Centrifuge

Release 0.8.4

December 09, 2016

Contents

WARNING!!! Centrifuge migrated to Go language!!! See new documentation - https://github.com/centrifugal/centrifugo

Simple real-time messaging in web applications.

CHAPTER 1

Overview

Centrifuge is a server for real-time messaging in web applications.

Centrifuge server migrated to Go language - it's now called Centrifugo and lives in another repo.

If someone wants to be Centrifuge maintainer to make it compatible with entire Centrifugal stack – write me email.

Version 0.8 is the last release of Centrifuge with new features. Here is a list of libraries versions compatible with Centrifuge:

- centrifuge-js 0.9.0
- cent v0.6.0
- adjacent v0.3.0
- web v0.1.0
- examples v0.1.0
- phpcent 0.6.1
- centrifuge-ruby v0.1.0

Please, see new documentation for the entire Centrifugal stack.

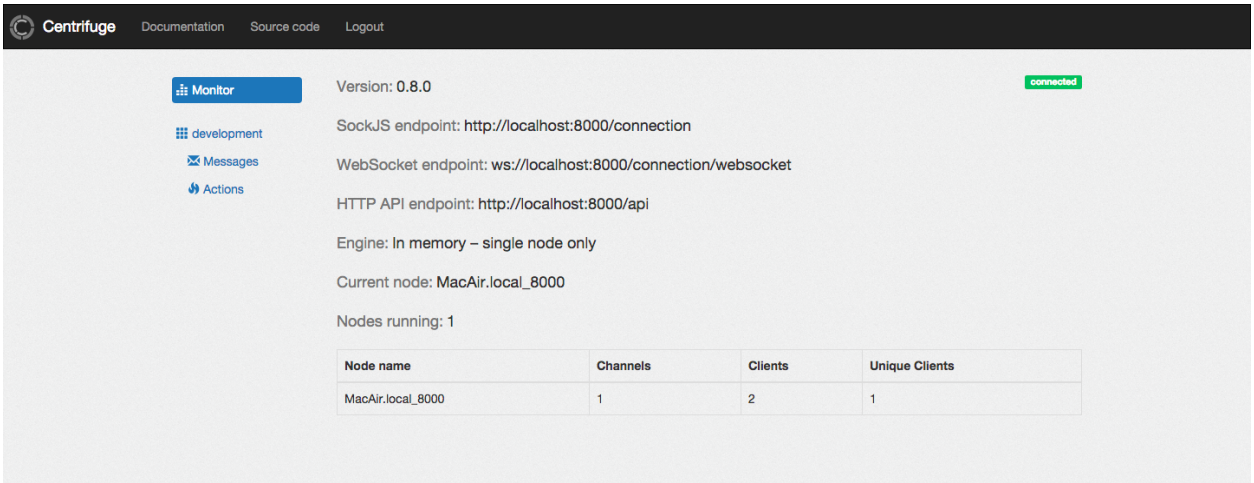
In a few words: clients (users of your web application/site) connect to Centrifuge from browser, after connecting clients subscribe on channels. Every message which was published into that channel will be delivered to all clients which are currently subscribed on that channel.

To connect to Centrifuge from browser pure Websockets or SockJS) library can be used. So it works in both modern and old browsers (IE 7 supported). Centrifuge has javascript client with simple API.

Centrifuge is built on top of Tornado - fast and mature asynchronous web server which can handle thousands of simultaneous connections.

Centrifuge scales using Redis as PUB/SUB broker and central state store. Single full-featured instance of Centrifuge run by default without extra dependency on Redis.

Centrifuge has administrative web interface to view configuration, monitor nodes state and watch messages in real-time.



Centrifuge is useful everywhere you need real-time web page updates.

There are tons of use cases - chats, graphs, comments, counters, games or when you just want to know how many users currently watching web page and who they are.

Centrifuge can be easily integrated with your existing web site - you don't need to change your project architecture and philosophy to get real-time events. Just install Centrifuge and let your users connect to it.

There are tons of examples in internet about how to add real-time events on your site. But very few of them provide complete, scalable, full-featured, ready to deploy solution. Centrifuge aims to be such a solution with simplicity in mind.

CHAPTER 2

Installation

Centrifuge was developed and tested on Linux and Mac OS X operating systems. The work on other systems is not guaranteed. On windows Centrifuge works with Memory Engine only.

It is written in Python. Python 2.6, 2.7, 3.3, 3.4 supported.

You can find nice guide about how to install Python on Mac OS X or Linux here and here respectively

To isolate Centrifuge environment it is recommended to use virtualenv. If you are not familiar with it yet - please, make time and read about it here

```
virtualenv --no-site-packages centrifuge/env
. centrifuge/env/bin/activate
```

On Python 3 make sure you have *pip* installed:

```
curl -O http://python-distribute.org/distribute_setup.py
python distribute_setup.py
easy_install pip
```

```
pip install centrifuge
```

Now you can run centrifuge:

```
centrifuge --config=config.json
```

We will talk about configuration file later in documentation.

Btw, you can speed up Centrifuge using 'ujson' module. As Centrifuge works a lot with JSON data - you can install *ujson* module to improve performance significantly. *pip install ujson* will do the work. This step is optional as Centrifuge uses built-in *json* module if no *ujson* available.

CHAPTER 3

Description

3.1 Overview

In this chapter I'll try to explain how Centrifuge actually works.

