

iLab node.js

Instruction Manual

Written by Samuel Colbran
contact@samuco.net

Version 1.0 (6/2/2014)

Table of Contents

System Overview	1
1. Prerequisites	2
1.1 Command line	2
1.2 Node.js	2
2. Service Broker	3
2.1 Purpose	3
2.2 Installation	3
2.3 Lab Servers	4
2.3.1 Experiments	4
2.3.2 Adding a legacy lab server	5
2.4 Agents	6
2.4.1 Adding an agent	6
2.5 Modules	7
2.6 Plugins	7
2.6.1 Default Plugins	7
2.6.2 Writing Plugins	8
3. RESTful Interface	9
3.1 Actions	9
4. Agents	16
4.1 Purpose	16
4.2 Modules	16
4.3 Agent example	17
Hypothetical problem	17
Solution	17
4.4 Installation	18
4.5 Setup	18
4.5.1 Dual channel	19
4.5.2 Single channel	19
4.6 Plugins	20
4.6.1 Writing Plugins	20
4.7 Starting the Agent	20

System Overview

Figure 1 shows an example system overview is that contains several access routes for users. This example uses the MIT legacy lab servers and experiment servers. The broker provides a bridge between the json and legacy SOAP (see section 2).

The example broker is connected to two agents **Guest Access** and **Example Course** (see section 4). These agents demonstrate the new possibility for having separate HTML pages depending on the access levels of the user.

Two separate databases have been used to demonstrate services. The mongo database could be used to store all results from all experiments. The SQL database could be used to store the results related to particular users. Services can be used by multiple agents.

Note: Services are a future module. They have not yet been implemented.

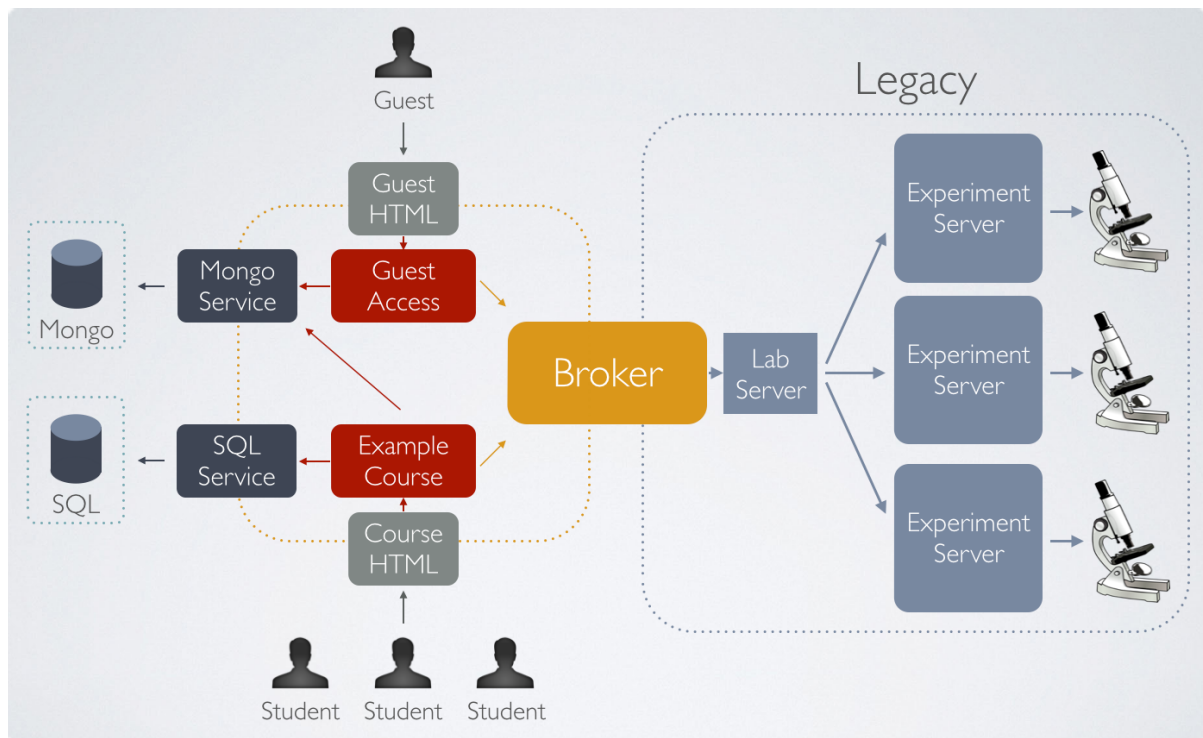


Figure 1. Example system overview

1. Prerequisites

1.1 Command line

The following instructions outline how to access the command line on your specific system.

