch can be unregistered using `unwatch` as so:

```
var watchData = {
    src: null,
    msgId: null
};

dataApi.watch('/foo',onChange).then(function(res){
    watchData.src = res.dst;
    watchData.msgId = res.replyTo;

});

// Somewhere else after the watch resolution occurs
dataApi.unwatch('/foo',watchData);
```

Watching a resource that does not exist

When registering a watch on a non-existant resource, the promise resolution will return as follows:

```
{
  "ver": 1,
  "src": "data.api",
  "msgId": "p:410",
  "time": 1435676177310,
  "response": "ok",
  "replyTo": "p:39925",
  "dst": "c1f6b99e.21851da2"
}
```

Watching a resource that already exists

When registering a watch on an existing resource, the promise resolution will be that of a `get` action:

```
{
  "ver": 1,
  "src": "data.api",
  "msgId": "p:41758",
  "time": 1435676337690,
  "lifespan": {
    "type": "Persistent"
  },
```

```
  "permissions": {},
  "eTag": 3,
  "resource": "/balls",
  "pattern": "/balls/",
  "collection": [
    "/balls/63abb1f0",
    "/balls/a650bb08"
  ],
  "entity": {
      "foo": "bar"
  },
  "response": "ok",
  "replyTo": "p:41340",
  "dst": "c1f6b99e.21851da2"
}
```

Watching a resource that gets deleted

When a resource that is being watched gets deleted, the watchers of said resource receive a change notification indicating the deletion.

```
{
    "ver": 1,
    "src": "cd78b1cf.21851da2",
    "msgId": "p:44993",
    "time": 1435676498955,
    "dst": "c1f6b99e.21851da2",
    "replyTo": "p:41340",
    "response": "changed",
    "resource": "/balls",
    "permissions": {},
    "entity": {
        "newValue": null,
        "OldValue":{
            "foo": "bar"
        },
        "oldCollection": [
            "/balls/63abb1f0",
            "/balls/a650bb08"
        ],
        "newCollection": null,
        "deleted": true
    }
}
```

The value of `entity.deleted` will be marked true to indicate the resource has been deleted. The watch will remain active
regardless of deletion and notify if it is recreated. If removing a watch on deletion of a resource is desired, applying
conditional logic for the `done()` flag based on `entity.deleted` can produce this functionality.

Data Api

A simple key/value JSON store for sharing common resources amongst applications. Persists creation, updates, and
deletions to its endpoint.

Common Actions

All common API actions apply to the Data API, one behavioural difference is resource creation, updates, and deletions
are persisted.

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

bulkGet: Requests the Data API to return multiple resources stored with a matching 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 pertaining to a specific key.

watch: Requests the Data API to respond any time the resource of the given key changes.

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

delete: Requests the Data API to remove a given resource.

Additional Actions

Additional actions for the Data API pertain to the the concept of children nodes. Refer to
Children Resources for further information.

addChild: Requests the Data API to create a resource below the resource provided. If a resource has children added
to it, it will begin updating its `collection` property with said children resources.

removeChild: Requests the Data API to remove the child resource. **deprecated**: Functionally uses delete action.

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();
});
```

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.

An example of children resources: items in a shopping cart

`/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 holding information pertaining 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 calculating and keeping an update total in `/shoppingCart` simple with minimal coding needed.

Storing a Child Resource in the Data API

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

Adding a child to the Data API triggers its parent to keep an up-to-date list of its child resources in its `collection` property. This means doing a `get` on `/shoppingCart` will return a collection of children in its resources.

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 `/shopingCart/1234` with `1234` being the random
runtime child key.

Retrieving a Child Resource in the Data API

To gather information of child resources, the `get` action is used on the desired parent resource.

