

Development documentation

Graphic interface part



XIAO DENG

Télécom paristech, juin 2016

Project PRIM

Table of Contents

[How to install 3](#_Toc454819672)

[Before installing 3](#_Toc454819673)

[Begin to install 3](#_Toc454819674)

[Application structure 4](#_Toc454819675)

[Basic components 5](#_Toc454819676)

[Object and object class (node) 5](#_Toc454819677)

[Rule (edge) 6](#_Toc454819678)

[Environment 6](#_Toc454819679)

[User 7](#_Toc454819680)

[Session 7](#_Toc454819681)

[Package structure and files 7](#_Toc454819682)

[Database 8](#_Toc454819683)

[Folders and files’ functions 9](#_Toc454819684)

# How to install

## Before installing

To run and develop this application, first you need to ensure that all the following software is installed correctly and works well:

* **Crossbar.io**

Crossbar.io is a networking platform for distributed and microservice applications, implementing the open [Web Application Messaging Protocol (WAMP)](http://wamp-proto.org/).

You can download it from its website <http://crossbar.io/>. When successful, the installation will have created a ***crossbar***command line tool, the path to the crossbar executable will depend on your environment. You can then verify the install by running ‘***crossbar version****’*, which lists the software versions of important Crossbar.io components

* **Node.js**

Node.js is a JavaScript runtime built on [Chrome's V8 JavaScript engine](https://developers.google.com/v8/).

To install it and to know how it works with crossbar.io, see:

“Getting started with JavaScript in Nodejs” : <http://crossbar.io/docs/Getting-started-with-NodeJS/>

* **ibmRationalSDL**

This is used to create SDL server stimulation. To install it, just download the compressed file, uncompressing it and put it in a directory, remember the path.

Begin to install

1. Download the install package from Github, then uncompressing it to a directory.

<https://github.com/dengxiao0509/PRIM.git>

*\*In this package you have the source code and all the JavaScript libraries needed.*

*You could find the structure of this package later in the documentation.*

1. Change the path **‘../ibmRationalSDL/bin/telelogic.profile’** in file “/PRIM/node/myscript.sh” to yours.
2. Open the terminal and go to the path ../PRIM. Run command ‘crossbar start’.

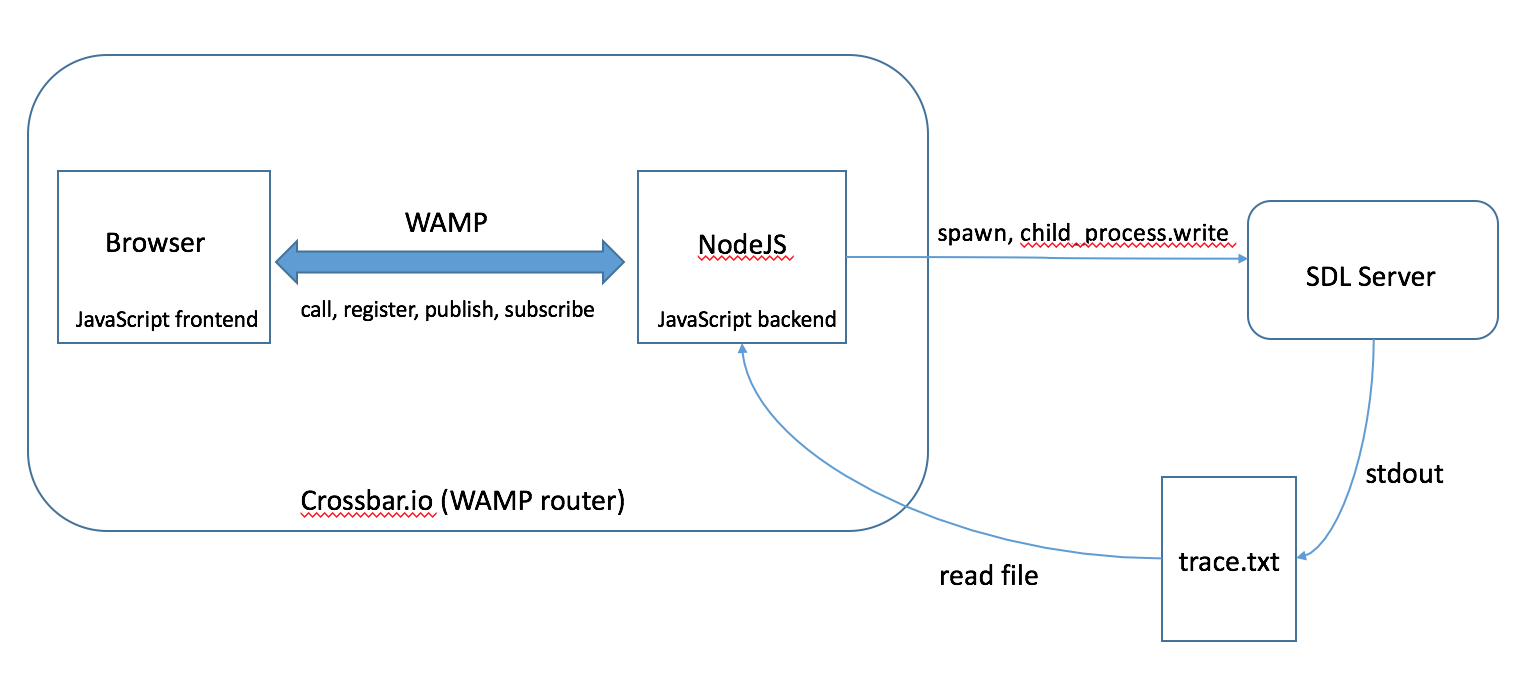
Make sure no errors appearing in terminal. At the end you should see several procedures registered messages.