Mac OS X

Open Terminal.app in /Applications/Utilities

Windows

Click the Start button. In the Search box, type Command Prompt, and then, in the list of results, double-click Command Prompt.

Linux

Open Terminal (located in Accessories in the Application menu on ubuntu). Other linux GUI's may have different command line software.

This guide requires considerable command line usage. The following box illustrates an example of a command that should be typed into the command line.

```
cd <example of command that should be typed>
```

1.2 Node.js

You will need to ensure that node.js has been installed on your computer before running any component of this package. Download the appropriate installer from <http://nodejs.org/download>. If you do not have access to a web browser or graphical user interface see <http://tinyurl.com/commandline-node>.

2. Service Broker

2.1 Purpose

The new service broker provides a **vastly different service** from the original MIT broker. All student interaction, such as authentication and client software, has been moved into a separate service (see 4.1 Agents). The new purpose of a service broker is to bridge communication between json and SOAP (for legacy lab servers).

The most basic broker converts json requests into SOAP and then sends it directly to the lab server. Additional caching or other logic may be incorporated.

The service broker provides a global administration to control access to lab servers. This is useful when you have several agents with different permissions.

2.2 Installation

If you have git installed on your system (see <https://help.github.com/articles/set-up-git>) you can use the following command to download the latest broker source code.

```
git clone https://github.com/ShadovvMoon/iLabServiceBroker.git
cd iLabServiceBroker/broker
```

Alternatively you can download the latest source [from GitHub](#), open a command window and then set the current directory to the 'broker' folder.

```
cd <path to broker directory>
```

To install the required dependencies, run the following command:

```
npm install
```

You can then start the broker by typing:

```
node index.js
```

Open a web browser and navigate to <http://localhost:8080>. Login with the username and password **admin** and **password** respectively. Click the **Admin** drop down menu in the upper right hand corner and then select **My Account**. Enter **password** as the old password, then type in a new password and click **Save**.

2.3 Lab Servers

Open a web browser and navigate to <http://localhost:8080> in a web browser. Login using the username and password that you set in the **installation** step. Click on the **Experiments** tab in the left side menu to bring up the Lab Server setup page.

2.3.1 Experiments

Experiments provides a useful overview for lab servers connected to your broker. Servers are colour coded depending on their status. A picture of the experiments tab is shown in figure 2.

- Blue** The lab server is successfully communicating with the broker.
- Yellow** The broker is trying to connect with the server.
- Red** The server is running, but has returned an error such as invalid passkey.