In a few words - clients from browsers connect to Centrifuge, after connecting clients subscribe on channels. And every message which was published into channel will be sent to all clients which are currently subscribed on this channel.

When you start Centrifuge instance you start Tornado on a certain port number. That port number can be configured using command-line option `--port` . By default `8000`. You can also specify the address to bind to with the `--address` option. For example you can specify `localhost` which is recommended if you want to keep Centrifuge behind a proxy (e.g.: Nginx). The port and the address will eventually be used by Tornado's TCPServer.

You should provide path to JSON configuration file when starting Centrifuge instance using `--config` option. Configuration file must contain valid JSON.

So the final command to start one instance of Centrifuge will be

```
centrifuge --config=config.json
```

You can scale and run more instances of Centrifuge on multiple machines using Redis engine. But for most cases one instance is more than enough.

Well, when you started one instance of Centrifuge - clients from web browsers can start connecting to it.

There are two endpoints for connections: - `/connection` for SockJS connections - `/connection/websocket` for pure Websocket connections

On browser side you now know the url to connect - for our simple case it is `http://localhost:8000/connection` in case of using SockJS library and `ws://localhost:8000/connection/websocket` in case of using pure Websockets.

To communicate with Centrifuge from browser you should use javascript client which comes with Centrifuge (find it in its own repository) and provides simple API. Please, read a chapter about client API to get more information.

Sometimes you need to run more instances of Centrifuge and load balance clients between them. As was mentioned above when you start default instance of Centrifuge - you start it with Memory Engine. In this case Centrifuge holds all state in memory. But to run several Centrifuge instances we must provide a way to share current state between instances. For this purpose Centrifuge utilizes Redis. To run Centrifuge with Redis you should run Centrifuge with Redis Engine instead of default Memory Engine.

First, install and run Redis (it's recommended to use Redis of version 2.6.9 or greater).

Now you can start several instances of Centrifuge. Let's start 2 instances.

Open terminal and run first instance:

```
CENTRIFUGE_ENGINE=redis centrifuge --config=config.json --port=8000
```

I.e. you tell Centrifuge to use Redis Engine providing environment variable `CENTRIGUGE_ENGINE` when launching it.

Explore available command line options specific for Redis engine using `--help`:

```
CENTRIFUGE_ENGINE=redis centrifuge --help
```

`CENTRIFUGE_ENGINE` can be `memory`, `redis` or path to custom engine class like `path.to.custom.Engine`

Then open another terminal window and run second instance on another port:

```
CENTRIFUGE_ENGINE=redis centrifuge --config=config.json --port=8001
```

Now two instances running and connected via Redis. Great!

But what is an url to connect from browser - `http://localhost:8000/connection` or `http://localhost:8001/connection`?

None of them, because Centrifuge must be kept behind proper load balancer such as Nginx. Nginx must be configured in a way to balance client connections from browser between our two instances. You can find Nginx configuration example in documentation or repo.

New client can connect to any of running instances. If client sends message we must send that message to other clients including those who connected to another instance at this moment. This is why we need Redis PUB/SUB here. All instances listen to special Redis channels and receive messages from those channels.

3.2 Projects

When you have Centrifuge instance and want to create web application using it - first you should do is to add your project into Centrifuge configuration file into **projects** array. **projects** is an array of projects in Centrifuge.

name - unique project name, must be written using ascii letters, numbers, underscores or hyphens.

secret - project secret key, used to sign API requests, create client tokens. Only Centrifuge and your web application backend must know the value of this secret. Make it unique and strong enough.

connection_lifetime - this is a time interval in seconds for connection to expire. Keep it as large as possible in your case. When clients connect to Centrifuge they provide timestamp - the UNIX time when their token was created. Every connection in project has connection lifetime (see below). This mechanism is disabled by default (connection_lifetime=0) and requires extra endpoint to be implemented in your application.

watch - publish messages into admin channel (messages will be visible in web interface). Turn it off if you expect high load in channels.

publish - allow clients to publish messages in channels (your web application never receive those messages)

anonymous - allow anonymous (with empty USER ID) clients to subscribe on channels

presence - enable/disable presence information

join_leave - enable/disable sending join(leave) messages when client subscribes on channel (unsubscribes from channel)

history_size - Centrifuge keeps all history in memory. In process memory in case of using Memory Engine and in Redis (which also in-memory store) in case of using Redis Engine. So it's very important to limit maximum amount of messages in channel history. This setting is exactly for this. By default history size is 0 - this means that channels will have no history messages at all.

history_lifetime - as all history is storing in memory it is also very important to get rid of old history data for unused (inactive for a long time) channels. This is interval in seconds to keep history for channel after last publishing into it. By default history lifetime is 0 - this means that channels will have no history messages at all. So to get history messages you should wisely configure both **history_size** and **history_lifetime** options.

3.3 Channels

The central part of Centrifuge is channels. Channel is a route for messages. Clients subscribe on channels, messages are being published into channels, channels everywhere.

Channel is just a string - `news`, `comments` are valid channel names.

BUT! You should remember several things.

First, channel name length is limited by 255 characters by default (can be changed via configuration file option `max_channel_length`)

Second, `:` and `#` and `$` symbols has a special role in channel names!

`:` - is a separator for namespace (see what is namespace below).

So if channel name is `public:chat` - then Centrifuge will search for namespace `public`.

`#` is a separator to create private channels for users without sending POST request to your web application. For example if channel is `news#user42` then only user with id `user42` can subscribe on this channel.

