IRCAnywhere Documentation

Release alpha

Ricki Hastings

January 21, 2017

Contents

1 Pre Requirements **3**
- 1.1 Installing Node.js and Npm . . . 3
- 1.2 Installing With The Installer . . . 4
- 1.3 Installing MongoDB . . . 4

2 Installing IRCAnywhere **7**
- 2.1 Automatic Install . . . 7
- 2.2 Installing . . . 7
- 2.3 HTTPS . . . 8
- 2.4 Running . . . 8
- 2.5 Updating . . . 8

3 Reverse Proxies **9**
- 3.1 Apache . . . 9
- 3.2 Nginx . . . 9

4 Using the module system **11**
- 4.1 Server Side . . . 11
- 4.2 Client Side . . . 12

5 Server API **13**
- 5.1 Application . . . 13
- 5.2 ChannelManager . . . 14
- 5.3 CommandManager . . . 16
- 5.4 EventManager . . . 22
- 5.5 IRCFactory . . . 24
- 5.6 IdentdServer . . . 25
- 5.7 IRCHandler . . . 26
- 5.8 IRCServer . . . 31
- 5.9 ModeParser . . . 32
- 5.10 Module . . . 33
- 5.11 ModuleManager . . . 33
- 5.12 NetworkManager . . . 34
- 5.13 RPCHandler . . . 36
- 5.14 ServerSession . . . 39
- 5.15 UserManager . . . 41
- 5.16 WebSocket . . . 44

IRCAnywhere is an application written in javascript which is designed to be a free alternative to IRCCloud. With IRCCloud, you have little control over your uptime and the privacy of your users. **IRCAnywhere** aims to be a replacement giving the control to you.

IRCAnywhere has been around for a while, and was first opened to the public as a proprietary service back in 2012. It got open sourced in May, 2013 and we quickly realised that it wasn't as simple and stable as it should be. Recently **IRCAnywhere** has undergone a massive rewrite completely from the ground up with some fundamental changes to the way it previously worked.

This documentation is for the most current versions `0.2-alpha` and `0.2-beta`. The code is fully open source and available on github. The documentation is currently lacking but expect the project to be fully documented in more stable releases, currently the server side API is documented to an extent which will be improved as the project matures.

Contents:

CHAPTER 1

Pre Requirements

IRCAnywhere requires a few tools to be installed before we can start installing the package. The following tools are required to proceed *if you have these installed you can skip to the next page*;

- nodejs
- npm
- MongoDB

1.1 Installing Node.js and Npm

First we'll install nodejs and npm *we recommend latest stable versions as always*.

- **Fedora**

  ```
  $ sudo yum install nodejs npm -y
  ```

- **Mac OSX**

  ```
  $ brew install node npm
  ```

- **Debian/Ubuntu**

  Latest versions of nodejs include npm aswell.

```
$ sudo add-apt-repository ppa:chris-lea/node.js
$ apt-get update
$ apt-get install nodejs
```

If this is not working for any reason, usually something similar to:

```
Err http://ppa.launchpad.net wheezy/main Sources
404  Not Found
```

Then I would recommend installing node with nvm

```
$ curl https://raw.github.com/creationix/nvm/v0.3.0/install.sh | sh
$ nvm install 0.10
$ nvm use 0.10
$ nvm alias default 0.10
```

1.2 Installing With The Installer

In the latest development branch we now have an install script to automate everything past this point. If you're running in a production environment it's recommended to install your own global version of MongoDB so if the time comes you're ready to scale it and secure it. Although if you're just interested in a quick get up and go then it's worth skipping to this section.

1.3 Installing MongoDB

Next we'll install MongoDB and set it up correctly.

- **Fedora**

  ```
  $ yum install mongo-10gen mongo-10gen-server
  ```

  You may need to create a */etc/yum.repos.d/mongodb.repo* file and add the following to it.

  64-bit operating system.

```
[mongodb]
name=MongoDB Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64/
gpgcheck=0
enabled=1
```

  32-bit operating system (not recommended for production).

```
[mongodb]
name=MongoDB Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/i686/
gpgcheck=0
enabled=1
```

- **Mac OSX**

```
$ brew update
$ brew install mongodb
```

- **Debian/Ubuntu**

```
$sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
$echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee /etc/a
$sudo apt-get update
$sudo apt-get install mongodb-org
```

The next step is setting up MongoDB correctly so we can take advantage of the oplog tailing features, to do this we need to start MongoDB with a replica set. It's likely your package manager started MongoDB when they finished installing it, so we need to shut it down first, do this with the following commands.

```
$ mongo
> use admin;
> db.shutdownServer();
```

or

```
$ killall -12 mongod
```

We now need to start mongodb with a single replica set for oplog tailing. Although a single mongo server wouldn't need to be a replica set usually, they allow it for testing purposes, if you're planning on running ircanywhere in a

production environment with a good number of users I would recommend setting up an actual cluster of servers here (you'll hear more about clustering ircanywhere processes together soon).

If you are running MongoDB from a config file, which is usually located at `/etc/mongodb.conf`. Then you can edit this file and include the following line at the bottom:

```
replSet = rs0
fork = true
```

You can then restart MongoDB using the config file with the following commands:

```
$ mongod --config /etc/mongodb.conf
$ mongo
> rs.initiate()
```

If the file doesn't exist you can start MongoDB with the following options to initiate a replica set (although I would recommend having a config file to save you passing in these options every time you reboot. Although this is getting out of the scope of this guide). You may need to run it as sudo.

```
$ mongod --logpath /var/log/mongodb.log --replSet rs0
$ mongo
```

Once you've started the mongo instance sucessfully you can connect to it with the *mongo* command, once connected you should see this:

```
MongoDB shell version: 2.4.9
connecting to: test
rs0:PRIMARY>
```

If you see the *:PRIMARY>* suffix then you've set the replica set up successfully. If you're still having trouble you can try following this more detailed guide at http://meteorhacks.com/lets-scale-meteor.html.

CHAPTER 2

Installing IRCAnywhere

2.1 Automatic Install

You can choose to do a manual install if you want to understand how things work, by following the rest of this document, or you can run the install script which will automatically install MongoDB, the npm dependencies, start MongoDB correctly and compile the client side files.

To do this simply run the command

```
$ ./install.sh
```

If you already have MongoDB installed but not set up correctly with Oplog tailing, we can do that for you aswell, simply run

```
$ sudo ./install.sh
```

If this goes through with no errors you can skip to the running section *here*.

2.2 Installing

Once the environment is setup properly you can proceed with installing **IRCAnywhere**.

The first step is to clone the github repo, or you can install from the `0.2-alpha` release. However a number of stability changes have been made since then so the `development` branch is usually the most stable.

```
$ git clone https://github.com/ircanywhere/ircanywhere.git
$ cd ircanywhere
$ git checkout development
```

or from *0.2-alpha*

```
$ wget https://github.com/ircanywhere/ircanywhere/archive/v0.2-alpha.tar.gz
$ tar xvf v0.2-alpha.tar.gz
$ cd ircanywhere-0.2-alpha
```

Then we need to install the dependencies (I've no idea how this runs on windows, I'm not expecting it to run well because we use fibers. I'd recommend unix/linux based operating systems).

```
$ npm install
```

Next you'll need to build the client source, you'll need to make sure `gulp` is installed via npm. Once that is done you can run these commands. You can set gulp up to watch files if you're doing any development work (including writing plugins) by running *gulp watch* after the following commands.

```
$ npm install -g gulp
$ gulp
```

Finally, edit the configuration file `config.example.json` a few things will need changed by default, the ip address and port, and you'll need to include a smtp url if you want to be able to send emails out (forgot password links wont work without emails). Your MongoDB settings should be fine if you've followed these instructions or automatically installed it with the installer. Finally rename it to `config.json`.