1. Open a navigator (Chrome or Firefox), go to address localhost:8080.

That it!

*\*Please make sure you have access to Internet. Because we need some online JavaScript source.*

# Application structure



*figure 1. Application Structure*

Firstly, we use Crossbar.io to generate a WAMP application with a JavaScript frontend and backend. Both components use the open source library **AutobahnJS** to provide WAMP functionality. We run the whole application with Crossbar.io serving as a WAMP router, static Web server for the frontend files and JavaScript/NodeJS application component host for the backend code.

The backend runs under NodeJS, the frontend in the browser. The frontend and backend components will talk with each other using all four main interactions available in WAMP:

* call a remote procedure
* register a procedure for remote calling
* publish an event to a topic
* subscribe to a topic to receive events

Secondly, to interact with the SDL server, NodeJS spawns a child process where runs the SDL server. Then, NodeJS writes messages to the stdin of child process. In the same time, NodeJS creates a listener which listens to the child process stdout change event, which means the output of SDL server will be redirected to a file called trace.txt. After that, NodeJS reads the file trace.txt to get the output of SDL server.

# Basic components

This graphical user interface concerns 1 web page.

To simulate the Internet of Things, we introduce notions of **object, object class, rule, environment** and **session**.

## Object and object class (node)

An **object** may be an electric appliance like a heater or a sensor, in the GUI, it is represented by a **node**.

Node JSON structure :

{

"label":"tSensor1", //node name

"id":"6p20b0ij1q2t", //node id, generated randomly

"devicestate":"off", //device state

"shape":"dot", //node shape

"size":20, //node size

"description":"this is a temperature sensor.", //description

"variables":{"env":["internal temp","external temp"],"state":"off"},

// env is an array which contains names of environment variables

"title":" <div><div>label: tSensor1</div><div>description : this is a temperature sensor.</div><div> --- variables --- </div><div>env : internal temp,external temp</div><div>state : off</div></div>", //content shown when hover over node

"x":-225, //node position in the graph

"y":-155

}

To make adding nodes easier, we introduced the concept of **node class**. It’s like a template, users could create a node from a node class by providing only a node name and device state. Other fields will be copied from the node class.

Node class JSON structure:

{

"id":"8g4kg0umovpm",

"label":"temp sensor",

"shape":"dot",

"size":20,

"description":"this is a temperature sensor.",

"variables":{"env":["internal temp","external temp"],"state":"off"

}

## Rule (edge)

A rule is the connection between several objects. It’s represented graphically by **edges**, arrows from all sensors to all effectors. **Sensor** signifies the condition (like if) and **effector** represents the action (like then).

Rule JSON structure:

{

"id":"4uhqk0bbfhe7",

"user":"admin", //user who creates this rule

"name":"admin\_rule", //rule name

"color":"#2ae957", //color of rule’s edges

"edges":["6s3e4.up3ku","7e4q1.5143uq"], //array of all edges’ id

"if": //condition

{

"71jdt0g8qa2c":[["brightness","<","20"],["brightness","<","20"]]

//node\_Id: [[variable1,comparator1,value1],[variable2,comparator2,value2],…]

},

"then": //action

{

"nln90vkv91so":[["state","=","on"],["power","=","200"]],

"826oe005s72c":[["state","=","on"],["power","=","100"]]

//node\_Id: [[variable1,”**=**”,value1],[]…]

}

}

Edge JSON structure:

{

"id":"6s3e4.up3ku",

"from":"71jdt0g8qa2c", //sensor node id

"to":"nln90vkv91so", //effector node id

"color":"#2ae957",

"title":"admin\_rule" //belonging rule name

}

## Environment

In this simulation system, users could define some environment variables, like temperature. Objects could include them as variables.

Environment variable JSON structure:

{

"internal temp":{"value":"10"}, //variable\_name:{“**value**”: variable\_value}

"external temp":{"value":"9"},

"humunity":{"value":"4"}

}

## User

User is authenticated by his login username. To create connection to WAMP router, user must log in first. We offer the possibility of registering.

Since now we defined 2 different kinds of users: administrator (username is “admin”) and other users. Consequently, there exists 2 different user interfaces.