Moreover you can provide several user IDs in channel name separated by comma: `dialog#user42,user43` - in this case only `user42` and `user43` will be able to subscribe on this channel.

If channel starts with `$` (by default) then it's considered private. Read special chapter in docs about private channel subscriptions.

3.4 Namespaces

Centrifuge allows to configure channel's settings using namespaces.

You can create new namespace, configure its settings and after that every channel which belongs to this namespace will have these settings. It's flexible and provides a great control over channel behaviour. You can reduce the amount of messages travelling around dramatically by configuring namespace (for example disable join/leave) messages if you don't need them.

Namespace has several parameters - they are the same as project's settings. But with extra one:

name - unique namespace name: must consist of letters, numbers, underscores or hyphens

As was mentioned above if you want to attach channel to namespace - you must include namespace name into channel name with `:` as separator:

For example:

```
news:messages
```

```
gossips:messages
```

Where `news` and `gossips` are namespace names.

CHAPTER 4

Configuration

4.1 Configuration file

Example

Here is minimal configuration file required:

```
{
  "projects": [
    {
      "name": "development",
      "secret": "secret",
      "namespaces": []
    }
  ]
}
```

Description:

- **projects** - array of registered projects

There is also a possibility to override default SockJS-Tornado settings using Centrifuge configuration file. Example:

```
{
    ...,
    "sockjs_settings": {
        "sockjs_url": "https://cdn.jsdelivr.net/sockjs/1.0/sockjs.min.js"
    }
}
```

Here I set custom `sockjs_url` option, list of all available options can be found in sockjs-tornado source code: show on Github

Centrifuge runs a tornado HTTPServer under the hood. If you want to configure it you can do so via the `tornado_settings`. Please note that the `io_loop` argument is not supported for now. Example:

```
{
    ...,
    "tornado_settings": {
        "xheaders": true
    }
}
```

Centrifuge also allows to collect and export various metrics into Graphite. You can configure metric collecting and exporting behaviour using `metrics` object in configuration JSON.

In example below I enable logging metrics, and export into https://www.hostedgraphite.com/ service providing prefix, host and port to send metrics via UDP.

```
{
    ...,
    "metrics": {
        "log": true,
        "graphite": true,
        "graphite_host": "carbon.hostedgraphite.com",
        "graphite_port": 2003,
        "graphite_prefix": "MY_HOSTED_GRAPHITE_KEY.centrifuge",
        "interval": 30
    }
}
```

Metrics will be aggregated in a 30 seconds interval and then will be sent into log and into Graphite.

At moment Centrifuge collects for each node:

- broadcast - time in milliseconds spent to broadcast messages (average, min, max, count of broadcasts)
- connect - amount and rate of connect attempts to Centrifuge
- transport - counters for different transports (websocket, xhr_polling etc)
- messages - amount and rate of messages published
- channels - amount of active channels
- clients - amount of connected clients
- unique_clients - amount of unique clients connected
- api - count and rate of admin API calls

4.2 Command-line options

Centrifuge has several command line arguments.

`--config` - path to configuration json file, by default `config.json`

`--debug` - run Centrifuge in Tornado debug mode - server will be reloaded when code changes.

`--port` - port to bind (default `8000`)

`--address` - address to bind to

`--name` - unique node name (optional) - will be used in web interface metric table or in graphite data path

`--web` - optional path to serve Centrifuge web interface single-page application

Some other command line options come with engine - explore them using `--help`, for example:

```
CENTRIFUGE_ENGINE=redis centrifuge --help
```

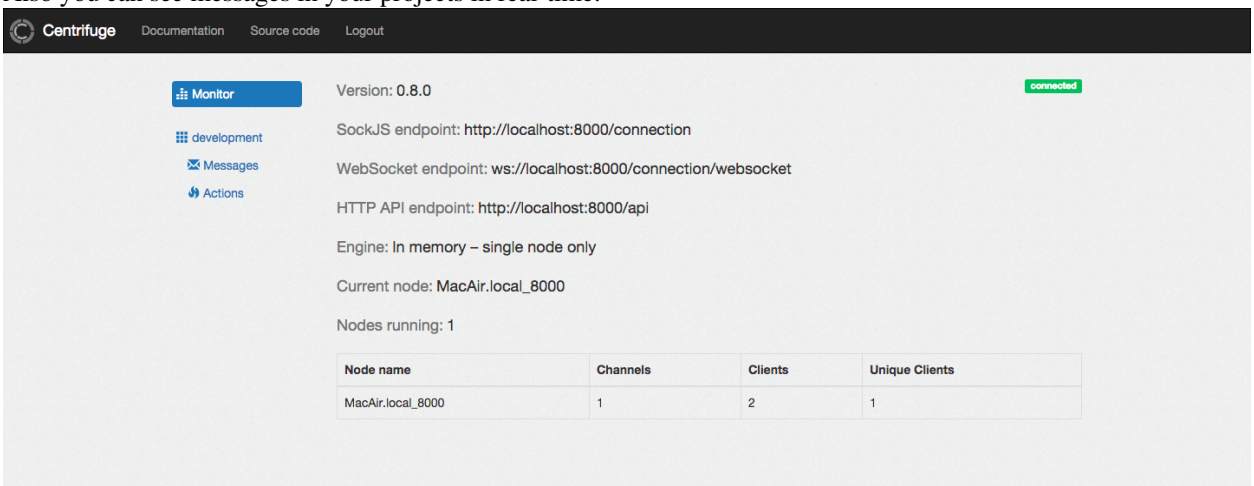
CHAPTER 5

Web interface

Centrifuge has with administrative web interface. It's a ReactJS based single-page application. You can find it in its own repo.