```
var dataApi = client.data();
var children = [];

dataApi.get("/shoppingCart").then(function(res){
    children = res.collection;
});
```

Children resources are structurally no different than their parent, so gathering the child resource is a normal `get`
action:

```
dataApi.get("/shoppingCart/1234").then(function(res){

});
```

Removing a Child Resource from the Data API

DEPRECATED: Functionally the removeChild action just routes the child resource to the "delete" action.

To remove a child resource from the Data API, it should be removed via it's parent resource with the `removeChild`
action.

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

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

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

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

Intents Api

An intent module for applications to share working features amongst one another, modeled after Android intents.

Aside from creating, updating, and deleting intent handler registrations the Intents API acts as a read only API.
It contains state machines internal to the bus to drive the handling of intents. Rather than acting as a key/value
store, this API returns data on read requests catered to the type of resource it is requesting. This allows a simplistic
client interface.
***

Common Actions

The following common Actions apply to the Intents API. Note that some actions have resource specific behavior covered below.

get: Requests the Intents API to return the resource stored with a specific key.

bulkGet: Requests the Intents API to return multiple resources stored with a matching key.

set: Requests the Intents API to store the given resource with the given key.

list: Requests the Intents API to return a list of children keys pertaining to a specific key.

watch: Requests the Intents API to respond any time the resource of the given key changes.

unwatch: Requests the Intents API to stop responding to a specified watch request.

delete: Requests the IntentsAPI to remove a given resource.

Additional Actions

register: Registers a handler resource for an intent.

invoke: Sends out data to be handled by a desired intent. The intent chooser will be displayed if multiple options
are available.

broadcast: Sounds out data to be handled by all registered handlers for a desired intent.

Intents API resource structure

The Intents API relies on the path-level separation resource scheme to easily separate differences in resources.

The table below breaks down the differences in each level. The bracket notation is not a literal part of the resource
string, rather an explanation of what that string segment resembles. (ex `/{color}/{size}` signifies the first level of
the path is some color ("blue", "green", ect), and the second level is a size ("small", "medium", ect).

/{major}/{minor}: A content type grouping of intent actions. For example,
`/application/vnd.ozp-iwc-launch-data-v1+json` would be the root path to all intent actions pertaining launch data.

/{major/{minor}/{action}: An intent definition for a certain content type. For example,
`/application/vnd.ozp-iwc-launch-data-v1+json/run` would be an invokable intent for running an application. This path is
also the root path for all of its registered handlers. **Invoking a definition will drive the user to choose a handler
preference if more than one is available**.

/{major/{minor}/{action}/{handlerId}: A registered handler for an intent. For example,
`/application/vnd.ozp-iwc-launch-data-v1+json/run/1234` would be an invokable intent for running an application.
Invoking a registered handler will trigger the callback on the client that registered.

/inFlightIntent/{id}: Internal state machine resources. used by the client library to report on the status of an
intent being handled. Not intended for use by widgets.

Resource	Content Type	set	get	delete	register
/{major}/{minor}		no	yes*	no	no
/{major}/{minor}/{action}		no	yes	yes	yes
/{major}/{minor}/{action}/{handlerId}	application/vnd.ozp-iwc-intent-handler-v1+json	yes	yes	yes	yes
/inFlightIntent/{id}		yes	yes	yes	no

Requesting an action on a resource that does not fit the given resource structure will result in a response of "badResource":

```
{
    "ver": 1,
    "src": "intents.api",
    "msgId": "p:1033",
    "time": 1435685026449,
    "response": "badResource",
    "entity": {
        "ver": 1,
        "src": "3de31e8b.a5efe614",
        "msgId": "p:867",
        "time": 1435685026438,
        "dst": "intents.api",
        "action": "get",
        "resource": "/foo",
        "entity": {}
    },
    "replyTo": "p:867",
    "dst": "3de31e8b.a5efe614"
}
```

/{major}/{minor}: requesting a get of this path will return all actions available to that content type.

For information on registering and invoking intents see Intent Registrations and Intent Invocations.

Accessing the API

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

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

client.connect().then(function(){
    var intentsApi = client.intents();
});
```

Registering an Intent Handler

When registering an intent handler, two entity properties are used to make choosing a handler easier for the end user:

1. **label**: A short string noting the widget handling the intent (typically the widget title).
2. **icon**: A url path to a icon to use for the widget.

When a label/icon is not provided, the page title of the widget is used as the label and the icon will default to a
predefined default icon.

```
var intentsApi = client.intents();

var config = {
    "contentType": "application/vnd.ozp-iwc-intent-handler-v1+json",
    "entity": {
        "label": "My JSON Viewer",
        "icon": "https://www.example.com/icon.png"
    }
};

var onInvoke = function(event) {
    var payload = event.entity;
    someWidgetFunction(payload);
};

intentsApi.register("/application/json/view", config, onInvoke);
```

If the registration resource path matches `/{minor}/{major}/{action}` ("/application/json/view") the handler Id will be
generated automatically and returned in the promise resolution.