2.3 HTTPS

IRCAnywhere can also be served via HTTPS. Setting it up involves little more than editing the configuration and setting the `ssl` property to `true`. Once this is done you will need to add the following files into `private/certs`

- `private/certs/key.pem`
- `private/certs/cert.pem`

2.4 Running

There are multiple ways you can run **IRCAnywhere**, you probably want to run it detaching from the console so it runs as a daemon, you can do that with the following commands:

```
$ npm start
```

or

```
$ node .  start
```

Note that the above commands wont restart it's self when an exception occurs. To do this you're going to want to respond to signals to reboot if the system crashes or gets killed for some other reason. Traditionally node applications are ran with `forever`, however there is a strange case causing `irc-factory` to reboot when the parent restarts which loses our ability to detach from IRC connections keeping them online between restarts, this is not good.

I use a program called https://github.com/visionmedia/mon to keep the process running. You should use `node . run` and not `node . start` when using `mon` because it will go into a restart loop if you don't.

```
$ mon -d "node .  run" -p ircanywhere.pid -l logs/mon.log
```

If you're running in a production environment it would be better to run this behind a nginx proxy or similar. You can see install instructions at reverse proxies section.

2.5 Updating

You can update IRCAnywhere by running the following two commands:

```
$ git pull
$ ./install.sh
```

And then restart accordingly, note client side files may be cached. A hard reset *ctrl+r* will force a full reload or try clearing your browser's cache.

CHAPTER 3

Reverse Proxies

3.1 Apache

You can run IRCAnywhere behind Apache quite easily, you can also choose to run it behind a subdirectory if you please, this example runs it behind *http://domain.com/ircanywhere*. You can do this by adding the following to your configuration file.

```
ProxyPreserveHost On
RewriteEngine On

RewriteRule ^/?ircanywhere/(.*) https://%{SERVER_NAME}:3000/$1 [P]

RewriteRule ^/?api/(.*) https://%{SERVER_NAME}:3000/api/$1 [P]
RewriteRule ^/?build/(.*) https://%{SERVER_NAME}:3000/build/$1 [P]
RewriteRule ^/?websocket/(.*) https://%{SERVER_NAME}:3000/websocket/$1 [P]
RewriteRule ^/?sounds/(.*) https://%{SERVER_NAME}:3000/sounds/$1 [P]

ProxyPassReverse /ircanywhere/ https://%{SERVER_NAME}:3000/

ProxyPassReverse /websocket https://%{SERVER_NAME}:3000/websocket/
ProxyPassReverse /build https://%{SERVER_NAME}:3000/build/
ProxyPassReverse /api https://%{SERVER_NAME}:3000/api/
ProxyPassReverse /sounds https://%{SERVER_NAME}:3000/sounds/
```

3.2 Nginx

The following will run ircanywhere under a subdomain in nginx, like apache this can be configured to be a top level domain or a sub directory easily.

```
server {
    listen 80;
    server_name ircanywhere.domain.com;

    location /websocket/ {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
```

```
    location / {
        proxy_http_version 1.1;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
        proxy_set_header   X-NginX-Proxy    true;
        proxy_set_header   Host             $http_host;
        proxy_set_header   Upgrade          $http_upgrade;
        proxy_redirect     off;
        proxy_pass         http://localhost:3000;
    }
}
```

CHAPTER 4

Using the module system

There are two ways to integrate your own code into IRCAnywhere, you can hook into the backend and frontend and choose to combine them in your modules.

Modules are located in the `modules/` folder and loaded automatically on startup, they require a specific directory structure to be loaded automatically. This guide will teach you how to create a basic hello world style module. For example, a module with server side code will have the following structure.

```
modules
`-- helloworld
    `-- server
        `-- index.js
```

Index.js will be the main entry point for the module, you can have other files and npm modules and require them into index.js if you need to.

Client side modules have the following directory structure.

```
modules
`-- helloworld
    `-- client
        `-- js
            `-- helloworld.js
        `-- less
        `-- templates
```

This code will be compiled into the javascript, css and templates if specified.

4.1 Server Side

Server side modules work by extending the *baseModule* object, each core component can be modified, extended, or have new functionality bound. Here is an example of how to bind a new command.

```
/* hello world example module */
baseModule.extend({
    commandManager: {
        'bind:helloworld': function (user, client, target, params) {
            console.log('hello world, you can use the arguments of this function to interact with the
            console.log(arguments);
        }
    }
});
```

Here we are extending the `commandManager` object, and telling the module manager to bind a new function named `helloworld`, this will automatically be picked up by the command manager and the command will be accessible via `/mycommand param1 param2`.

We can choose to override existing core functionality by using this method aswell, an example would be overriding the initialising function of any core component by doing `bind:init` **this should be done with caution**. We can use hooks to perform actions at specific intervals aswell by using `pre`, `post` and `hook` instead of `bind`. For example

```
ircHandler: {
    'pre:closed': function (next, client, message) {
        console.log('do something before ircHandler.closed() is called');

        next();
        // call closed
    }
}
```

This will call this function before `ircHnalder.closed()` is executed, allowing us to extend core functionality easily.

4.2 Client Side

Client side modules are compiled into the final javascript build, and can modify any of the frontend's functionality in a similar way to how the backend works. Because of how Ember works functionally, there isn't really a huge need for an API, we can just build our own controllers, views, templates exactly how they are built in the core frontend code.

In this example below you can see how a client side implementation of the `/helloworld` command can be created.

```
App.InputController.reopen({
    commands: {
        '/helloworld': function() {
            console.log('hello world');
            console.log(arguments);
        }
    },
});
```

We can use Ember's `reopen` to reopen any class and override existing core functionality. The client side javascript is currently undocumented and the module system is still partially complete, this section will be updated in the future.

CHAPTER 5

Server API

This is the IRCAnywhere Server API which is autogenerated from the source code and is very much a WIP.

5.1 Application

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/app.js

class Application.**Application**()
The applications's main object, contains all the startup functions. All of the objects contained in this prototype are extendable by standard rules.

Examples:

```
application.post('init', function(next) {
    console.log('do something after init() is run');
    next();
});
```

Returns void

Application.**verbose**
A flag to determine whether verbose logging is enabled or not

Type boolean

Application.**packagejson**
A copy of the project's package.json object

Type object

Application.**init**()
This is the main entry point for the application, it should NOT be called under any circumstances. However it can safely be extended by hooking onto the front or back of it using pre and post hooks. Treat this method like the main() function in a C application.

Returns void

Application.**cleanCollections**()
Clean *channelUsers* and *events* collection if needed. Usually when someone has installed 0.1-beta before installing this version and has incompatible data lingering around.

Returns void

`Application.`**`setupOplog`**`()`
This method initiates the oplog tailing query which will look for any incoming changes on the database. Incoming changes are then handled and sent to the global event emitter where other classes and modules can listen to for inserts, updates and deletes to a collection to do what they wish with the changes.

Returns void

`Application.`**`setupWinston`**`()`
This function sets up our winston logging levels and transports. You can safely extend or override this function and re-run it to re-initiate the winston loggers if you want to change the transport to send to loggly or something via a plugin.

Returns void

`Application.`**`handleError`**`()`
Handle things such as domain errors and properly report

Returns void

`Application.`**`setupNode`**`()`
Checks for a node record to store in the file system and database This is done to generate a 'unique' but always the same ID to identify the system so we can make way for clustering in the future.

Returns void

`Application.`**`selectCipherSuite`**`()`
This function will select a suitable cipher suite and return a string to be used in createServer

Returns void

`Application.`**`setupServer`**`()`
This function is responsible for setting up the express webserver we use to serve the static files and the sock.js server which hooks onto it to handle the websockets. None of the routes or rpc callbacks are handled here.

Returns void

5.2 ChannelManager

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/channels.js

class `ChannelManager.`**`ChannelManager`**`()`
This object is responsible for managing everything related to channel records, such as the handling of joins/parts/mode changes/topic changes and such. As always these functions are extendable and can be prevented or extended by using hooks.