It allows to view projects, namespaces, publish JSON messages into channels etc.

Also you can see messages in your projects in real-time.



You can log in using **password** from configuration file.

Remember that in production you **must** always use strong password.

CHAPTER 6

Server API

6.1 Overview

Look at project/namespace options. There is an option called `publish`. When checked this option allows browser clients to publish into channels of namespace directly. If client publishes a message into channel directly - your application will not receive that message (it just goes through Centrifuge towards subscribed clients). This pattern can be useful sometimes but in most cases you need to receive event from client, process it - validate and save into database and then broadcast to other connected clients. In this case you must not use channels which namespace allows publishing.

The common pattern in this situation is receive new message via AJAX, do whatever you need with it and then publish into Centrifuge using HTTP API. If your backend written on Python you can use Cent API client. If you use other language don't worry - I will describe how to communicate with Centrifuge API endpoint right now.

Centrifuge API url endpoint is `/api/PROJECT_ID`, where PROJECT_ID must be replaced with your project ID (you can find it in Centrifuge's web interface).

So if your Centrifuge domain is `https://centrifuge.example.com` and project ID is `c54e65c4v6565` then an API address will be `https://centrifuge.example.com/api/c54e65c4v6565`.

All you need to do to use HTTP API is to send correctly constructed POST request to this endpoint.

API request must have two POST parameters: `data` and `sign`.

`data` is a json string representing command you want to send to Centrifuge (see below) and `sign` is an HMAC based on project secret key, project ID and `data` json string. This `sign` is used to validate request.

`data` is a json string made from object with two properties: `method` and `params`.

`method` is a name of action you want to do. `params` is an object with method arguments.

First lets see how to construct such request in Python. If Python is your language then you don't have to implement this yourself as `Cent` module exists. But this example here can be useful for someone who want to implement interaction with Centrifuge API in language for which we don't have API client yet.

Lets imagine that you have a project with name `development` and secret key `uutryx`. In web interface of Centrifuge you can find HTTP API url, if you run Centrifuge locally then it will be something like `http://localhost:8000/api`

So you should send POST request to `http://localhost:8000/api/development`.

```
from urllib2 import urlopen, Request
from cent.core import generate_api_sign
import json
```

```
req = Request("http://localhost:8000/api/development")

commands = [
    {
        "method": "publish",
        "params": {"channel": "docs", "data": {"json": True}}
    }
]
encoded_data = json.dumps(commands)
sign = generate_api_sign("uutryx", "development", encoded_data)

data = urlencode({'sign': sign, 'data': encoded_data})
response = urlopen(req, data, timeout=5)
```

See how to generate HMAC api sign in special chapter "Tokens and signatures"

There are lots of commands you can call. Some for managing project structure, some for managing channels. Lets take a closer look on them.

6.2 Methods for managing channels

Those are **publish**, **unsubscribe**, **presence**, **history**, **disconnect**

Lets just go through each of methods and look what they do and which params you need to provide.

publish - send message into channel of namespace. `data` is an actual information you want to send into channel. **It must be valid JSON**.

```
{
    "method": "publish",
    "params": {
        "channel": "CHANNEL NAME",
        "data": {}
    }
}
```

unsubscribe - unsubscribe user with certain ID from channel.

```
{
    "method": "unsubscribe",
    "params": {
        "channel": "CHANNEL NAME",
        "user": "USER ID"
    }
}
```

disconnect - disconnect user by user ID.

```
{
    "method": "disconnect",
    "params": {
        "user": "USER ID"
    }
}
```

presence - get channel presence information (all clients currently subscribed on this channel).

```
{
    "method": "presence",
    "params": {
        "channel": "CHANNEL NAME"
    }
}
```

history - get channel history information (list of last messages sent into channel).

```
{
    "method": "history",
    "params": {
        "channel": "CHANNEL NAME"
    }
}
```

6.3 Cent

Cent is a way to communicate with Centrifuge from python code or from console (terminal).

To install:

```
pip install cent
```

By default Cent uses *.centrc* configuration file from your home directory (`~/.centrc`).

Here is an example of config file's content:

```
[football]
address = http://localhost:8000/api
key = PROJECT_KEY
secret = 994021f2dc354d7893d88b90d430498e
timeout = 5
```

The most obvious case of using Cent is broadcasting events into channels.

It is easy enough:

```
cent football publish --params='{"channel": "news", "data": {"title": "World Cup 2018", "text": "some
```

- **cent** is the name of program
- **football** is the name of section in configuration file
- **publish** is the method name you want to call
- **–params** is a JSON string with method parameters, in case of publish you should provide channel and data parameters.

If request was successful you'll get something like this in response:

```
{'error': None, 'body': True, 'method': 'publish'}
```

In case of any error you will get its description.

Cent contains Client class to send messages to Centrifuge from your python-powered backend:

```
from cent.core import Client

client = Client("http://localhost:8000/api", "project_key", "project_secret")
```

```
client.add(
    "publish",
    {
        "channel": "python",
        "data": "hello world"
    }
)
result, error = client.send()
```

you can use `add` method to add several messages which will be sent. But up to 100 (default, can be configured via Centrifuge configuration file using `admin_api_message_limit` option)

6.4 Python

If your backend Python powered and you don't want to install Cent, you can just copy `Client` class from Cent source code (`cent.core.Client`) and use it as was shown above.