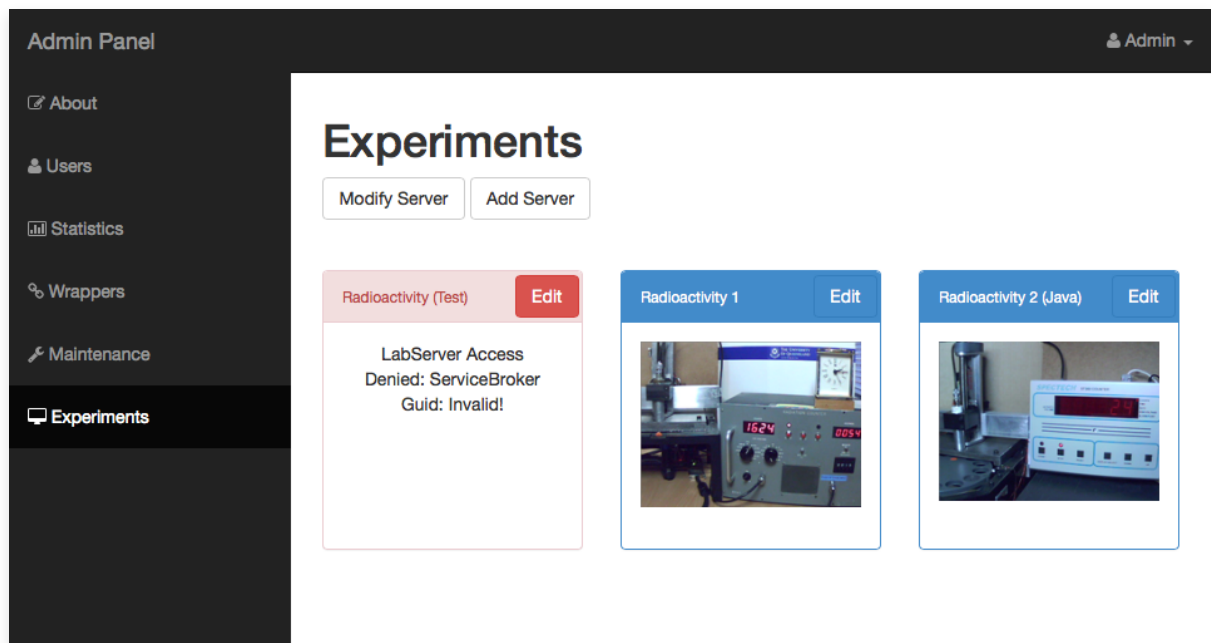


Figure 2. Experiments - main page.

2.3.2 Adding a legacy lab server

You will first need to add the new service broker (the guid can be found in the **About** tab) to the legacy lab server. Please contact your lab server admin to do this.

Click on the **Add Server** button in the **Experiments** tab.

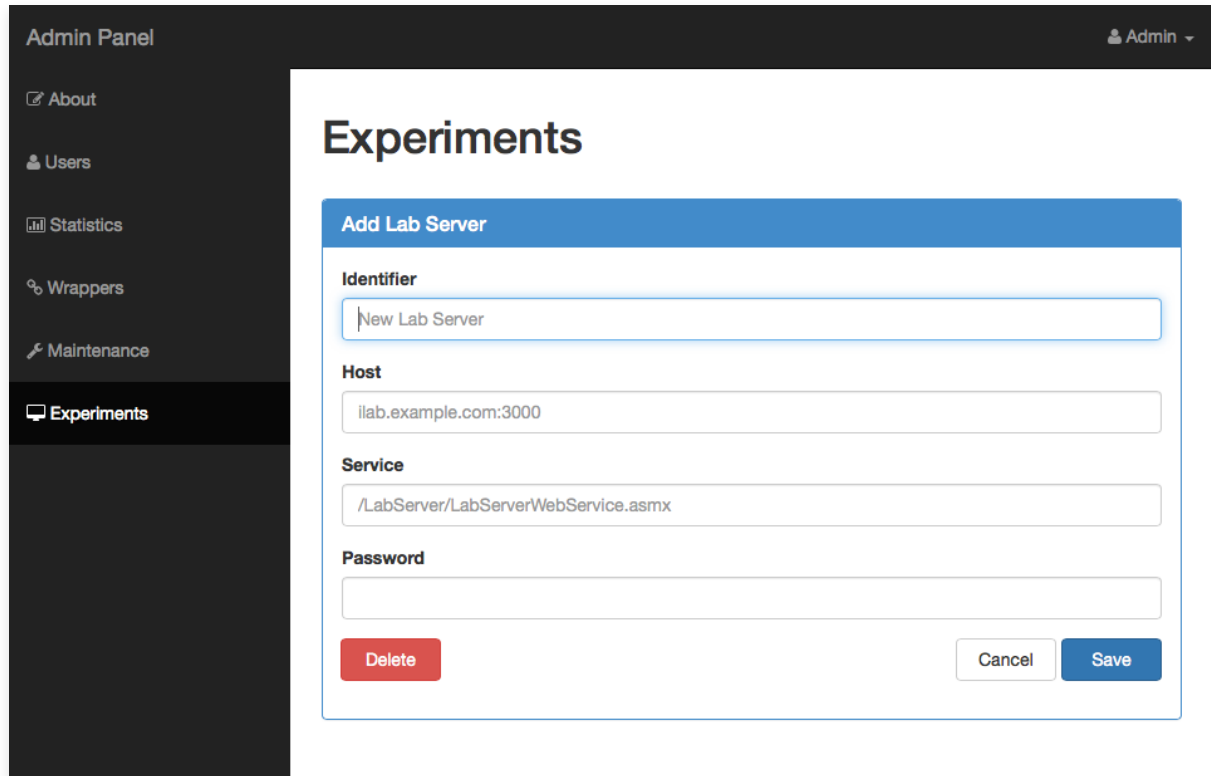
The screenshot shows the 'Admin Panel' interface. On the left is a dark sidebar with navigation links: 'About', 'Users', 'Statistics', 'Wrappers', 'Maintenance', and 'Experiments' (which is highlighted). The main content area is titled 'Experiments' and contains a form titled 'Add Lab Server'. The form has four input fields: 'Identifier' (containing 'New Lab Server'), 'Host' (containing 'ilab.example.com:3000'), 'Service' (containing '/LabServer/LabServerWebService.asmx'), and 'Password' (empty). At the bottom of the form are three buttons: 'Delete' (red), 'Cancel' (white), and 'Save' (blue).

