Cupd Documentation

Release 1.0

Thomas Pelletier

February 09, 2011

CONTENTS

Cupd is a real time, nodejs-powered and websocket-based dashboard server. It exposes an API for services, which publish data, and embeds a small web server to display a dashboard to the real users. Basically, it allows you to plug any kind of data source (RSS feeds, IRC chat, web statistics, video streaming and so on) and broadcast it to an unlimited number of users, through an highly customizable dashboard interface.

CHAPTER

ONE

QUICKSTART

You must have nodejs and a websocket-capable web browser installed. Once you are ready, do something like this:

```
$ git clone git://github.com/pelletier/cupd.git
$ cd cupd
$ node app/cupd.js
```

Your Cupd dashboard is now available on http://localhost:3000/. However, without services (ie, data sources), it is not really interesting. Fire another terminal, and:

```
$ cd cupd/plugins/hello_world
$ ruby hello_world.rb
```

Once the plugin is initialized, you can switch back to your web browser and see a nice `Hello world` with the local time running above.

CHAPTER

TWO

QUESTIONS? WANT TO GET INVOLVED?

Pick one:

- Fork and request a pull on GitHub.
- Drop me an email.
- Get in touch on Twitter.

CHAPTER

THREE

CONTENTS

3.1 Installation

Before serving amazing real-time dashboards to the world, you need to download and install Cupd's dependencies which are NodeJS and ExpressJS.

3.1.1 NodeJS

Because the server side of Cupd is built on top of NodeJS you need to install it before starting. The easiest way is to check your operating system documentation and install a prepackaged version of NodeJS. Otherwise you can build and install it from source.

3.1.2 ExpressJS

Cupd embeds a small web server which heavily uses ExpressJS therefore you will obviously need it. According to their installation guide you can use one of the following way to install it:

With curl:

```
$ curl -# http://expressjs.com/install.sh | sh
```

With npm:

```
$ npm install express
```

Or building from the Git trunk:

```
$ git clone https://github.com/visionmedia/express.git
$ git submodule update --init
$ make install
$ make install-support
```

3.2 Services API

Services API is the entry point for data providers. It is built on top of websocket. So first of all, pick a websocket **client** library for your favorite language:

- Python: http://github.com/mtah/python-websocket
- Ruby: http://github.com/gimite/web-socket-ruby

- Javascript: http://github.com/guille/node.websocket.js/
- Java: http://github.com/adamac/Java-WebSocket-client
- Erlang: http://github.com/davebryson/erlang_websocket
- ...

Once you've got familiar with it read further.

3.2.1 How it works

It is pretty easy: first, you do the handshake, then you send payloads of data. All the exchanges are JSON-formatted, and must follow a given structure.

3.2.2 Handshake

Once you are connected to the server, it will send you the following message:

```
{
    "type": "welcome",
    "uid": "ACA764FEF7ED43D3BF7D31472B952987"
}
```

The `type` field must be in every single message. The `welcome` message basically allows you a unique id. You **must** retain the given uuid, because all the messages *you* are going to send must contain an `uid` field which contains this string.

You now have to authenticate.

> **Note:** For now, authenticating is not yet fully designed. We plan to add a public /private keys authentication here.

Send a message like the following:

```
{
    "type": "auth",
    "uid": "ACA764FEF7ED43D3BF7D31472B952987",
    "name": "your_app",
}
```

Your application name must be consistent with the one you use in your widget.

> **Note:** The authentication will enable a strict check of the uniqueness of your application name in a future release.

3.2.3 Sending data

Now that the Cupd server trusts you (ahem), you can start sending payloads of data. To do so, just send a message like the following:

```
{
    "type": "data",
    "data": <Your JSON formated data>
}
```

Pretty simple, yes? You data can obviously be a string, an hash, a list and so on. Your widget will receive the exact content of the `data` field, but de-serialized.

3.3 Widgets

Your widget is the dedicated area on which you will display your very interesting data on the screens of your customers (I mean end-users).

Here is a widget example (in fact, it is the hello_world example):

```
/*
 * Hello world
 *
 * An example widget for Cupd.
 * It is meant to show the minimal requirements for building a working Cupd
 * widget. Please read the corresponding wiki pages for more informations.
 */

/* Your exported code must stay in a single object */
my_code = {};

/* The refresh function is called by the dashboard to force your widget to
 * display nicely (for instance when the user loads the page, or changes
 * critical display settings).
 */
my_code.refresh = function(){
    var me = $('#'+this.name);
    me.html("<h3>Hello, world!</h3><p></p>");
    me.css('color', 'white');
};

/* The update function is called when your widget receives some informations
 * from its server-side friendly plugin.
 * The data argument is the raw data received by the dashboard.
 */
my_code.update = function(data){
    $('#'+this.name+' p').html(data.text);
};

/* Finaly register your code into the client's dashboard code base.
 * 'hello_world' is the slugified name of the plugin. It is the plugin
 * identifier, so stay consistent otherwise your code will be doomed.
 */
code_base['hello_world'] = my_code;
```

Widgets are plain javascript. However, please note two things: firstly, the dashboard is jQuery powered, hence it is available to you, so feel free to use it . Secondly, Cupd exposes an API and provides some guidelines. It is very recommended to use them *both*.

Concerning the file system, your widget has a dedicated space to live in: `/plugins/<your_widget_name>/` (I will refer to it has "your widget space" in the rest of the document").

> **Guideline:** Use only alphanumeric characters and underscores in your widget name. And avoid using spaces too.

3.3.1 Needed skeleton

You absolutely need one file in your widget space: `widget.js`. It will contains your code which will handle display refreshes and data payloads. This code is loaded dynamically by the user's browser. You must create an empty object which is going to be registered to the dashboard. It must contains two functions: `update(data)` and `refresh()`.

Then you must register your widget to the dashboard using the `code_base` global variable. Let's dig a little deeper on all of this.

3.3.2 Your widget object

In all your functions, you can use the `this` statement to refer to your widget. It is the same object you register with the `code_base` variable, but it also contains:

- The `name` property, which is unique ID generated by the dashboard to refer as the current instance of your widget.
- (nothing else for now: more to come)

3.3.3 Handling refreshes

First of all, you need to know that when your widget is loaded (ie, when the users connects to Cupd), the dashboard automatically creates a `div` on the page, with your widget's name as its `id` field. What is going to be displayed in this div is up to you, and the `refresh()` method.

Example:

```
my_code.refresh = function(){
    var me = $('#'+this.name); // We first grab that so important div
    me.html("<h3>Hello, world!</h3><p></p>"); // Let's display the skeleton
                                              // of widget.
    me.css('color', 'white'); // Add some style
};
```

Not really difficult. The refresh function is not called very often: only when the user loads the page, and if he plays enough with its window this so that the engine is not able to redraw widgets properly (which is rather rare).

3.3.4 Handling incoming data

Because you want to display data, you must receive them first. It is the main purpose of the `update(data)` method. It is fired each time some new data for your widget is received by the dashboard. `data` is a Javascript object: the payload sent by your service has been de-serialized by the dashboard.

Example:

```
my_code.update = function(data){
    $('#'+this.name+' p').html(data.text);
};
```

In this example, the widget receives an hash, containing one key: `text`, so it changes the value of the previously created <p> (see refresh) to the freshly received text.

3.3.5 Registering

It's the easiest part, and also the most important. Once all your methods have been implemented, you must register your widget with its proper name using the global `code_base` hash.

Example:

```
code_block['my_widget_name'] = my_code;
```

And... it is over! You've done your job and your users are ready to enjoy your real time data.

- *search*