If the registration resource path matches `/{minor}/{major}/{action}/{handlerId}` ("/application/json/view/123") the
handler Id given will be used.

The registration promise resolution does not handle the intent invocation, rather reports the status of the registration:

```
{
    "ver": 1,
    "src": "intents.api",
    "msgId": "p:2380",
    "time": 1435690954688,
    "response": "ok",
    "entity": {
        "resource": "/application/json/view/3229d7e2"
    },
    "replyTo": "p:15791",
    "dst": "3de31e8b.a5efe614"
}
```

The `entity.resource` property of the response is the resource that was used for the handler. To unregister simply
delete said resource:

```
intentsApi.delete("/application/json/view/3228d7e2");
```

Invoking an Intent

When a widget wants to offload operations to other widgets on the bus, intent invocations are used. Much like Android intents, a payload is not a requirement to send with the intent but is an added benefit. Intents can be used
for notifications (broadcast to all widgets on the bus), triggering some background operation (shutdown utilities),
offloading common tasks (visualizing data, compiling output files, converting documents), and much more.

To invoke an intent, the `invoke` action is used:

```
var intentsApi = client.intents();
var payload = {
    "Hello": "World!"
};

intentsApi.invoke("/application/view/json",{ entity: payload}).then(function(res){
    // resolves when the intent API receives the request.
});
```

What if there are no handlers?

If there are no handlers open for the desired intent invocation, the invoke promise will reject with a response of
"noResource". At current state of the platform, launching a registered application to handle an intent invocation has
not been implemented.

It is planned to replace the promise rejection with giving the user a choice of registered applications to launch to
handle the desired invocation.

What if there is more than one handler?

If more than one handler is available, the user is prompted with an "intent chooser" that allows a decision to be made
by the user for which handler (widget) should accept the invocation.

User preferences on intent handlers is currently in development. The user will be able to save handler choices so that
they do not use an "intent chooser" unless the IWC can't find a preference.

What if I want to have all handlers accept my invocation?

Using the `broadcast` action, all handlers will accept the intent invocation and process it:
To invoke an intent, the `invoke` action is used:

```
var intentsApi = client.intents();
var payload = {
    "Hello": "World!"
};

intentsApi.broadcast("/application/view/json",{ entity: payload});
```

System API

The System API contains information on applications registered to the IWC bus, as well as the functionality of launching
them. The System API is read only

Common Actions

All common API actions apply to the System API, but only read actions are permitted.

get: Requests the System API to return the resource stored with a specific key.

bulkGet: Requests the System API to return multiple resources stored with a matching key.

set: Requests the System API to store the given resource with the given key.

list: Requests the System API to return a list of children keys pertaining to a specific key.

watch: Requests the System API to respond any time the resource of the given key changes.

unwatch: Requests the System API to stop responding to a specified watch request.

delete: Requests the System API to remove a given resource.

Additional Actions

launch: Opens a desired application. (see Launching Applications)

System API resource structure

The System API relies on the path-level separation resource scheme to easily separate differences in resources.

The table below breaks down the differences in each level. The bracket notation is not a literal part of the resource string,
rather an explanation of what that string segment resembles. (ex `/{color}/{size}` signifies the first level of the path
is some color (“blue”, “green”, ect), and the second level is a size (“small”, “medium”, ect).

All of the following resources can be gathered as collections using the list action (see Listing Resources)

/system: Read only list of resources pertaining to the bus configuration. **Not currently implemented**

/user: Read only list of resources pertaining to the current user. **Not currently implemented**

/application: Read only list of applications registered to the bus.

/application/{id}: Read only information pertaining to an application registered to the bus.

Resource	Content Type	set	get	delete
/system	application/vnd.ozp-iwc-list-v1+json	no	yes*	no
/system/{id}	application/vnd.ozp-iwc-system-v1+json	no	yes	no
/user	application/vnd.ozp-iwc-list-v1+json	no	yes*	no
/user/{id}	application/vnd.ozp-iwc-user-v1+json	no	yes	no
/application	application/vnd.ozp-list-v1+json	no	yes*	no
/application/{id}	application/vnd.ozp-application-v1+json	no	yes	no

/system, /user, /application: these 3 resources when requested with a `get` action, will return the equivalent of
the `list` action on the resource path.