Returns void

`ChannelManager.`**`queueJoin`**`(`*id*, *channel*, *key*`)`
Queues a channel for join

Arguments

- **`id`** (*`objectid`*) – A valid Mongo ObjectID for the networks collection
- **`channel`** (*`string`*) – A valid channel name
- **`key`** (*`string`*) – A key to join the channel if necessary

Returns void

ChannelManager.**getChannel**(*network*, *channel*)

Gets a tab record from the parameters passed in, strictly speaking this doesn't have to be a channel, a normal query window will also be returned. However this class doesn't need to work with anything other than channels.

A new object is created but not inserted into the database if the channel doesn't exist.

Arguments

- **network** (*string*) – A network string such as 'freenode'
- **channel** (*string*) – The name of a channel **with** the hash key '#ircanywhere'

Returns A promise with a channel object straight out of the database.

ChannelManager.**insertUsers**(*key*, *channel*, *users*[, *force*])

Inserts a user or an array of users into a channel record matching the network key network name and channel name, with the option to force an overwrite

Arguments

- **key** (*objectid*) – A valid Mongo ObjectID for the networks collection
- **channel** (*string*) – The channel name '#ircanywhere'
- **users** (*[object]*) – An array of valid user objects usually from a who/join output
- **[force]** (*boolean*) – Optional boolean whether to overwrite the contents of the channelUsers

Returns A promise containing final array of the users inserted

ChannelManager.**removeUsers**(*key*[, *channel*, *users*])

Removes a specific user from a channel, if users is omitted, channel should be equal to a nickname and that nickname will be removed from all channels records on that network.

Arguments

- **key** (*objectid*) – A valid Mongo ObjectID for the networks collection
- **[channel]** (*string*) – A valid channel name
- **users** (*array*) – An array of users to remove from the network *or* channel

Returns void

ChannelManager.**updateUsers**(*key*, *users*, *values*)

Updates a user or an array of users from the specific channel with the values passed in.

Arguments

- **key** (*objectid*) – A valid Mongo ObjectID for the networks collection
- **users** (*array*) – A valid users array
- **values** (*object*) – A hash of keys and values to be replaced in the users array

Returns void

ChannelManager.**updateModes**(*key*, *capab*, *channel*, *mode*)

Takes a mode string, parses it and handles any updates to any records relating to the specific channel. This handles user updates and such, it shouldn't really be called externally, however can be pre and post hooked like all other functions in this object.

Arguments

- **key** (*objectid*) – A valid Mongo ObjectID for the networks collection

- **capab** (*object*) – A valid capabilities object from the 'registered' event
- **channel** (*string*) – Channel name
- **mode** (*string*) – Mode string

Returns void

ChannelManager.**updateTopic**(*key*, *channel*, *topic*, *setby*)

Updates the specific channel's topic and setby in the internal records.

Arguments

- **key** (*objectid*) – A valid Mongo ObjectID for the networks collection
- **channel** (*string*) – A valid channel name
- **topic** (*string*) – The new topic
- **setby** (*string*) – A setter string, usually in the format of 'nickname!username@hostname'

Returns void

5.3 CommandManager

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/commands.js

class CommandManager.**CommandManager**()

Responsible for handling all incoming commands from websocket clients

Returns void

CommandManager.**init**()

Called when the application is booted and everything is ready, sets up an observer on the commands collection for inserts and handles them accordingly. Also sets up aliases, this should not be recalled, although can be extended to setup your own aliases.

Returns void

CommandManager.**_ban**(*client*, *channel*, *nickname*, *ban*)

Sets +b/-b on a specific channel on a chosen client, not extendable and private.

Arguments

- **client** (*object*) – A valid client object
- **channel** (*string*) – A channel name
- **nickname** (*string*) – A nickname or hostname to ban
- **ban** (*boolean*) – Whether to ban or unban

Returns void

CommandManager.**createAlias**(*command*, *alias*)

Creates an alias from the first parameter to the remaining ones.

Examples:

```
commandManager.createAlias('/part', '/p', '/leave');
// sets an alias for /p and /leave to forward to /part
```

Arguments

- **command** (*string*) – A command to alias
- **alias** (*...string*) – A command to map to

Returns void

CommandManager.**parseCommand**(*user*, *client*, *target*, *command*)

Parse a command string and determine where to send it after that based on what it is ie just text or a string like: '/join #channel'

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **command** (*string*) – The command string

Returns void

CommandManager.**msg**(*user*, *client*, *target*, *params*, *out*, *id*)

'/nickserv' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string
- **out** (*boolean*) – Used to force the message to target or params[0]
- **id** (*objectid*) – The object id of the command so we can remove it if we need to

Returns void

CommandManager.**msg**(*user*, *client*, *target*, *params*, *out*, *id*)

'/msg' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string
- **out** (*boolean*) – Used to force the message to target or params[0]
- **id** (*objectid*) – The object id of the command so we can remove it if we need to

Returns void

CommandManager.**notice**(*user*, *client*, *target*, *params*)

'/notice' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**me**(*user*, *client*, *target*, *params*)
'/me' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**join**(*user*, *client*, *target*, *params*)
'/join' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**part**(*user*, *client*, *target*, *params*)
'/part' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**cycle**(*user*, *client*, *target*, *params*)
'/cycle' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**topic**(*user*, *client*, *target*, *params*)
'/topic' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**mode**(*user*, *client*, *target*, *params*)
'/mode' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**invite**(*user*, *client*, *target*, *params*)
'/invite' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**kick**(*user*, *client*, *target*, *params*)
'/kick' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**kickban**(*user*, *client*, *target*, *params*)
'/kickban' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object

- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**ban**(*user*, *client*, *target*, *params*)
'/ban' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**unban**(*user*, *client*, *target*, *params*)
'/unban' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**nick**(*user*, *client*, *target*, *params*)
'/nick' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**ctcp**(*user*, *client*, *target*, *params*)
'/ctcp' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**away**(*user*, *client*, *target*, *params*)
'/away' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**unaway**(*user*, *client*)
'/unaway' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object

Returns void

CommandManager.**close**(*user*, *client*, *target*)
'/close' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username

Returns void

CommandManager.**query**(*user*, *client*, *target*)
'/query' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username

Returns void

CommandManager.**quit**(*user*, *client*)
'/quit' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object

Returns void

CommandManager.**reconnect**(*user*, *client*)
'/reconnect' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object

Returns void

CommandManager.**list**(*user*, *client*, *target*, *params*)
'/list' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**whois**(*user*, *client*, *target*, *params*)
'/whois' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

CommandManager.**raw**(*user*, *client*, *target*, *params*)
'/raw' command

Arguments

- **user** (*object*) – A valid user object
- **client** (*object*) – A valid client object
- **target** (*string*) – Target to send command to, usually a channel or username
- **params** (*string*) – The command string

Returns void

5.4 EventManager

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/events.js

class EventManager.**EventManager**()
Constructor, does nothing

Returns void

EventManager.**channelEvents**
A list of events relating to channels

EventManager.**_insert**(*client*, *message*, *type*[, *user*, *force*])
Inserts an event into a backlog after all the checking has been done this api is private and EventManager.insertEvent should be used instead

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object from *irc-message*
- **type** (*string*) – Event type
- **[user]** (*object*) – An optional user object
- **[force]** (*boolean*) – An optional force boolean to force the event into the '*' status window

Returns void

EventManager.**insertEvent** (*client*, *message*, *type*, *cb*)

Inserts an event into the backlog, takes a client and message object and a type Usually 'privmsg' or 'join' etc.

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object from *irc-message*
- **type** (*string*) – Event type
- **cb** (*function*) – Callback function to be executed after insert

Returns void

EventManager.**determineHighlight** (*client*, *message*, *type*, *ours*)

Determine whether a message should be marked as a highlight or not for the specific IRC client. Currently this does not support anything other than looking at their nickname.