User JSON structure:

{

"xdeng":{"secret":"xdengxdeng","role":"frontend"},

"admin":{"secret":"adminadmin","role":"frontend"}

//user\_name:{“**secret**”: password, ”**role**”:”**frontend**} , ‘role’ is always ‘frontend’

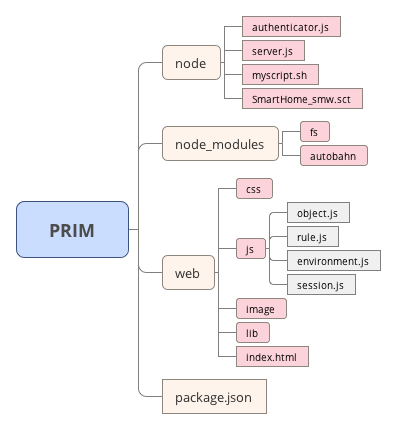
}

## Session

A session includes all object, object class, rule, environment data (data.txt), as well as history of commands sent to SDL server(cmds.txt), correspondences between nodes’ IDs (graph) and PIDs (SDL server).

A session is represented by a **folder** in the path ‘/tmp/savedSessions’ whose name is the session name. In this folder there are 3 files : data.txt, cmds.txt and nodeid\_pid.txt. Users could **save**, **open** a session or create a **new** session. All in all, sessions provide the possibility of saving current IoT and reload an IoT created before.

# Package structure and files



*figure2. package structure*

*(\*Nodes with round borders are folders, others are files.)*

## Database

These files will be created in the system directory /tmp automatically. Note that when you restart your computer, all data will be lost.

* **data.txt**

JSON string with node(object), rule(edge), environment, user data.

{

“nodeClasses”:

{“node class id”:{NodeClassJSON},

…

},

“nodes”:

{“node id”: {NodeJSON},

…

},

“edges”:

{“edge id”:{EdgeJSON},

…

},

“rules”:

{“rule id”:{RuleJSON},

…

},

“env”:{

EnvJSON,

…

}

“users”:{

UserJSON,

…

},

“currentSession”: session\_name,

“isConnectSDL”: true

}

* **cmds.txt**

List of all commands written to SDL server.

* **trace.txt**

Trace of SDL server inputs and outputs.

* **pid\_nodeId.txt**

For all objects: Correspondence between node id (graph) and pid (SDL server).

{node\_id1:pid1,node\_id2:pid2, …}

* **users.txt**

All registered users’ usernames and passwords, in the format of User JSON.

## Folders and files’ functions

* **/node**

In this folder, there are 2 JavaScript files used to create 2 NodeJS backend applications serving in web server.

* + **server.js**

Component used for checking data validity, writing data to text files and reading data from text files according to requests sent from browsers.

* + **authenticator.js**

Component used for authentication, including registering users and checking user name/password. Registered users’ information is stored in the file user.txt.

\*Crossbar.io supports authenticating WAMP sessions using different mechanisms ( <http://crossbar.io/docs/Challenge-Response-Authentication/> ). The one we used is WAMP-CRA.

* **/node\_modules**

All the NodeJS libraries used, including AutobahnJS, fs, etc.

* **/web**

Here stored all frontend code and resources.

* + **index.html**

Display of the main page. Including all JavaScript files in /js to create interactive page. Also, it includes functions about user authentication, to open a WAMP session, to subscribe procedures, to draw the graph, etc.

* + **/js**

JavaScript files to handle interactive events, to send requests to web server and to process responses from web server.

The following 4 files have the same code structure (see code comments). Take the first one for instance.

* + - **object.js**

All JavaScript code concerning objects and object classes.

There are 3 popups related:

* + - * addObjClassPopup: add/edit object class
      * newNodePopup: add object
      * editNodePopup: edit object

In this script, we defined all related icons’ and buttons’ click event handler. Normally, including opening, submitting (‘save’) and hiding(‘cancel’) popup.

Most importantly, submit event handlers are defined here. For submitting a form, we call a remote procedure registered in server.js to change data in database. Then server.js will publish to all subscribed browsers with renewed data, which realizes synchronization between all views.

In the end, some procedures subscribed (in index.html) to topic of objects and object classes changes in database are defined.

* + - **rule.js**
* addRulePopup : add/edit rule
* addRuleChooseSE : choose sensors and effectors
  + - **environment.js**
* addEnvPopup : add environment variable(s)
* editEnvPopup : edit an environment variable
  + - **session.js**
      * saveSessionPopup : save session
      * openSessionPopup : open session
  + **/css**

All style sheet files.

For example, ‘rule.css’ includes styles of elements in rule popups (add/edit rule); ‘popup.css’ includes styles of common elements in all popups.

* + **/image**

All icons and images.

* + **/lib**

All local JavaScript libraries. Note that we have used several online resources:

* **autobahn.min.jgz**

Including WAMP functions, like open/close session, call/register and subscribe/publish procedures. Documentation: <https://github.com/crossbario/autobahn-js>.

* + - **jquery-ui.js**

Documentation: <https://jqueryui.com/>.

* + - **vis.min.js**

A dynamic, browser based visualization library. We use it to draw graph. Documentation: <http://visjs.org/> .

* + - **jquery.leanModal.min.js**

JQuery plugin to manage popups. We use it to manage login/register popup. Documentation: <http://leanmodal.finelysliced.com.au/>.