Requesting an action on a resource that does not fit the given resource structure will result in a response of "badResource":

```
{
    "ver": 1,
    "src": "system.api",
    "msgId": "p:1233",
    "time": 1435697455893,
    "response": "badResource",
    "entity": {
        "ver": 1,
        "src": "3de31e8b.a5eaa6134",
        "msgId": "p:88546",
        "time": 1435697461811,
        "dst": "system.api",
        "action": "get",
        "resource": "/hello",
        "entity": {}
    },
    "replyTo": "p:88546",
    "dst": "3de31e8b.a5efe614"
}
```

Why can't I register applications in the System API?

Application registration is meant to be isolated from the IWC. The bus does not care how an application was registered,
it gathers its listings from it's backend. This prevents the IWC from becoming a "application management tool" and
allows it to function simply as an "application usage tool".

Accessing the API

To call functions on the System API, reference `client.system()` on the connected client.

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

```
});

client.connect().then(function(){
    var systemApi = client.system();
});
```

Launching an Application Through the System API

Applications have the possibility to launch other applications in the IWC. Rather than just opening a link in a new tab,
the System API can be used to pass important information to the launching application much like how Android allows
passing serialized data to new activities.

To launch an application, simply call the `launch` action on the corresponding resource.

```
var systemApi = client.system();

systemApi.launch("/application/ea0c6018-4f12-410d-93b7-fe925b3a6ca2");
```

To launch an application with data passed to it:

```
var data = {
    "Hello": "world!"
};
systemApi.launch("/application/ea0c6018-4f12-410d-93b7-fe925b3a6ca2",{entity: data});
```

The launched application can gather the launch data after it's client has connected as so:

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

Names Api

Status of the bus. This api exposes information regarding applications connected to the bus. This is a read-only API.

Common Actions

All common API actions apply to the Names API, but only read actions are permitted.

get: Requests the Names API to return the resource stored with a specific key.

bulkGet: Requests the Names API to return multiple resources stored with a matching key.

set: Requests the Names API to store the given resource with the given key.

list: Requests the Names API to return a list of children keys pertaining to a specific key.

watch: Requests the Names API to respond any time the resource of the given key changes.

unwatch: Requests the Names API to stop responding to a specified watch request.

delete: Requests the Names API to remove a given resource.

Names API resource structure

The Names API relies on the path-level separation resource scheme to easily separate differences in resources.

The table below breaks down the differences in each level. The bracket notation is not a literal part of the resource string,
rather an explanation of what that string segment resembles. (ex `/{color}/{size}` signifies the first level of the path
is some color ("blue", "green", ect), and the second level is a size ("small", "medium", ect).

All of the following resources can be gathered as collections using the list action (see
Listing Resources)

/api/{address}: An API on the bus. Contains an array of actions pertaining to the API.

/address/{address}: A connection to the bus. All components of the IWC bus have an assigned address. Contains
information on the component assigned to the address.

/multicast/{group}/{member}: A group-specific multicast member. Multicast messages are distributed to all members
of the group (internal messaging not client messaging). Contains information on the component assigned to the address.

/router/{address}: A router connected to the IWC bus. Each window has a router to distribute messages amongst its
participants. Contains information on its participants.

Resource	Content Type	set	get	delete
/api/{address}	application/vnd.ozp-iwc-api-v1+json	no	yes	no
/address/{address}	application/vnd.ozp-iwc-address-v1+json	no	yes	no
/multicast/{group}/{member}	application/vnd.ozp-iwc-multicast-address-v1+json	no	yes	no
/router/{address}	application/vnd.ozp-iwc-router-v1+json	no	yes	no

Requesting an action on a resource that does not fit the given resource structure will result in a response of "badResource":

```
{
    "ver": 1,
    "src": "names.api",
    "msgId": "p:95269",
    "time": 1435697461893,
    "response": "badResource",
    "entity": {
        "ver": 1,
        "src": "3de31e8b.a5efe614",
        "msgId": "p:88546",
        "time": 1435697461811,
        "dst": "names.api",
        "action": "get",
        "resource": "/hello",
        "entity": {}
    },
    "replyTo": "p:88546",
    "dst": "3de31e8b.a5efe614"
}
```

Accessing the API

To call functions on the Names API, reference `client.names()` on the connected client.