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object from *irc-message*
- **type** (*string*) – Event type
- **ours** (*boolean*) – Whether this message comes from this client

Returns true or false

EventManager.**getPrefix** (*client*, *user*)

Gets the channel prefix for the irc client and the user object. A valid object returned is in the format of:

```
{prefix: '+', sort: 5};
```

Arguments

- **client** (*object*) – A valid client object
- **user** (*object*) – A valid user object

Returns A valid prefix object

EventManager.**getEventByType** (*type*, *network*, *userId*)

Gets the most recent event from the database by its type.

Arguments

- **type** (*string*) – Event type
- **network** (*objectid*) – Event network
- **userId** (*string*) – Id of the user

Returns Promise that resolves to event.

EventManager.**getUserPlayback**(*network*, *userId*)
Gets the message playback for an IRC server user since he was last seen.

Arguments

- **network** (*objectid*) – Network to get playback from
- **userId** (*string*) – Id of the user

Returns Promise that resolves to array of playback events.

5.5 IRCFactory

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/factory.js

class IRCFactory.**IRCFactory**()
The IRCFactory object which handles communication with the irc-factory package This object is not hookable or extendable because plugins can deny the execution of functions when they hook into it, the results could be disasterous. If incoming events need to be hooked onto you could hook onto the IRCHandler object.

The default *irc-factory* options are below:

```
{
    events: 31920,
    rpc: 31930,
    automaticSetup: true
}
```

The *fork* setting comes from our configuration object and is inserted when *init* is ran.

Returns void

IRCFactory.**options**
The *irc-factory* options to use

Type object

IRCFactory.**init**()
Initiates the irc factory and it's connection and sets up an event handler when the application is ready to run.

Returns void

IRCFactory.**handleEvent**(*event*, *object*)
Handles incoming factory events, events are expected to come in the following format:

```
[ '52d3fc718132f8486dcde1d0', 'privmsg' ] { nickname: 'ricki-',
    username: 'ia1',
    hostname: '127.0.0.1',
    target: '#ircanywhere-test',
    message: '#ircanywhere-test WORD UP BROSEPTH',
    time: '2014-01-22T18:20:57.323Z',
```

More advanced docs can be found at https://github.com/ircanywhere/irc-factory/wiki/Events

Arguments

- **event** (*[string]*) – A valid event array from irc-factory *['52d3fc718132f8486dcde1d0', 'privmsg']*
- **object** (*object*) – A valid event object from irc-factory

Returns void

IRCFactory.**create**(*network*)
Sends the command to *irc-factory* to create a new irc client with the given settings. If the client already exists it will be dropped by *irc-factory*.

Arguments

- **network** (*object*) – A valid client object

Returns void

IRCFactory.**destroy**(*key*, *forced*)
Sends the command to destroy a client with the given key. If the client doesn't exist the command will just be dropped.

Arguments

- **key** (*objectid*) – A client key which has the type of a Mongo ObjectID
- **forced** (*boolean*) – Whether we forced a client disconnect or not

Returns void

IRCFactory.**send**(*key*, *command*, *args*)
Calls an RPC command on the irc-factory client, usually used to send commands such as /WHO etc. It's probably best to use CommandManager in most cases

Arguments

- **key** (*objectid*) – A client key which has the type of a Mongo ObjectID
- **command** (*string*) – An IRC command to send, such as 'mode' or 'join'
- **args** (*array*) – An array of arguments to send delimited by a space.

Returns void

5.6 IdentdServer

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/identd.js

class IdentdServer.**IdentdServer**()
This is the IdentdServer object which creates an integrated identd server and can be turned off via the configuration, this is a singleton and should never be instantiated more than once.

The configuration option *identd.enable* and *identd.port* will control whether this runs and what port it runs on, the default port is 113 but you can bind it to whatever you like and use iptables to forward to 113, without doing that IRCAnywhere will need elevated permissions to bind.

Returns void

IdentdServer.**init**()
Initiates the identd server and handles any configuration options

Returns void

IdentdServer.**onData**(*socket*, *data*)
Handles incoming data to the identd server, this shouldn't ever be called, but the documentation is here so people know what the function is doing and where responses are handled etc. Protocol information can be found at http://en.wikipedia.org/wiki/Ident_protocol

Arguments

- **socket** (*socket*) – A valid socket from net.createServer callback
- **data** (*bufferobject*) – http://nodejs.org/api/net.html#net_event_data

Returns void

IdentdServer.**parse**(*line*)
Once the data has been handled it needs to be parsed so we can figure out what the identd request is and respond to it accordingly to validate our connecting user.

Arguments

- **line** (*string*) – The parsed ident line

Returns The response for the requester

5.7 IRCHandler

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/irchandler.js

class IRCHandler.**IRCHandler**()
The object responsible for handling an event from IRCFactory none of these should be called directly, however they can be hooked onto or have their actions prevented or replaced. The function names equal directly to irc-factory events and are case sensitive to them.

Returns void

IRCHandler.**blacklisted**
An array of blacklisted commands which should be ignored

Type array

IRCHandler.**_formatRaw**(*raw*)
Formats an array of RAW IRC strings, taking off the :leguin.freenode.net 251 ricki- : at the start, returns an array of strings with it removed

Arguments

- **raw** (*array*) – An array of raw IRC strings to format

Returns A formatted array of the inputted strings

IRCHandler.**opened**(*client*, *message*)
Handles the opened event from *irc-factory* which just tells us what localPort and any other information relating to the client so we can make sure the identd server is working.

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**registered**(*client*, *message*)
Handles the registered event, this will only ever be called when an IRC connection has been fully established and we've recieved the *registered* events. This means when we reconnect to an already established connection we won't get this event again.

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**closed**(*client*, *message*)
Handles a closed connection

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**failed**(*client*, *message*)
Handles a failed event, which is emitted when the retry attempts are exhaused

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**lusers**(*client*, *message*)
Handles an incoming lusers event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**motd**(*client*, *message*)
Handles an incoming motd event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**join**(*client*, *message*)
Handles an incoming join event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**part**(*client*, *message*)
Handles an incoming part event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**kick**(*client*, *message*)
Handles an incoming kick event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**quit**(*client*, *message*)
Handles an incoming quit event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**nick**(*client*, *message*)
Handles an incoming nick change event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**who**(*client*, *message*)
Handles an incoming who event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**names**(*client*, *message*)
Handles an incoming names event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**mode**(*client*, *message*)
Handles an incoming mode notify event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**mode_change**(*client*, *message*)
Handles an incoming mode change event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

IRCHandler.**topic**(*client*, *message*)
Handles an incoming topic notify event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**topic_change**(*client*, *message*)
Handles an incoming topic change event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**privmsg**(*client*, *message*)
Handles an incoming privmsg event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**action**(*client*, *message*)
Handles an incoming action event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**notice**(*client*, *message*)
Handles an incoming notice event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**usermode**(*client*, *message*)
Handles an incoming usermode event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**ctcp_response**(*client*, *message*)
Handles an incoming ctcp_response event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**ctcp_request**(*client*, *message*)
Handles an incoming ctcp request event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**unknown**(*client*, *message*)
Handles an incoming unknown event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**banlist**(*client*, *message*)
Handles an incoming banlist event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**invitelist**(*client*, *message*)
Handles an incoming invitelist event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**exceptlist**(*client*, *message*)
Handles an incoming exceptlist event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**quietlist**(*client*, *message*)
Handles an incoming quietlist event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**list**(*client*, *message*)
Handles an incoming list event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

IRCHandler.**whois**(*client*, *message*)
Handles an incoming whois event

Arguments

- **client** (*object*) – A valid client object
- **message** (*object*) – A valid message object

Returns void

5.8 IRCServer

(c) 2013-2014 http://ircanywhere.com

Author: Rodrigo Silveira

IRCAnywhere server/server.js

class IRCServer.**IRCServer**()
This is the IRC server that manages IRC client connections to ircanywhere. The IRC server can be turned off by configuration. this is a singleton and should never be instantiated more than once.