Figure 3. Experiments - Adding a server

The identifier is a **unique** string that clients will use to access the lab server. It will appear throughout the admin interface and will also be returned by the `getLabList` broker method.

The host field should contain both the host and an optional port number for the lab server (separated by a colon). For an example lab server,
<http://ilab.example.com:3000/LabServer/LabServerWebService.asmx>

The host field would contain:
ilab.example.com:3000

The service field contains the path to the SOAP web service. From our example, the service field would be:
[/LabServer/LabServerWebService.asmx](#)

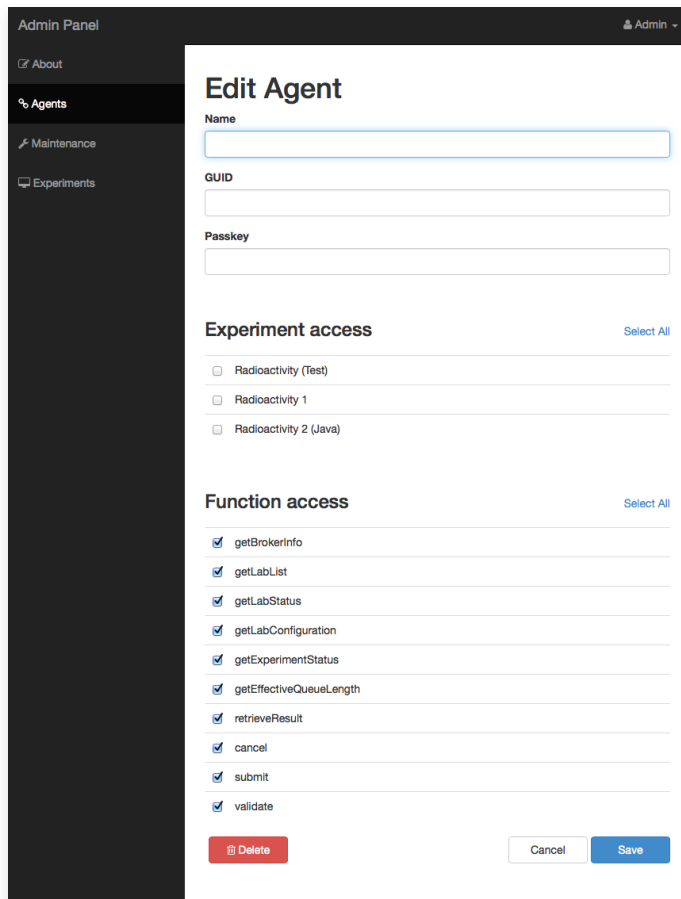
The password corresponds to the incoming passkey for the Lab Server. This is often tied to the service broker guid inside the lab server configuration.

2.4 Agents

An agent is designed to provide a way of ‘modifying broker behaviour’ without making any changes to the broker source code. This is useful for access control and keeping the system stable. See section 4 for more information about agents.

2.4.1 Adding an agent

Click on the **Add Agent** button in the **Agents** tab.