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

client.connect().then(function(){
    var namesApi = client.names();
});
```

Frequently Asked Questions

What browser technologies does the IWC use to communicate?

The IWC has been designed to adapt to communication capabilities based on the users browser version. Currently
the IWC operates using local storage as its lowest level transport (cross domain client to client communications), and window.postMessage to communicate between the client libraries and the bus.

Why use the IWC over just window.postMessage?

PostMessage requires reference of the window which will receive the message, this limits communication to a point-to-point topology. The IWC at it's lowest level is a bus topology. It uses bus topology underneath it's PostMessage point-to-point (client library to it's bus connection) to communicate across all bus connections using
localStorage. This adds a layer of abstraction for the IWC-using widget such that it does not need to maintain knowledge of open IWC connections of different domains as the bus transport layer takes care of it.

Why can't I use the system API to register an application?

Application registration for a bus is on an administrative level. Deployed Ozone platforms are intended to maintain user based application registrations. The system API is read only.

How do I know if another IWC widget is registered to handle my widget's intent invocation?

Individual application listings in the system API contain information about intents the widget claims to be registered for. The IWC is aware of opened applications and will open a chooser window should more than 1 open
widget be registered to handle said intent.

Can the IWC open a widget to handle an my widget's intent invocation?

It is planned to direct the user to a choosing dialog for opening widgets to handle an intent invocation. It has not been implemented at this time.

Can the IWC retain my choice when choosing a widget to handle my widget's intent invocation?

It is planned to allow users to store their choices for intent decisions. This is in development but not available at this time.

Is there documentation on commonly used intents and the data they expect?

At the current stage of development there is not a definitive list of commonly used intents. As the platform continues
to grow this will be addressed and documented.

Adding an IWC Client to your application

An IWC client is an application's connection to an IWC bus. An IWC bus is always local to the user, but bound by the domain it is hosted from. For this example we will be using a predefined IWC bus hosted on the ozp-iwc github page `http://ozone-development.github.io/iwc`.

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.

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>
  ...
```

Javascript
The IWC library is encapsulated in the `ozpIwc` namespace.

Connecting your IWC client to an IWC bus

Connecting a client to a bus is as simple as instantiating a `client = new ozpIwc.Client(...)`.

JavaScript

From 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 ran 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), the client address is obtainable. 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);
});
```

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.

Syntactically, to make an api call to the **data.api** to **set** the value `{foo: "bar"}` to key `/buz` you would do the following.

```
client.api("data.api").set("/buz",{
   entity: {
       foo: "bar"
   }
});
```

This client api call-style holds true through all api's on the bus such that the following syntax structure is formed:

```
client.api(${api's name}).${the action}(${the resource key}, ${the resource value});
```

An advanced shorthand-call can be used to shorten the code length of an api call. `client.api(${api name})` can be simplified for any known api module to calling `client.${api name without '.api'}`

The example of the data.api set call above can be shortened to the following:

```
client.data().set("/buz", {
    entity: {
        foo: "bar"
    }
});
```

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.

IWC Client files

Bower: from your project's directory run `bower install ozone-development/ozp-iwc`

The IWC Client file you will reference from your application is at `bower_components/ozp-iwc/dist/js/ozpIwc-client.min.j`

Manually: grab the latest distribution of the IWC client from the IWC git repository and place it in your project's directory. (Location is arbitrary, just locatable from your html)

NOTE: For the remainder of this guide the IWC client will be referenced from the bower installed location

Making IWC Api Calls With expected Asynchronous Responses

Some api calls expect data returned, for example a `get` action. Because of this, the action call returns a promise. This allows operating on asynchronous IWC responses to be as easy as

```
client.data().get("/buz").then(function(response){
    console.log(response);
});
```

Response:

```
{
    foo: "bar"
}
```

Although, if making the `set` and `get` calls one after another, it is not guaranteed that the `set` finishes before the `get`. With the help of promises the order of operations can be enforced:

```
client.data().set("/buz",{
    entity: {
        foo: "bar"
    }
}).then(function(setReply){
    return client.api("data.api").get("/buz");
}).then(function(getReply){
    console.log(getReply.entity);
});
```

Response:

```
{
    foo: "bar"
}
```

Versioning

The IWC follows Semantic Versioning. To check the version of a deployed IWC bus, reference the `bower.json` or `package.json` file of the bower_components/ozp-iwc (version reference currently not available in the dist directory)

The IWC Components

The `dist` directory for the IWC is as follows

Note: *(removed* .min.js & *.js.map variants from this tree for readability)*

The minimum files required to host the IWC bus are as follows:

1. iframe_peer.html
2. ozpIwc-bus.js (or ozpIwc-bus.min.js)
3. defaultWiring.js
4. ozpIwc-client.js (or ozpIwc-bus.min.js)

Below is a breakdown of what purpose these files serve and if they are configurable (IWC structure configuration not styling). Further information on individual aspects are through given links.

File/Dir	Configurable?	Purpose
debugger.html	no	Built-in debugger of the IWC
debugger	no	Resources used for the built in debugger available on the IWC
doc	no	Directory of YUIDocs built from the IWC source code.
iframe_peer.html	yes	An invisible document used by the IWC client to reference the IWC Bus.
intentsChooser.html	no	A pop-up document used by the IWC client to get user interaction for handling inte
js/defaultWiring.js	yes	The wiring of IWC Bus components and the Ozone backend.
js/ozpIwc-bus.js	no	The IWC bus library.
js/ozpIwc-client.js	no	The IWC client library.
js/ozpIwc-metrics.js	no	The IWC metrics library.

iframe_peer.html

Below is an example of a configured iframe_peer.html.

```
<!DOCTYPE html>
<html>
<head>
    <title>Iframe Peer</title>
    <script type="text/javascript">
        // These should be customized by the deployment
        var ozpIwc = ozpIwc || {};
        ozpIwc.apiRootUrl = "https://www.owfgoss.org/ng/dev-alpha/mp/api";
        ozpIwc.marketplaceUsername= "testAdmin1";
        ozpIwc.marketplacePassword= "password";
        ozpIwc.runApis=true;
        ozpIwc.acceptPostMessageParticipants=true;
    </script>
    <script type="text/javascript" src="js/ozpIwc-bus.js"></script>
    <script type="text/javascript" src="js/defaultWiring.js"></script>
```

```
  </head>
  <body>
  </body>
</html>
```

Overview

An iframe_peer.html file is necessary for an IWC deployment. When a client application opens a connection to the IWC bus, it opens the domains `iframe_peer.html`. For an IWC client to connect to a deployed bus, **the iframe_peer.html file location decides the path the client connects to**.

File location

If the iframe_peer.html file is located at `http://ozone-development.github.io/iwc/iframe_peer.html` a client application can connect to the IWC bus with a peerUrl of `http://ozone-development.github.io/iwc`. See the example connection below:

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

For more information on IWC client aspects see [[IWC App Integration]]

The ozpIwc-bus.js & defaultWiring.js files can be hosted anywhere else on server so long as they:

1. remain within the same origin.
2. can be loaded by the iframe_peer.html

Configuration

The following IWC bus properties can be configured in the iframe_peer.html. Note that by file hierarchy iframe_peer.html is the first loaded entity of the IWC bus, thus declaring the `ozpIwc` namespace is necessary for configurations.

Property | Type | Default Value | Definition
———|——|————-|———
ozpIwc.apiRootUrl | String| “/api” | The location of Api backend root hal data index.json (relative or absolute)
ozpIwc.marketplaceUsername | String | “” | Basic authentication username for the backend (For development purposes only)
ozpIwc.marketplacePassword | String | “” | Basic authentication password for the backend (For development purposes only)
ozpIwc.runApis | Boolean | true | If false, api’s defined in the defaultWiring.js will not be loaded
ozpIwc.acceptPostMessageParticipants | Boolean | true | If false, the IWC bus will be confined to one browsing context.

Gathering the IWC components to Host

All parts of the IWC bus can be gathered via bower by running `bower install ozone-development/ozp-iwc` or adding the following to your bower.json dependencies

```
 "ozp-iwc": "ozone-development/ozp-iwc"
```

The required components will be in `bower_components/ozp-iwc/dist`

If gathering the IWC via bower is not possible, The dist directory from the master branch of this repository is what you will need to copy over to your hosting instance.

debugger.html

A built in debugger for the IWC. This application is packaged at the same directory as the iframe_peer.html.
The debugger to function must be in the same directory as `iframe_peer.html`