The configuration option *ircServer.enable* and *ircServer.port* will control whether this runs and what port it runs on. The default port is 6667.

Returns void

IRCServer.**init**()
Setup server and start listening for connections.

Returns void

IRCServer.**onConnect**(*socket*)
Handles new server connection. Starts a session.

Arguments

- **socket** (*object*) – Connection socket to the client

Returns void

5.9 ModeParser

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/modeparser.js

class ModeParser.**ModeParser**()
Responsible for parsing mode strings into unstandable actions and also responsible for applying those actions to a channel/user object.

None of these functions can be hooked onto or extended seen as though it's just not needed and could be malicious if people are altering mode string, bugs relating to this are difficult to find, if you want to hook a mode change hook to IRCHandler.mode_change()

Returns void

ModeParser.**sortModes**(*capabilities*, *modes*)
Sorts a mode string into an object of instructions that we can use to perform actions based on what the mode string suggests, ie apply operator to 'someone', or set +m on the channel

Arguments

- **capabilities** (*object*) – A valid capabilities object from a client
- **modes** (*string*) – A mode string *+no-v rickibalboa Gnasher*

Returns A valid modeArray object

ModeParser.**changeModes**(*capabilities*, *modes*, *modeArray*)
Handles the object of instructions returned from sortModes, and applies them

Arguments

- **capabilities** (*object*) – A valid capabilities object from a client
- **modes** (*object*) – The current mode string for the channel (not including all parameters)
- **modeArray** (*object*) – A valid modeArray object from *sortModes()*

Returns The channel mode string with the changes applied.

ModeParser.**handleParams**(*capabilities*, *users*, *modeArray*)
Applies any mode changes that contain status related modes, usually qaohv modes minus: rickibalboa: -o > will remove the o flag from the nickname record minus: rickibalboa: +v > will set the v flag on the nickname record

Arguments

- **capabilities** (*object*) – A valid capabilities object from a client
- **users** (*object*) – A valid users array for a channel
- **modeArray** (*object*) – A valid modeArray from *sortModes*

Returns An array of users that have been affected by the mode change

5.10 Module

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/basemodule.js

class Module.**Module**()
Base class for creating modules

Returns void.. js:class:: Module.undefined()

Module.**extend**(*object*)
Extend function used to extend objects with base module functionality so they can hook into core events. This should only be called once per module, if a module needs to contain multiple files, this object can be pulled in from multiple files with exports and require and the output can be extended once.

Arguments

- **object** (*object*) – The module object to extend

Returns void.. js:class:: Module.undefined()

Module.**bindFunction**(*key*, *classObject*, *split*, *fn*, *object*)
Used to bind hooks for a function on a core object

Arguments

- **key** (*string*) – Class name
- **classObject** (*object*) – Class object
- **split** (*array*) – An array containing two values, 'pre|post|hook' and method name
- **fn** (*function*) – A callback
- **object** (*object*) – The base object where the fn belongs in

Returns void

5.11 ModuleManager

(c) 2013-2014 http://ircanywhere.com

Author: Rodrigo Silveira

IRCAnywhere server/module.js

class ModuleManager.**ModuleManager**()
Handles loading of modules.

Returns void

ModuleManager.**loadModule**(*moduleName*)
Loads a module by name. The name should be the name of the folder containing the module.

Arguments

- **moduleName** (*string*) – Name of module to load.

Returns void

ModuleManager.**loadAllModules**()
Loads all modules.

Returns void

ModuleManager.**bindModule**(*module*)
Bind events and expose module to core functionality and vice versa

Arguments

- **module** (*object*) – A valid module object returned from require()

Returns void

5.12 NetworkManager

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/networks.js

class NetworkManager.**NetworkManager**()
Responsible for handling everything related to networks, such as tracking changes removing, creating, changing tabs, creating and deleting networks etc.

Returns void

NetworkManager.**flags**
An object containing the valid network statuses

Type object

NetworkManager.**init**()
Called when the application is ready to proceed, this sets up event listeners for changes on networks and tabs collections and updates the Client object with the changes to essentially keep the object in sync with the collection so we can do fast lookups, but writes to the collection will propogate through and update Clients

Returns void

NetworkManager.**getClients**()
Gets a list of networks, used by IRCFactory on synchronise to determine who to connect on startup, doesn't ever really need to be called also can be modified with hooks to return more information if needed.

Returns A promise containing the clients that should be started up

NetworkManager.**getClientsForUSer**(*userId*)
Gets a list of active networks for a user.

Arguments

- **userId** (*string*) – Id of the user

Returns A promise that will resolve to the clients for the given user

NetworkManager.**getActiveChannelsForUser**(*userId*, *networkId*)
Gets a list of active channels for a user.

Arguments

- **userId** (*string*) – Id of the user
- **networkId** (*string*) – Id of the network

Returns A promise that will resolve to the active channels for the given user

`NetworkManager.`**`addNetworkApi`**(*req*)

Handles the add network api call, basically handling authentication validating the parameters and input, and on success passes the information to *addNetwork()* which handles everything else

Arguments

- **`req`** (*`object`*) – A valid request object from express

Returns An output object for the API call

`NetworkManager.`**`editNetworkApi`**(*req*)

Handles the edit network api call, everything the add network call does except it takes a network ID as a parameter validates the new data. On success it passes to *editNetwork()* which handles the rest.

Arguments

- **`req`** (*`object`*) – A valid request object from express

Returns An output object for the API call

`NetworkManager.`**`addNetwork`**(*user*, *network*, *status*)

Adds a network using the settings specified to the user's set of networks This just adds it to the database and doesn't attempt to start it up.

Arguments

- **`user`** (*`object`*) – A valid user object from the *users* collection
- **`network`** (*`object`*) – A valid network object to insert
- **`status`** (*`string`*) – A valid network status

Returns A promise to determine whether the insert worked or not

`NetworkManager.`**`editNetwork`**(*user*, *network*)

Edits an existing network, updating the record in the database. We'll inform irc-factory that the network information has changed and perform a reconnect.

Arguments

- **`user`** (*`object`*) – A valid user object from the *users* collection
- **`network`** (*`object`*) – A valid network object to update

Returns A promise to determine whether the insert worked or not

`NetworkManager.`**`addTab`**(*client*, *target*, *type*[, *select*, *active*])

Adds a tab to the client's (network unique to user) tabs, this can be a channel or a username.

Arguments

- **`client`** (*`object`*) – A valid client object
- **`target`** (*`string`*) – The name of the tab being created
- **`type`** (*`string`*) – The type of the tab either 'query', 'channel' or 'network'
- **`[select]`** (*`boolean`*) – Whether to mark the tab as selected or not, defaults to false
- **`[active]`** (*`boolean`*) – Whether to mark the tab as active or not, defaults to true

Returns void

`NetworkManager.`**`activeTab`**(*client*[, *target*, *activate*])

Changes a tabs activity (not selection) - for example when you're kicked from a channel the tab wont be removed

it will be just set to active: false so when you see it in the interface it will appear as (#ircanywhere) instead of #ircanywhere We can omit target and call activeTab(client, false) to set them all to false (such as on disconnect)

Arguments

- **client** (*object*) – A valid client object
- **[target]** (*string*) – The name of the tab being altered, discard to mark all as active or inactive.
- **activate** (*boolean*) – Whether to set the tab as active or not

Returns void

NetworkManager.**removeTab**(*client*[, *target*])

Removes the specified tab, be careful because this doesn't re-select one, you're expected to look for a removed tab, if it's the currently selected one, go back to a different one.