6.5 Ruby

Oleg Bovykin contributed an implementation of Centrifuge API client for Ruby.

6.6 Java

There is an implementation of Centrifuge API client written by Markus Coetzee. The source code is available here

6.7 PHP

There is an implementation of Centrifuge API client written by Dmitriy Soldatenko. The source code is available here

CHAPTER 7

Javascript client

7.1 Simple javascript browser client

Javascript client is very simple. Lets take a look at its API step by step.

When you have Centrifuge instance running - it is time to communicate with it from browser.

First, import javascript client into your web page, the simplest way is just include `script` tag:

```
<script src="https://rawgit.com/centrifugal/centrifuge-js/master/centrifuge.js"></script>
```

Javascript client built on top of Event Emitter written by Oliver Caldwell.

First, create new centrifuge object instance:

```
var centrifuge = new Centrifuge({
    url: 'http://centrifuge.example.com/connection',
    project: 'PROJECT KEY',
    user: 'USER ID',
    timestamp: 'timestamp',
    token: 'token'
});
```

url is a Centrifuge endpoint - SockJS or Websocket. Note that in case of using SockJS it must be imported on your page before Centrifuge's javascript client:

```
<script src="https://cdn.jsdelivr.net/sockjs/1.0/sockjs.min.js" type="text/javascript"></script>
<script src="https://rawgit.com/centrifugal/centrifuge-js/master/centrifuge.js" type="text/javascript
```

In case of using SockJS additional configuration parameter can be used - *transports*. It defines allowed transports and by default equals ['websocket', 'xdr-streaming', 'xhr-streaming', 'eventsource', 'iframe-eventsource', 'iframe-htmlfile', 'xdr-polling', 'xhr-polling', 'iframe-xhr-polling', 'jsonp-polling'].

project is project key from Centrifuge. Every project in Centrifuge has unique name (key). You can see it on project settings page of administrative web interface. This is just a string.

user is your web application's current user unique ID. It must be string and can be empty if you want to allow anonymous access to your real-time application.

timestamp is current UNIX seconds as string. For example for Python `str(int(time.time()))`

token is a secret key generated by your web application based on project secret key, project key and user ID. Every project in Centrifuge has secret key. You must never show it to anyone else. The only two who know secret key are Centrifuge itself and your web application backend. To create token you must use HMAC algorithm. To understand how to generate client connection token see special chapter Tokens and signatures.

Correct token guarantees that connection request to Centrifuge contains valid information about project key, user ID and timestamp. Token is similar to HTTP cookie, client must not show it to anyone else. Remember that you must always use channels in private namespaces when working with limited access data.

Also you can optionally provide extra parameter `info` when connecting to Centrifuge:

```
var centrifuge = new Centrifuge({
    url: 'http://centrifuge.example.com/connection',
    project: 'PROJECT ID',
    user: 'USER ID',
    timestamp: 'timestamp',
    info: '{"first_name": "Alexandr", "last_name": "Emelin"}',
    token: 'token'
});
```

`info` is an additional information about user connecting to Centrifuge. It must be valid JSON string. But to prevent client sending wrong `info` this JSON string must be used while generating token.

If you don't want to use `info` - you can omit this parameter while connecting to Centrifuge. But if you omit it then make sure that it have not been used in token generation.

If you are using Python - then you can use *generate_token* function from *cent* library to generate tokens for your users.

So, now centrifuge client configured and you are ready to start communicating.

First, we should actually connect to Centrifuge:

```
centrifuge.connect();
```

This line makes actual connection request to Centrifuge with data you provided as initialization step.

After successful connect you can subscribe on channels. But you can only start subscribing when connection with Centrifuge was successfully established. If you try to subscribe on channel before connection established - your subscription request will be rejected by Centrifuge. There is an event about successful connection and you can bind your subscription logic to it in this way:

```
centrifuge.on('connect', function() {
    // now your client connected
});
```

Also you `disconnect` and `error` events available:

```
centrifuge.on('disconnect', function() {
    // do whatever you need in case of disconnect
});

centrifuge.on('error', function(error_message) {
    // called every time error occurred
});
```

When your client connected, it is time to subscribe on channel:

```
var subscription = centrifuge.subscribe('namespace:channel', function(message) {
    // called when message received from this channel
});
```

If namespace of channel has *publish* option enabled you can publish messages into this channel. But you can not do it immediately after subscription request. You can only publish when `subscribe:success` event will be fired. The same in case of presence and history requests. Lets publish message, get presence and get history data as soon as our subscription request returned successful subscription response:

```
subscription.on('ready', function() {

    // publish into channel
    subscription.publish("hello");

    // get presence information (who is currently subscribed on this channel)
    subscription.presence(function(message) {
        console.log(message);
    });

    // get history (last messages sent) for this channel
    subscription.history (function(message) {
        console.log(message);
    });

    subscription.on('join', function(message) {
        // called when someone subscribes on channel
    });

    subscription.on('leave', function(message) {
        // called when someone unsubscribes from channel
    });

});
```

You can unsubscribe from subscription:

```
subscription.unsubscribe();
```

In some cases you need to disconnect your client from Centrifuge:

```
centrifuge.disconnect();
```

After calling this client will not try to reestablish connection periodically. You must call `connect` method manually.