The screenshot shows the 'Edit Agent' form in the Admin Panel. The form has a dark sidebar on the left with navigation links: 'About', 'Agents', 'Maintenance', and 'Experiments'. The main content area is titled 'Edit Agent' and contains the following fields and sections:

- Name:** A text input field.
- GUID:** A text input field.
- Passkey:** A text input field.
- Experiment access:** A section with a 'Select All' link and three checkboxes:
 - ☐ Radioactivity (Test)
 - ☐ Radioactivity 1
 - ☐ Radioactivity 2 (Java)
- Function access:** A section with a 'Select All' link and ten checkboxes, all of which are checked:
 - ☒ getBrokerInfo
 - ☒ getLabList
 - ☒ getLabStatus
 - ☒ getLabConfiguration
 - ☒ getExperimentStatus
 - ☒ getEffectiveQueueLength
 - ☒ retrieveResult
 - ☒ cancel
 - ☒ submit
 - ☒ validate

At the bottom of the form are three buttons: a red 'Delete' button with a trash icon, a grey 'Cancel' button, and a blue 'Save' button.

Figure 4. Agent interface

The **name** field is a **unique** string that is used to identify the agent in the admin interface.

The **guid** is a **unique** string that identifies the agent. It corresponds to the value of `config.wrapper_uid` in the agent config.

The **passkey** is a string that corresponds to the value of `config.wrapper_key` in the agent config.

The **Experiment access** checkboxes determine which experiments can be accessed by the agent. Agents by default do not have access to any experiments. Experiments added after the agent are not accessible by the agent until the specific checkbox is ticked.

The **Function access** checkboxes determine which functions can be accessed by the agent. Agents by default have access to all functions.

2.5 Modules

The broker source code is broken up into several **modules** each controlling specific aspects of the broker. The main module for the broker is the **server**, which handles all broker communication. Communication channels are setup through the **express** module, which is stored in the **app** variable. These variables are important for writing **plugins**.

2.6 Plugins

Authentication plugins are used to implement a new authentication or communication method (such as Facebook, Google+ or your own system). They are stored inside the broker/net/auth/ folder. Plugins can be enabled or disabled by modifying the config.js inside the broker directory.

eg, the noauth plugin can be enabled by adding the following to the config file.

```
config.auth_plugins.push({
  name:      "noauth",
  file:      "noauth.js"
});
```

2.6.1 Default Plugins

The default service broker comes with three plugins - noauth, wrapper and admin.

noauth

The noauth plugin is disabled by default. It allows communication to the service broker and lab servers without requiring any authentication. Clients can communicate with the broker with both json and jsonp through /noauth-json and /noauth-jsonp respectively.

Note: Do not enable this plugin on a public service broker as it allows unrestricted access to all lab servers without any authentication.

admin

The admin plugin allows the admin interface to execute special commands on the service broker. It is recommended that you do not disable this plugin as it is required to use the admin interface.

wrapper

The wrapper plugin gives agents access to lab servers. It is recommended that you do not disable this plugin as it is required to communicate with agents.

2.6.2 Writing Plugins

The server module calls the **createAuth** function when the plugin is first loaded. Use this initialisation function to setup any communication channels to your plugin. If your plugin deems a client request to be acceptable (after checking authentication, access restrictions, input format etc), then you can parse a json dictionary to the **server** module containing all of the necessary request details.

The template below can be used as a starting point for a plugin.

```
var sys = require('sys');
var config = require('../../config');

(function () {
  var root = module.exports;
  function isAuthenticated(data)
  {
    return true;
  }
  function createAuth(app, server)
  {
    //Listener for JSONP
    app.get('/<your plugin>-jsonp', function(req, res)
    {
      if (isAuthenticated(req.query)){
        server.receiveDataFromClient({
          request:req,
          response:res,
          json:req.query,
          type:'jsonp'
        });
      }
    });

    //Listener for JSON
    app.post('/<your plugin>-json', function(req, res)
    {
      if (isAuthenticated(req.body)){
        server.receiveDataFromClient({
          request:req,
          response:res,
          json:req.body,
          type:'json'
        });
      }
    });

    //Add new listeners here
  }
  root.createAuth = createAuth;
})();
```

Note: The server.js only supports json input/output. If you wish to use an alternative input then you will need to convert it to json before passing it through to the server variable. The current version of the broker does not allow plugins to modify the output.