Arguments

- **client** (*object*) – A valid client object
- **[target]** (*string*) – The name of the tab being altered, discard to remove all.

Returns void

NetworkManager.**connectNetwork**(*network*)

Connect the specified network record, should only really be called when creating a new network as IRCFactory will load the client up on startup and then determine whether to connect the network itself based on the options.

However, it's also called when it appears that there is no connected client on the /reconnect command (and any other similar commands). We can determine this (sloppy) from checking client.internal.status. If in the case that it does exist, it doesn't matter if this is called really because irc-factory will prevent a re-write if the key is the same. We could consider looking at the response from factory synchronize but it might not yield a good result because of newly created clients since startup.

Arguments

- **network** (*object*) – A valid network or client object

Returns void

NetworkManager.**changeStatus**(*query*, *status*)

Update the status for a specific network specified by a MongoDB query. The reason for this and not a straight ID is so we can do certain things such as checking if a network is marked as 'disconnected' during the *closed* event to determine whether to keep it as 'disconnected' or mark it as 'closed'. So we can do much more elaborate queries here than just ID checking

Arguments

- **query** (*object*) – A MongoDB query to select a network
- **status** (*boolean*) – A valid network status

Returns void

5.13 RPCHandler

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/rpc.js

class RPCHandler.**RPCHandler**()
Singleton class to handle the inbound and outbound RPC calls on the websocket lines

Returns void

RPCHandler.**init**()
Called when the application is ready, sets up an observer on our collections so we can figure out whether we need to propogate them to clients.

Returns void

RPCHandler.**push**(*uid*, *command*, *data*)
Pushes the data and command out to any sockets associated to that uid

Arguments

- **uid** (*string*) – A valid user id converted from an object ID
- **command** (*string*) – The command to send
- **data** (*string*) – The json data to send

Returns void

RPCHandler.**handleUsersUpdate**(*doc*)
Handles any update changes to the users collection and sends changes to clients

Arguments

- **doc** (*object*) – A valid MongoDB document with an _id

Returns void

RPCHandler.**handleNetworksAll**(*doc*)
Handles any all changes to the network collection

Arguments

- **doc** (*object*) – A valid MongoDB document with an _id

Returns void

RPCHandler.**handleTabsAll**(*doc*)
Handles all changes to the tabs collections

Arguments

- **doc** (*object*) – A valid MongoDB document with an _id

Returns void

RPCHandler.**handleEventsAll**(*doc*)
Handles any changes to the events collection

Arguments

- **doc** (*object*) – A valid MongoDB document with an _id

Returns void

RPCHandler.**handleCommandsAll**(*doc*)
Handles all operations on the commands collection

Arguments

- **doc** (*object*) – A valid MongoDB document with an _id

Returns void

RPCHandler.**handleChannelUsersAll**(*doc*)
Handles any changes on the channelUsers collection

Arguments

- **doc** (*object*) – A valid MongoDB document with an _id

Returns void

RPCHandler.**onSocketOpen**(*socket*)
Handles a new websocket opening and attaches the RPC events

Arguments

- **socket** (*object*) – A valid sock.js socket

Returns void

RPCHandler.**handleAuth**(*socket*, *data*)
Handles the authentication command sent to us from websocket clients Authenticates us against login tokens in the user record, disconnects if expired or incorrect.

Arguments

- **socket** (*object*) – A valid sock.js socket
- **data** (*object*) – A valid data object from sock.js

Returns void

RPCHandler.**handleConnect**(*socket*)
Handles new websocket clients, this is only done after they have been authenticated and it's been accepted.

Arguments

- **socket** (*object*) – A valid sock.js socket

Returns void

RPCHandler.**handleCommand**(*socket*, *data*, *exec*)
Handles the exec command RPC call. Which should be used to execute /commands from the clientside without inserting them into the backlog.

Arguments

- **socket** (*object*) – A valid sock.js socket
- **data** (*object*) – A valid data object from sock.js
- **exec** (*boolean*) – Whether to exec the command or backlog it

Returns void

RPCHandler.**handleReadEvents**(*socket*, *data*)
Handles the command which marks events as read. It takes a MongoDB query and updates them with that query.

Arguments

- **socket** (*object*) – A valid sock.js socket
- **data** (*object*) – A valid data object from sock.js

Returns void

RPCHandler.**handleSelectTab**(*socket*, *data*)
Handles the selectTab command which is used to change the currently active tab for that user.

Arguments

- **socket** (*object*) – A valid sock.js socket
- **data** (*object*) – A valid data object from sock.js

Returns void

RPCHandler.**handleUpdateTab**(*socket*, *data*)
Handles the update tab command, we're allowed to change client side only settings here `hiddenUsers` and `hiddenEvents` only at the moment.

Arguments

- **socket** (*object*) – A valid sock.js socket
- **data** (*object*) – A valid data object from sock.js

Returns void

RPCHandler.**handleInsertTab**(*socket*, *data*)
Allows users to create new tabs on the fly from the client side. Restricted to `channel` and `query` tabs.

Arguments

- **socket** (*object*) – A valid sock.js socket
- **data** (*object*) – A valid data object from sock.js

Returns void

RPCHandler.**handleGetEvents**(*socket*, *data*)
Handles queries to the events collection

Arguments

- **socket** (*object*) – A valid sock.js socket
- **data** (*object*) – A valid data object from sock.js

Returns void

5.14 ServerSession

(c) 2013-2014 http://ircanywhere.com

Author: Rodrigo Silveira

IRCAnywhere server/serversession.js

disconnectUser()
Disconnects the socket.

Returns void.. js:class:: ServerSession.ServerSession(socket)

Handles the communication between an IRC client and ircanywhere's IRC server. Instantiated on every new client connection.

Arguments

- **socket** (*object*) – Connection socket to the client

Returns void

ServerSession.**debug**
A flag to determine whether debug logging is enabled or not

Type boolean

ServerSession.**init**()
Initializes session.

Returns void

ServerSession.**pass**(*message*)
Handles PASS message from client. Stores password for login.

Arguments

- **message** (*object*) – Received message

Returns void

ServerSession.**nick**(*message*)
Handles NICK message from client. If message arrives before welcome, just store nickname temporarily to use on the welcome message. Otherwise forward it.

Arguments

- **message** (*object*) – Received message

Returns void

ServerSession.**quit**()
Handles QUIT message from client. Disconnects the user.

Returns void

ServerSession.**user**(*message*)
Handles USER message from client. Start login sequence. Username should contain network information if more then one network is registered. Username with network is in the form:

Arguments

- **message** (*object*) – Received message

Returns void

ServerSession.**setup**()
Sets up client to listen to IRC activity.

Returns void

ServerSession.**handleEvent**(*event*)
Handle IRC events.

Arguments

- **event** (*object*) – Event to handle

Returns void

ServerSession.**handleIrcMessage**(*ircMessage*)
Forwards messages that are not stored in the events collection in the database.

Arguments

- **ircMessage** (*object*) – Irc Message object

Returns void

ServerSession.**sendWelcome**()
Sends stored welcome message from network to client. Message order is registered, lusers, nick (to set to stored nick), motd and usermode.

ServerSession.**sendJoins**()
Sends to client a join message for each active channel tab.

ServerSession.**sendChannelInfo**(*tab*)
Sends channel information, such as NAMES, TOPIC etc

Arguments

- **tab** (*object*) – Channel tab

Returns void

ServerSession.**sendPlayback**()
Sends playback messages to client.

Returns void

ServerSession.**privmsg**(*message*)
Handles PRIVMSG messages from client. Forwards to ircHandler and to ircFactory.

Arguments

- **message** (*object*) – Received message

Returns void

ServerSession.**onClientMessage**(*message*, *command*)
Handles all message that do not have a specific handler.