Starting from Centrifuge 0.5.0 there is an experimental message batching support. It allows to send several messages to Centrifuge in one request - this can be especially usefull when connection established via one of non-streaming HTTP polyfills.

You can start collecting messages to send calling `startBatching()` method:

```
centrifuge.startBatching();
```

When you want to actually send all collected messages to Centrifuge call `flush()` method:

```
centrifuge.flush();
```

Maximum amount of messages in one batching request is 100 (this is by default and can be changed in Centrifuge configuration file using `client_api_message_limit` option).

Finally if you don't want batching anymore call `stopBatching()` method:

```
centrifuge.stopBatching();
```

call `stopBatching(true)` to flush all messages and stop batching.

Version 0.7.0 introduced new **'pusher<https://pusher.com/docs/client_api_guide/client_private_channels>'_**-like private channel subscription mechanism. Now if channel name starts with `$` (by default) then subscription on this channel will be checked via AJAX POST request from javascript to your web application.

You subscribe on private channel as usual:

```
centrifuge.subscribe('$private', function(message) {
    // process message
});
```

But in this case client will first check subscription via your backend sending POST request to `/centrifuge/auth` endpoint (by default). This request will contain `client` parameter which is your connection client ID and `channels` parameter - one or multiple private channels client wants to subscribe to. Your server should validate all this subscriptions and return properly signed responses.

There are also two new public API methods in 0.7.0 which can help to subscribe to many private channels sending only one POST request to your web application backend: `startAuthBatching` and `stopAuthBatching`. When you `startAuthBatching` centrifuge js client will collect private subscriptions until `stopAuthBatching` call - and then send them all at once.

Read more about private channels in special documentation chapter.

7.2 Plugins

centrifuge.dom.js - jQuery plugin to use DOM elements to manipulate non dynamic subscriptions.

CHAPTER 8

Private channels

So Centrifuge 0.7.0 introduced new private channel subscription mechanism. All channels starting with `$` (by default) considered private. In chapter about javascript client you have seen that subscribing on private channel from javascript client does not differ from subscribing on usual channels. But you should implement an endpoint in your web application that will check if current users can subscribe on certain private channels.

By default javascript client will sen AJAX POST request to `/centrifuge/auth` url. You can change this address and add additional request headers via js client initialization options (`authEndpoint` and `authHeaders`).

POST request includes two parameters: `client` and `channels`. Client is a string with current client ID and channels is one or more channels current user wants to subscribe to.

I think it's simplier to explain on example.

Lets imagine that client wants to subscribe on two private channels: `$one` and `$two`.

Here is a javascript code to subscribe on them:

```
centrifuge.subscribe('$one', function(message) {
    // process message
});

centrifuge.subscribe('$two', function(message) {
    // process message
});
```

In this case Centrifuge will send two separate POST requests to your web app. There is an option to batch this requests into one using `startAuthBatching` and `stopAuthBatching` methods. Like this:

```
centrifuge.startAuthBatching();

centrifuge.subscribe('$one', function(message) {
    // process message
});

centrifuge.subscribe('$two', function(message) {
    // process message
});

centrifuge.stopAuthBatching();
```

In this case one POST request with 2 channels in `channels` parameter will be send. Let's look at simplified example for Tornado how to implement auth endpoint:

```
from cent.core import generate_channel_sign

```

```
class CentrifugeAuthHandler(tornado.web.RequestHandler):

    def check_xsrf_cookie(self):
        pass

    def post(self):

        client_id = self.get_argument("client")
        channels = self.get_arguments("channels")
        logging.info("{0} wants to subscribe on {1}".format(client_id, ", ".join(channels)))

        to_return = {}

        for channel in channels:
            info = json.dumps({
                'channel_extra_info_example': 'you can add additional JSON data when authorizing'
            })
            sign = generate_channel_sign(
                options.secret_key, client_id, channel, info=info
            )
            to_return[channel] = {
                "sign": sign,
                "info": info
            }

        # but here we allow to join any private channel and return additional
        # JSON info specific for channel
        self.set_header('Content-Type', 'application/json; charset="utf-8"')
        self.write(json.dumps(to_return))
```

In this example we allow user to subscribe on any private channel. If you want to reject subscription - then you can add "status" key and set it to something not equal to 200, for example 403:

```
from cent.core import generate_channel_sign

class CentrifugeAuthHandler(tornado.web.RequestHandler):

    def check_xsrf_cookie(self):
        pass

    def post(self):

        client_id = self.get_argument("client")
        channels = self.get_arguments("channels")
        logging.info("{0} wants to subscribe on {1}".format(client_id, ", ".join(channels)))

        to_return = {}

        for channel in channels:
            sign = generate_channel_sign(
                options.secret_key, client_id, channel
            )
            to_return[channel] = {
                "status": 403,
            }

        # but here we allow to join any private channel and return additional
        # JSON info specific for channel
```

```
        self.set_header('Content-Type', 'application/json; charset="utf-8"')
        self.write(json.dumps(to_return))
```

If user deactivated in your application then you can just return 403 Forbidden response:

```
from cent.core import generate_channel_sign

class CentrifugeAuthHandler(tornado.web.RequestHandler):

    def check_xsrf_cookie(self):
        pass

    def post(self):
        raise tornado.web.HTTPError(403)
```

This will prevent client from subscribing to any private channel.