3. RESTful Interface

The node.js service broker and agents support both json and jsonp over the RESTful interface.

3.1 Actions

An action is a process on the server that can be called using json or jsonp. The required arguments are described below in detail for every action supported by the service broker and agents. An example json request to the service broker may look like:

```
{
  action: "getLabConfiguration",
  id: "example lab id"
}
```

The same request in jsonp would look like:

```
action=getLabConfiguration&id=example%20lab%20id
```

The following actions do not require any arguments other than the action name.

getBrokerInfo

Purpose

Returns information about the service broker such as vendor name.

Returns

```
{
  vendor: <name>
}
```

Where <name> is the string specified in the **About** tab of the Broker admin panel.

Example Request

```
{
  action: "getBrokerInfo"
}
```

Example Response

```
{"vendor":"Samuco iLab Broker"}
```

getLabList

Purpose

Returns a list of lab servers that are supported (and are allowed to see) by the service broker.

Returns

```
["example lab id", "example lab id 2"]
```

Where **example lab id** and **example lab id 2** are the unique identifiers for lab servers in the **experiments** tab.

Example Request

```
{
  action: "getLabList"
}
```

Example Response

```
["Radioactivity 1"]
```

The following actions also require an **id** argument. The id argument is a unique string that identifies the lab server, corresponding to the identifier in the experiments tab.

getLabConfiguration

Purpose

Gets the configuration of a lab server.

Returns

```
{
  labConfiguration: <configuration dictionary>
}
```

Example Request

```
{
  action: "getLabConfiguration",
  id : "Radioactivity 1"
}
```

Example Response

```
{"labConfiguration":{"$":
{"title":"Radioactivity","version":"4.0"},"labCamera":[{"url":["http://
ilab.example.com/Radioactivity.htm"]}], "labInfo":[{"text":[""], "url"
...etc
```

getLabStatus

Purpose

Checks on the status of a lab server.

Returns

```
{
  online: <true/false>,
  labStatusMessage: <status message string>
}
```

Example Request

```
{
  action: "getLabStatus",
  id : "Radioactivity 1"
}
```

Example Response

```
{"online":"true","labStatusMessage":"1:Powered down"}
```

getEffectiveQueueLength

Purpose

Checks the queue length of a lab server.

Returns

```
{
  effectiveQueueLength: <queue length string>,
  estWait: <estimated wait string>
}
```

Example Request

```
{
  action: "getEffectiveQueueLength",
  id : "Radioactivity 1"
}
```

Example Response

```
{"effectiveQueueLength":"10","estWait":"2450"}
```

Submit

Purpose

Submits an experiment specification to a lab server for validation and execution.

Requires

```
{
  experimentSpecification: <escaped xml string>
}
```

Returns

```
{
  vReport:
  [{
    accepted: <true/false>,
    estRuntime: <estimated runtime string>
  }]
  experimentID: <token identifying the experiment>
  minTimeToLive: <time before this experiment data is deleted>
  wait:
  [{
    effectiveQueueLength: <experiment position in queue>
    estWait: <estimated wait time string until experiment runs>
  }]
}
```

Example Request

```
{
  action:"submit",
  id:"Radioactivity 1",
  experimentSpecification:"<setupName>Radioactivity versus Distance</
setupName><setupId>RadioactivityVsDistance</
setupId><sourceName>Strontium-90</sourceName><absorberName>None</
absorberName><distance>15,20</distance><duration>1</duration><repeat>1</
repeat>"
}
```

Example Response

```
{ "vReport":
  [{"accepted":"true","estRuntime":"58"}], "experimentID":"63", "minTimeToLive": "0", "wait": [{"effectiveQueueLength":"1", "estWait":"0"}] }
```

Validate

Purpose

Checks whether the experiment specification would be accepted if submitted for execution.