Arguments

- **message** (*object*) – Received message
- **command** (*string*) – Messages command

Returns void

ServerSession.**sendRaw**(*rawMessage*)
Sends a raw message to the client

Arguments

- **rawMessage** (*string*) – Raw message

Returns void

5.15 UserManager

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/users.js

loginServerUser(*email*, *password*)
Handles login of IRC server user

Arguments

- **email** (*string*) – User email
- **password** (*string*) – User password

updateLastSeen(*userId*[, *lastSeen*])
Update lastSeen entry of user.

Arguments

- **userId** (*string*) – Id of the user
- **[lastSeen]** (*date*) – New lastSeen value

class UserManager.**UserManager**()

Responsible for handling user related actions ie registering, logging in, forgot passwords etc. Most of these actions are triggered via API calls.

Returns void

UserManager.**init**()

Sets up the API routes and anything else needed by the user manager class. Such as timers and the SMTP connection

Returns void

UserManager.**timeOutInactive**()

Responsible for disconnecting any inactive users

This function is ran every hour or so, but not perfectly precise, but it shouldn't drift off too much because it re-corrects it self.

Returns void

UserManager.**isAuthenticated**(*data*)

Checks the sent in authentication string (should be "token=actualToken") all in string format, this is how it is sent in the authentication command and how it lies as a cookie. It also takes a full cookie string, such as "someKey=1; someOtherKey=2; token=actualToken" and the token will only be parsed and used.

Returns a valid user object which can be used to set on the socket for example or HTTP request, returns false if invalid

Arguments

- **data** (*object*) – A valid data object from sock.js

UserManager.**registerUser**(*req*)

Handles user registrations, it takes req and res objects from express at the moment however it should probably stay this way, because the api to register a user is at /api/register. I can't see a reason to change this to take individual parameters.

Arguments

- **req** (*object*) – A valid request object from express

Returns An output object for the API call

UserManager.**userLogin**(*req*, *res*)

Handles the login call to /api/login and sets an appropriate cookie if successful.

Arguments

- **req** (*object*) – A valid request object from express
- **res** (*object*) – A valid response object from express

Returns An output object for the API call

UserManager.**userLogout**(*req*)

Handles the call to /api/logout which is self explanatory.

Arguments

- **req** (*object*) – A valid request object from express

Returns An output object for the API call

`UserManager.`**`forgotPassword`**(*req*)

Handles the call to /api/forgot to send a forgot password link

Arguments

- **`req`** (*object*) – A valid request object from express

Returns An output object for the API call

`UserManager.`**`resetPassword`**(*req*)

Handles the call to /api/reset which will be called when the reset password link is visited Checking is done to make sure a token exists in a user record.

Arguments

- **`req`** (*object*) – A valid request object from express

Returns An output object for the API call

`UserManager.`**`updateSettings`**(*req*)

Handles the call to /api/settings/updatesettings which will update the settings for that user checking for authentication and validating if necessary.

Arguments

- **`req`** (*object*) – A valid request object from express

Returns An output object for the API call

`UserManager.`**`resetPassword`**(*req*)

Handles the call to /api/settings/changepassword which is almost identical to resetPassword however it checks for authentication and then changes the password using that user, it doesn't take a token though.

Arguments

- **`req`** (*object*) – A valid request object from express

Returns An output object for the API call

`UserManager.`**`updatePassword`**(*user*, *password*, *confirmPassword*[, *currentPassword*])

Updates a users password, doesn't bypass any checkings, just doesn't define how you select the user, so via a token or direct user object

Arguments

- **`user`** (*promise*) – A valid promise object from *isAuthenticated*
- **`password`** (*string*) – The new password to set
- **`confirmPassword`** (*string*) – The same password again
- **`[currentPassword]`** (*string*) – The current password

Returns An output object for the API call

`UserManager.`**`onUserLogin`**(*me*[, *force*])

An event which is called when a successful login occurs, this logic is kept out of the handler for /api/login because it's specific to a different section of the application which is the networkManager and ircFactory.

Arguments

- **`me`** (*object*) – A valid user object
- **`[force]`** (*boolean*) – Whether to force the reconnect of a disconnected client

Returns void

UserManager.**parse**(*file*, *replace*)
Looks for a template and parses the {{tags}} into the values in replace and returns a string, used to parse emails. Very basic parsing which will probably be replaced by something more powerful in the future with HTML outputs.

Arguments

- **file** (*string*) – The name of the email template
- **replace** (*object*) – A hash of keys and values to replace in the template

Returns A parsed email template

5.16 WebSocket

(c) 2013-2014 http://ircanywhere.com

Author: Ricki Hastings

IRCAnywhere server/websocket.js

class WebSocket.**WebSocket**(*socket*)
Wrapper for sock.js sockets.

Arguments

- **socket** (*object*) – A valid sock.js socket

Returns void

WebSocket.**bindEvents**()
Binds our sock.js events to _socket.

Returns void

WebSocket.**isValid**(*parsed*)
Checks if an incoming message object is valid.

Arguments

- **parsed** (*object*) – An incoming message object to parse

Returns Whether the object is valid or not

WebSocket.**onMessage**(*raw*)
Handles an incoming message.

Arguments

- **raw** (*string*) – A raw line from a sock.js websocket

Returns void

WebSocket.**onClose**()
Handles closing the websocket connection.

Returns void

WebSocket.**send**(*event*, *data*[, *close*])
Sends outgoing packets

Arguments

- **event** (*string*) – The event to send

- **data** (*object*) – The data object to send, should be JSON
- **[close]** (*boolean*) – Whether to close the connection after the data has been sent or not

Returns void

`WebSocket.`**sendBurst**(*data*)

Compiles a temporary GET route and sends it to a socket

Arguments

- **data** (*object*) – The data object to push into a route and send down the socket

Returns void

A

Application.Application() (class), 13
Application.cleanCollections() (Application method), 13
Application.handleError() (Application method), 14
Application.init() (Application method), 13
Application.packagejson (Application attribute), 13
Application.selectCipherSuite() (Application method), 14
Application.setupNode() (Application method), 14
Application.setupOplog() (Application method), 14
Application.setupServer() (Application method), 14
Application.setupWinston() (Application method), 14
Application.verbose (Application attribute), 13

C

ChannelManager.ChannelManager() (class), 14
ChannelManager.getChannel() (ChannelManager method), 15
ChannelManager.insertUsers() (ChannelManager method), 15
ChannelManager.queueJoin() (ChannelManager method), 14
ChannelManager.removeUsers() (ChannelManager method), 15
ChannelManager.updateModes() (ChannelManager method), 15
ChannelManager.updateTopic() (ChannelManager method), 16
ChannelManager.updateUsers() (ChannelManager method), 15
CommandManager._ban() (CommandManager method), 16
CommandManager.away() (CommandManager method), 20
CommandManager.ban() (CommandManager method), 20
CommandManager.close() (CommandManager method), 21
CommandManager.CommandManager() (class), 16
CommandManager.createAlias() (CommandManager method), 16
CommandManager.ctcp() (CommandManager method), 20
CommandManager.cycle() (CommandManager method), 18
CommandManager.init() (CommandManager method), 16
CommandManager.invite() (CommandManager method), 19
CommandManager.join() (CommandManager method), 18
CommandManager.kick() (CommandManager method), 19
CommandManager.kickban() (CommandManager method), 19
CommandManager.list() (CommandManager method), 21
CommandManager.me() (CommandManager method), 18
CommandManager.mode() (CommandManager method), 19
CommandManager.msg() (CommandManager method), 17
CommandManager.nick() (CommandManager method), 20
CommandManager.notice() (CommandManager method), 17
CommandManager.parseCommand() (CommandManager method), 17
CommandManager.part() (CommandManager method), 18
CommandManager.query() (CommandManager method), 21
CommandManager.quit() (CommandManager method), 21
CommandManager.raw() (CommandManager method), 22
CommandManager.reconnect() (CommandManager method), 21
CommandManager.topic() (CommandManager method), 19
CommandManager.unaway() (CommandManager

method), 21
CommandManager.unban() (CommandManager method), 20
CommandManager.whois() (CommandManager method), 22