CHAPTER 9

Tokens and signatures

Centrifuge uses HMAC algorithm to create connection token and to sign various data when communicationg with Centrifuge via server or client API.

In this chapter we will see how to create tokens and signatures for different actions. If you use Python all functions available in `Cent` library and you don't need to implement them. This chapter can be useful for developers building their own library (in other language for example) to communicate with Centrifuge.

Lets start with connection token

9.1 Connection token

When you connect to Centrifuge from browser you should provide several connection parameters: "project", "user", "timestamp", "info" and "token".

We discussed the meaning of parameters in previous chapters - here we will see how to generate a proper token.

Let's look at Python code for this:

```
import six
import hmac
from hashlib import sha256

def generate_token(secret_key, project_key, user_id, timestamp, info=None):
    if info is None:
        info = json.dumps({})
    sign = hmac.new(six.b(str(secret_key)), digestmod=sha256)
    sign.update(six.b(project_key))
    sign.update(six.b(user_id))
    sign.update(six.b(timestamp))
    sign.update(six.b(info))
    token = sign.hexdigest()
    return token
```

We initialize HMAC with project secret key and `sha256` digest mode and then update it with project_key, user_id, timestamp and info. Info is an optional arguments and if no info provided it defaults to empty object.

9.2 Private channel subscription sign

When client wants to subscribe on private channel Centrifuge sends AJAX POST request to your web application. This request contains client ID string and one or multiple private channels. In response you should return an object

where channels are keys.

For example you received request with channels "$one" and "$two". Then you should return JSON with something like this in response:

```
{
    "$one": {
        "info": "{}",
        "sign": "sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss"
    },
    "$two": {
        "info": "{}",
        "sign": "sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss"
    }
}
```

Where `info` is additional information about connection for this channel and `sign` is properly constructed HMAC based on client ID, channel name and info. Lets look at Python code generating this sign:

```
import six
import hmac
from hashlib import sha256

def generate_channel_sign(secret_key, client_id, channel, info=None):
    if info is None:
        info = json.dumps({})
    auth = hmac.new(six.b(str(secret_key)), digestmod=sha256)
    auth.update(six.b(str(client_id)))
    auth.update(six.b(str(channel)))
    auth.update(six.b(info))
    return auth.hexdigest()
```

Not so different from generating token. Note that as with token - info is already JSON encoded string.

9.3 API request sign

When you use Centrifuge server API you should also provide sign in each request.

Again, Python code for this:

```
import six
import hmac
from hashlib import sha256

def generate_api_sign(self, secret_key, project_key, encoded_data):
    sign = hmac.new(six.b(str(secret_key)), digestmod=sha256)
    sign.update(six.b(project_key))
    sign.update(encoded_data)
    return sign.hexdigest()
```

`encoded_data` is already a JSON string with your API commands. See available commands in server API chapter.

CHAPTER 10

Engines

Engine in Centrifuge is a class responsible for managing subscriptions, publishing messages into appropriate channels, handling published messages, handling presence and history information.

Centrifuge has 2 built-in engines - in Memory engine and Redis engine. By default Memory engine is used.

To set engine you should use `CENTRIFUGE_ENGINE` environment variable.

Available values are `memory`, `redis` or python path to custom engine:

Memory engine:

```
CENTRIFUGE_ENGINE=memory centrifuge --config=config.json
```

Redis engine:

```
CENTRIFUGE_ENGINE=redis centrifuge --config=config.json
```

Redis engine using path to class:

```
CENTRIFUGE_ENGINE="centrifuge.engine.redis.Engine" centrifuge --config=config.json
```

10.1 Memory engine

Supports only one node. Nice choice to start with. Supports all features keeping everything in process memory.

10.2 Redis engine

Allows scaling Centrifuge running multiple processes on same or different machine. Keeps presence and history data in Redis, uses redis PUB/SUB to support running multiple instances of Centrifuge. Also it allows to call API commands.

See available redis engine specific options using `--help`:

```
CENTRIFUGE_ENGINE=redis centrifuge --help
```

How to publish via Redis engine API listener? Start Centrifuge with Redis engine and `--redis_api` option:

```
CENTRIFUGE_ENGINE=redis centrifuge --logging=debug --config=config.json --redis_api
```

Then use Redis client for your favorite language, ex. for Python:

```
import redis
import json

client = redis.Redis()

to_send = {
    "project": "development",
    "data": [
        {
            "method": "publish",
            "params": {"channel": "$public:chat", "data": {"input": "hello"}}
        },
        {
            "method": "publish",
            "params": {"channel": "events", "data": {"event": "message"}}
        },
    ]
}

client.rpush("centrifuge.api", json.dumps(to_send))
```

So you send JSON object with project name as a value for `project` key and list of commands as a value for `data` key.

Note again - you don't have response here. If you need to check response - you should use HTTP API.

`publish` is the most usable command in Centrifuge so Redis API listener was invented with primary goal to reduce HTTP overhead when publishing quickly. This can also help using Centrifuge with other languages for which we don't have HTTP API client yet.

CHAPTER 11

Insecure mode

Version 0.7.0 of Centrifuge introduced new insecure mode.

To start Centrifuge in this mode use `CENTRIFUGE_INSECURE` environment variable:

```
CENTRIFUGE_INSECURE=1 centrifuge --logging=debug --debug --config=config.json
```

You can also set `insecure` option to `true` in configuration file to do the same.