Requires

```
{
  experimentSpecification: <escaped xml string>
}
```

Returns

```
{
  accepted: <true/false>,
  estRuntime: <estimated runtime string>
}
```

Example Request

```
{
  action:"validate",
  id:"Radioactivity 1",
  experimentSpecification:"<setupName>Radioactivity versus Distance</
setupName><setupId>RadioactivityVsDistance</
setupId><sourceName>Strontium-90</sourceName><absorberName>None</
absorberName><distance>15,20</distance><duration>1</duration><repeat>1</
repeat>"
}
```

Example Response

```
{"accepted":"true","estRuntime":"58"}
```

The following actions require both an **id** and **experimentID** arguments.

retrieveResult

Purpose

Retrieves the results from (or errors generated by) a previously submitted experiment.

Returns

```
{
  statusCode: <status>,
  experimentResults: <result xml string>,
  xmlResultExtension: <optional string>,
  xmlBlobExtension: <optional string>,
  warningMessages: <optional string>,
  errorMessage: <string returned status is 4>
}
```

Where the value of <status> means

- 1 - The experiment is still waiting in the queue
- 2 - The experiment is currently running
- 3 - The experiment finished successfully
- 4 - The experiment ran but had errors
- 5 - The experiment was cancelled before running
- 6 - Invalid experiment ID

Example Request

```
{
  action: "retrieveResult",
  id : "Radioactivity 1",
  experimentID: 3
}
```

Example Response

```
{"statusCode": "3", "experimentResults": "<experimentResult><timestamp>Mon Feb 03 10:47:09 GMT+10:00 2014</timestamp><title>Radioactivity 1</title><version>4.0</version><experimentId>63</experimentId><sbName>Example iLab</sbName><unitId>1</unitId><setupName>Radioactivity versus Distance</setupName><setupId>RadioactivityVsDistance</setupId><sourceName>Strontium-90</sourceName><absorberName>None</absorberName><distance>15,20,30</distance><duration>1</duration><repeat>1</repeat><dataType>Real</dataType><dataVector distance= \"15\">44</dataVector><dataVector distance= \"20\">33</dataVector><dataVector distance= \"30\">30</dataVector></experimentResult>"}
```


Cancel

Purpose

Cancels a previously submitted experiment. If the experiment is already running, the cancel action endeavours to abort execution, but there is no guarantee that the experiment will not run to completion.

Returns

```
{
  cancelled: <true/false>
}
```

Example Request

```
{
  action:"cancel",
  id:"Radioactivity 1",
  experimentID: 3
}
```

Example Response

```
{"cancelled":"true"}
```

4. Agents

4.1 Purpose

An agent is designed to provide a way of ‘modifying broker behaviour’ without making any changes to the broker source code. This is useful for access control and keeping the system stable.

All actions supported by the broker are also supported by the agent. The most basic agent acts as a wrapper for commands (simply passing commands through to the broker). A more advanced agent could introduce logic inside commands or other authentication systems.

4.2 Modules

The agent source code is broken up into several **modules** controlling specific aspects of the agent. The main module for the agent is the **core**, which handles all agent communication. Communication channels are setup through the **express** module, which is stored in the **core.app** variable. These variables are important for writing **plugins** (see section 4.6). It is recommended that you do not modify the **core** module as it will change with future updates.

The core module has several functions to assist with plugin development. Some useful functions are:

`sendReplyToClient(client, json)`

Sends the json dictionary to the client in the appropriate format.

`receiveDataFromClient(client)`

Handle the json input from the client and send out a response assuming full authentication.

`rejectDataFromClient`

Sends a json dictionary to the client simulating a rejection response. This is useful for rejecting experiment specifications which are deemed unacceptable.

`javascriptToken`

Returns javascript containing an access token that can be added to a HTML page. This is useful for authenticating anything coming from the client webpage to the agent.

4.3 Agent example

Hypothetical problem

You need a large amount of data to get an accurate picture of the overall experiment and therefore want to create a way for anyone on the internet to access your experiment. A set of restrictions needs to be added to experiments to ensure that they do not run for hours. You also want to be able to skip the experiment queue to run your own experiments.