D

disconnectUser() (built-in function), 39

E

EventManager._insert() (EventManager method), 22
EventManager.channelEvents (EventManager attribute), 22
EventManager.determineHighlight() (EventManager method), 23
EventManager.EventManager() (class), 22
EventManager.getEventByType() (EventManager method), 23
EventManager.getPrefix() (EventManager method), 23
EventManager.getUserPlayback() (EventManager method), 24
EventManager.insertEvent() (EventManager method), 23

I

IdentdServer.IdentdServer() (class), 25
IdentdServer.init() (IdentdServer method), 25
IdentdServer.onData() (IdentdServer method), 26
IdentdServer.parse() (IdentdServer method), 26
IRCFactory.create() (IRCFactory method), 25
IRCFactory.destroy() (IRCFactory method), 25
IRCFactory.handleEvent() (IRCFactory method), 24
IRCFactory.init() (IRCFactory method), 24
IRCFactory.IRCFactory() (class), 24
IRCFactory.options (IRCFactory attribute), 24
IRCFactory.send() (IRCFactory method), 25
IRCHandler._formatRaw() (IRCHandler method), 26
IRCHandler.action() (IRCHandler method), 29
IRCHandler.banlist() (IRCHandler method), 30
IRCHandler.blacklisted (IRCHandler attribute), 26
IRCHandler.closed() (IRCHandler method), 27
IRCHandler.ctcp_request() (IRCHandler method), 30
IRCHandler.ctcp_response() (IRCHandler method), 30
IRCHandler.exceptlist() (IRCHandler method), 30
IRCHandler.failed() (IRCHandler method), 27
IRCHandler.invitelist() (IRCHandler method), 30
IRCHandler.IRCHandler() (class), 26
IRCHandler.join() (IRCHandler method), 27
IRCHandler.kick() (IRCHandler method), 28
IRCHandler.list() (IRCHandler method), 31
IRCHandler.lusers() (IRCHandler method), 27
IRCHandler.mode() (IRCHandler method), 28
IRCHandler.mode_change() (IRCHandler method), 29
IRCHandler.motd() (IRCHandler method), 27
IRCHandler.names() (IRCHandler method), 28
IRCHandler.nick() (IRCHandler method), 28
IRCHandler.notice() (IRCHandler method), 29
IRCHandler.opened() (IRCHandler method), 26
IRCHandler.part() (IRCHandler method), 27
IRCHandler.privmsg() (IRCHandler method), 29
IRCHandler.quietlist() (IRCHandler method), 31
IRCHandler.quit() (IRCHandler method), 28
IRCHandler.registered() (IRCHandler method), 27
IRCHandler.topic() (IRCHandler method), 29
IRCHandler.topic_change() (IRCHandler method), 29
IRCHandler.unknown() (IRCHandler method), 30
IRCHandler.usermode() (IRCHandler method), 30
IRCHandler.who() (IRCHandler method), 28
IRCHandler.whois() (IRCHandler method), 31
IRCServer.init() (IRCServer method), 31
IRCServer.IRCServer() (class), 31
IRCServer.onConnect() (IRCServer method), 31

L

loginServerUser() (built-in function), 41

M

ModeParser.changeModes() (ModeParser method), 32
ModeParser.handleParams() (ModeParser method), 32
ModeParser.ModeParser() (class), 32
ModeParser.sortModes() (ModeParser method), 32
Module.bindFunction() (Module method), 33
Module.extend() (Module method), 33
Module.Module() (class), 33
ModuleManager.bindModule() (ModuleManager method), 34
ModuleManager.loadAllModules() (ModuleManager method), 33
ModuleManager.loadModule() (ModuleManager method), 33
ModuleManager.ModuleManager() (class), 33

N

NetworkManager.activeTab() (NetworkManager method), 35
NetworkManager.addNetwork() (NetworkManager method), 35
NetworkManager.addNetworkApi() (NetworkManager method), 35
NetworkManager.addTab() (NetworkManager method), 35
NetworkManager.changeStatus() (NetworkManager method), 36
NetworkManager.connectNetwork() (NetworkManager method), 36
NetworkManager.editNetwork() (NetworkManager method), 35
NetworkManager.editNetworkApi() (NetworkManager method), 35

NetworkManager.flags (NetworkManager attribute), 34
NetworkManager.getActiveChannelsForUser() (NetworkManager method), 34
NetworkManager.getClients() (NetworkManager method), 34
NetworkManager.getClientsForUSer() (NetworkManager method), 34
NetworkManager.init() (NetworkManager method), 34
NetworkManager.NetworkManager() (class), 34
NetworkManager.removeTab() (NetworkManager method), 36

R

RPCHandler.handleAuth() (RPCHandler method), 38
RPCHandler.handleChannelUsersAll() (RPCHandler method), 37
RPCHandler.handleCommand() (RPCHandler method), 38
RPCHandler.handleCommandsAll() (RPCHandler method), 37
RPCHandler.handleConnect() (RPCHandler method), 38
RPCHandler.handleEventsAll() (RPCHandler method), 37
RPCHandler.handleGetEvents() (RPCHandler method), 39
RPCHandler.handleInsertTab() (RPCHandler method), 39
RPCHandler.handleNetworksAll() (RPCHandler method), 37
RPCHandler.handleReadEvents() (RPCHandler method), 38
RPCHandler.handleSelectTab() (RPCHandler method), 38
RPCHandler.handleTabsAll() (RPCHandler method), 37
RPCHandler.handleUpdateTab() (RPCHandler method), 39
RPCHandler.handleUsersUpdate() (RPCHandler method), 37
RPCHandler.init() (RPCHandler method), 37
RPCHandler.onSocketOpen() (RPCHandler method), 38
RPCHandler.push() (RPCHandler method), 37
RPCHandler.RPCHandler() (class), 36

S

ServerSession.debug (ServerSession attribute), 39
ServerSession.handleEvent() (ServerSession method), 40
ServerSession.handleIrcMessage() (ServerSession method), 40
ServerSession.init() (ServerSession method), 39
ServerSession.nick() (ServerSession method), 40
ServerSession.onClientMessage() (ServerSession method), 41
ServerSession.pass() (ServerSession method), 40
ServerSession.privmsg() (ServerSession method), 41
ServerSession.quit() (ServerSession method), 40
ServerSession.sendChannelInfo() (ServerSession method), 41
ServerSession.sendJoins() (ServerSession method), 40
ServerSession.sendPlayback() (ServerSession method), 41
ServerSession.sendRaw() (ServerSession method), 41
ServerSession.sendWelcome() (ServerSession method), 40
ServerSession.setup() (ServerSession method), 40
ServerSession.user() (ServerSession method), 40

U

updateLastSeen() (built-in function), 41
UserManager.forgotPassword() (UserManager method), 43
UserManager.init() (UserManager method), 42
UserManager.isAuthenticated() (UserManager method), 42
UserManager.onUserLogin() (UserManager method), 43
UserManager.parse() (UserManager method), 43
UserManager.registerUser() (UserManager method), 42
UserManager.resetPassword() (UserManager method), 43
UserManager.timeOutInactive() (UserManager method), 42
UserManager.updatePassword() (UserManager method), 43
UserManager.updateSettings() (UserManager method), 43
UserManager.userLogin() (UserManager method), 42
UserManager.userLogout() (UserManager method), 42
UserManager.UserManager() (class), 42

W

WebSocket.bindEvents() (WebSocket method), 44
WebSocket.isValid() (WebSocket method), 44
WebSocket.onClose() (WebSocket method), 44
WebSocket.onMessage() (WebSocket method), 44
WebSocket.send() (WebSocket method), 44
WebSocket.sendBurst() (WebSocket method), 45
WebSocket.WebSocket() (class), 44