Insecure mode:

- disables client timestamp and token check
- allows anonymous access for all channels
- allows client to publish into all channels
- suppresses connection check

When using insecure mode you can create client connection in this way:

```
var centrifuge = new Centrifuge({
    "url": url,
    "project": project,
    "insecure": true
});
```

Note that there is no `token`, `user` and `timestamp` parameters so you can connect to Centrifuge without any backend code.

This allows to use Centrifuge as a quick and simple solution when making real-time demos, presentations, testing ideas etc. But this is only for personal demonstration use cases - this mode should never work in production until you really want it to be there.

Look at demo to see insecure mode in action.

CHAPTER 12

Connection check

This mechanism was introduced in Centrifuge 0.7.0

Before 0.7.0 Centrifuge had another connection check mechanism which was unreasonably difficult.

When client connects to Centrifuge with proper connection credentials his connection can live forever. This means that even if you banned this user in your web application he will be able to read messages from channels he already subscribed to. It's not desirable in some cases.

Project has special option: `connection_lifetime`. Connection lifetime is 0 by default and this value means that connection check mechanism is off.

When connection lifetime is set to value greater than 0 then this is a time in seconds how long connection will be valid after successful connect. When connection lifetime expires Centrifuge will send a signal to javascript client and it will make an AJAX POST request to your web application. By default this request goes to `/centrifuge/refresh` url endpoint. You can change it using javascript option `refreshEndpoint`. In response your server must return JSON with connection credentials:

```
to_return = {
    'project': "PROJECT ID",
    'user': "USER ID,
    'timestamp': "CURRENT TIMESTAMP AS INTEGER",
    'info': "ADDITIONAL CONNECTION INFO",
    'token': "TOKEN BASED ON PARAMS ABOVE",
}
return json.dumps(to_return)
```

You should just return the same connection credentials when rendering page initially. Just with current timestamp. Centrifuge javascript client will then send them to Centrifuge and connection will be refreshed for a connection lifetime period.

If you don't want to refresh connection for this user - just return 403 Forbidden on refresh request to your web application backend.

CHAPTER 13

Deployment

13.1 Nginx configuration

Minimal Nginx version - 1.3.13

Here is an example Nginx configuration to deploy Centrifuge.

```
#user nginx;
worker_processes 4;

#error_log /var/log/nginx/error.log;

events {
    worker_connections 1024;
    #use epoll;
}

http {
    # Enumerate all the Tornado servers here
    upstream centrifuge {
        #sticky;
        ip_hash;
        server 127.0.0.1:8000;
        #server 127.0.0.1:8001;
        #server 127.0.0.1:8002;
        #server 127.0.0.1:8003;
    }
    include mime.types;
    default_type application/octet-stream;

    keepalive_timeout 65;
    proxy_read_timeout 200;
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    gzip on;
    gzip_min_length 1000;
    gzip_proxied any;

    # Only retry if there was a communication error, not a timeout
    # on the Tornado server (to avoid propagating "queries of death"
    # to all frontends)
    proxy_next_upstream error;
```

```
    map $http_upgrade $connection_upgrade {
        default upgrade;
        ''      close;
    }

    server {
        listen 8081;
        server_name localhost;

        location / {
            proxy_pass_header Server;
            proxy_set_header Host $http_host;
            proxy_redirect off;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Scheme $scheme;
            proxy_pass http://centrifuge;
        }
        location /socket {
            proxy_buffering off;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Scheme $scheme;
            proxy_pass http://centrifuge;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
        }
        location /connection {
            proxy_buffering off;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Scheme $scheme;
            proxy_pass http://centrifuge;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
        }
    }
}
```

Look carefully at commented `sticky;` directive in upstream section.

In this configuration example we use `ip_hash` directive to proxy client with the same ip address to the same backend process. This is very important when we have several processes.

When client connects to Centrifuge - session created - and to communicate those client must send all next requests to the same backend process. But `ip_hash` is not the best choice in this case, because there could be situations where a lot of different browsers are coming with the same IP address (behind proxies) and the load balancing system won't be fair. Also fair load balancing does not work during development - when all clients connecting from localhost.

So best solution would be using something like nginx-sticky-module which uses a cookie to track the upstream server for making each client unique.

Also you will probably want API calls spread over one or several Centrifuge instances, you can just create new custom upstream and use it for `/api` location:

```
...

upstream centrifuge-api {
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
```

```
    server 127.0.0.1:8002;
}

server {

    ...

    location /api {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
        proxy_pass http://centrifuge-api;
    }

    ...

}

...
```

13.2 Supervisord configuration example

In 'deploy' folder of Centrifuge's repo you can find supervisord configuration example. Something like this:

centrifuge.conf (put it into `/etc/supervisor/conf.d/centrifuge.conf`)

```
[program:centrifuge]
process_name = %(process_num)s
environment=PYTHONPATH="/opt/centrifuge/src"
directory = /opt/centrifuge/src
command = /opt/centrifuge/env/bin/python /opt/centrifuge/src/centrifuge/node.py --log_file_prefix=/va
numprocs = 1
numprocs_start = 8000
user = centrifuge
```

13.3 Centos 6

In repository you can find everything to build rpm for Centos 6.

If you are not a Centos 6 user you can find a lot of useful things there, which were successfully tested in production environment.

CHAPTER 14

License

Centrifuge has MIT license.

Code is here - https://github.com/centrifugal/centrifuge