Solution

Setup an agent with a plugin that does not require any authentication. Link this plugin to a HTML page showing your interface for the experiment. Inside the `override.js` file, add the following code to the `receiveDataFromClient` function:

```
//Reject any experiments that take longer than 60 seconds.
var allowed_time_seconds = 60;
if (client.json.action == 'submit' || client.json.action == 'validate')
{
    var lab_id = client.json.id;
    var specification = client.json.experimentSpecification;

    var responseFunction = (function(response_client){
        return function(obj, err){
            if (obj['accepted'] == true)
            {
                if (parseInt(obj['estWait']) <= allowed_time_seconds)
                {
                    if (client.json.action == 'validate')
                        core.sendReplyToClient(response_client, obj);
                    else
                        core.receiveDataFromClient(client);
                }
                else
                    core.rejectDataFromClient(client);
            }
        };
    })(client);
    core.sendActionToServer({id:lab_id, action:'validate',
        experimentSpecification:specification}, responseFunction);
}
else
{
    core.receiveDataFromClient(client);
}
```

This code adds an additional check to the validate and submit functions. The lab server is asked to validate the experiment specification and return an estimated execution time. If the execution time is less than 60 seconds then the action will continue as normal. If not, then the agent will inform the client that validation has failed.

4.4 Installation

If you have git installed on your system (see <https://help.github.com/articles/set-up-git>), you may use the following command to download the latest broker source code.

```
git clone https://github.com/ShadovvMoon/iLabServiceBroker.git
cd iLabServiceBroker/wrapper
```

Alternatively, you may download the latest source from GitHub, open a command window and then set the current directory to the 'wrapper' folder.

```
cd <path to agent directory>
```

To install the required dependencies, run the following command:

```
npm install
```

4.5 Setup

To successfully start the agent you will need to complete the required fields shown below in the config.js file:

```
config.broker_host    = 'localhost';
config.broker_port    = 8080;

//Agent info
config.wrapper_uid    = '';
config.wrapper_key    = '';
```

The wrapper_uid and wrapper_key correspond to the GUID and Passkey in the broker admin panel. These do not need to follow any set format (a random string of any length is suitable). ASCII characters are required.

4.5.1 Dual channel

Agents use two secure communication channels with the broker. The first channel (agent to broker) is used to forward any json/jsonp action to the broker. The second channel (broker to agent) allows the broker to send information directly to the agent when it becomes available. The second channel is used to prevent unnecessary polling of the broker.



Using the dual channel mode will require the following settings in the config.js file:

```
config.simple_wrapper = false
config.wrapper_host    = 'localhost';
config.wrapper_port    = 3000;
```

The wrapper_host and wrapper_port settings should **not** be localised, unless the broker is located on the same machine. This address needs to be accessible by the broker.

4.5.2 Single channel

A single channel agent does not have the secondary communication channel that allows the broker to directly send information to the agent. The only reason to use this mode is when the agent has a dynamic address or cannot be reached by the broker due to firewalls.



You can enable the simple agent mode by setting the following in the config file:

```
config.simple_wrapper = true;
```

4.6 Plugins

Plugins are used when you wish to extend the agent without modifying any core code. This may include adding a new authentication method (such as Facebook, Google+ or your own system) or a custom handler of events. Plugins are stored inside the `wrapper/plugins/` folder. Plugins can be enabled or disabled by modifying the config file.

For example, the blackboard plugin can be enabled by adding the following to the config file.

```
config.plugins.push({
  name: "blackboard",
  settings: {consumer_key: '', shared_secret: ''}
});
```

4.6.1 Writing Plugins

The core module calls the **setupPlugin** function when the plugin is first loaded. Use this initialisation function to setup any communication channels to your plugin. If your plugin deems a client request to be acceptable (after checking authentication, access restrictions, input format etc), then you can parse a json dictionary to the **core** module containing all of the necessary request details.

You can also serve up resources such as a HTML page. The javascript on these pages is given an access token which can be used to communicate with the agent.

The template below can be used as a starting point for a plugin.

```
(function () {
  var root = module.exports;
  function setupPlugin(core, settings)
  {
    var app = core.app;
    var plugin_port = app.get('port');

    /*Your actions here*/
  }
  root.setupPlugin = setupPlugin;
})();
```

Note: The configjs only supports json input/output. If you wish to use an alternative input, first convert it to json before passing it through to the config variable. The current version of the agent does not allow plugins to modify the output.

4.7 Starting the Agent

You can start the agent by typing

```
node index.js
```