OpenParty Documentation

Release 0.2.0

Lesterpig

Oct 25, 2017

Contents

OpenParty is a web engine for **multiplayer chat games**, designed for speed and comfort. This framework is proudly powered by **Node.js** and uses **Angular.js**, **Express** and **Socket.io** modules.

OpenParty is an open-source project hosted on GitHub. We'll welcome all suggestions or pull requests (translations, new features, fixes...) !

CHAPTER 1

User documentation

Installation

OpenParty is designed to work with **Linux**, **Mac**, and even **Windows** distributions. However, the framework provides a http server, and we strongly suggest you to host OpenParty servers in Linux distributions (Debian 7 preferred).

Note: The following instructions are designed to work for a Debian 7 installation.

Softwares

The framework comes with several (small) prerequisites to work properly.

- node.js
- npm

You just have to pick-up these modules from package repositories. Binary files are available for windows.

Here are the steps for a quick installation on a bare Debian 7:

```
$ su root
# apt-get install -y curl git
# curl -sL https://deb.nodesource.com/setup_0.12 | bash -
# apt-get install -y nodejs
# exit
```

Get OpenParty

You can use **Git** repository to get the latest version, or pick a stable release.

```
$ git clone https://github.com/Lesterpig/openparty.git
$ cd openparty
$ npm install
$ cp config/config_sample.js config/config.js
```

Updates and upgrades are easily resolved by git itself:

```
$ git pull
```

Configuration

For global parameters, you just have to edit the *config/config.js* file to fit your needs.

At this point, you **don't have any gameplay definitions stored on the server**. Go to the next section to discover how to install gameplays.

> **Warning:** You'll have to restart the OpenParty server to apply the modifications.

Gameplays installation

One OpenParty web server can handle **many different gameplays** on the *same* processus. Gameplays are stored in the *data* directory (then, one subdirectory per gameplay).

```
openparty/
  config/
  data/
    gameplay1/
      definition.js
      ...
    gameplay2/
      definition.js
      ...
    ...
  docs/
  ...
```

The *data* directory is scanned at startup, and valid gameplay definitions (stored in *definition.js* files).

Example gameplays

For test purposes, we provide you several tiny gameplays.

```
$ git clone https://github.com/Lesterpig/openparty-examples data
```

Usage

Start and play !

```
$ npm start
```

You can now play with OpenParty and loaded gameplays in your **web browser**.

If you are running the server locally, you can test with *http://localhost:3040*. To stop the server, just kill it with *CTRL+C* keys.

Auto-restart on fail

You can use the **forever** module to automatically restart OpenParty on crash. More over, it's an easy way to restart your application after an upgrade. Forever will run OpenParty silently in the background.

To install it:

```
$ su root
# npm install -g forever
```

To start your application in the background:

```
$ forever start --killSignal=SIGINT app.js
```

To restart or stop it:

```
$ forever restart app.js
$ forever stop app.js
```

Use nginx as a proxy

> **Warning:** You'll need **nginx v1.3.13 or higher** to use it as a proxy for OpenParty.

Here is an example of a basic config file for nginx, assuming OpenParty is running on port 3040.

```
server {
  listen 80;

  server_name yourdomain.com;

  location / {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header X-NginX-Proxy true;
    proxy_pass http://127.0.0.1:3040/;
    proxy_redirect off;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
```

```
    }
}
```

CHAPTER 2

Developer documentation

Contributing

We welcome every contribution! Don't hesitate to submit **issues** and **pull requests**, we'll analyze each problem and patch.

General rules

- Check if anyone has the same problem as you in the **issues** before posting a new one
- Write your code in **english** with **2-spaces indentation**
- Test your patch before submit it (and not only with automated tests)
- Open one pull request per patch (to avoid mixed patchs)
- Try to rebase your commits into a single commit. We'll do it for you, but you would help us a lot :)
- Each patch sould come with its own test cases. The coverage target is **100%** !

How to contribute ?

You can do several things to help OpenParty, from minor fixes to major features. Here are some assignments you can execute to train yourself.

0. Fork the repository and build it on your computer.
1. Resolve CodeClimate issues, and open a pull request. It's often small fixes like missing semicolons or undefined variables.
2. Resolve "minor" issues on GitHub
3. Add missing test cases, and update documentation if needed
4. Ask for "major" work before starting anything that might break everything ;)

We also need **translations** of the framework. You can check this issue for more information.

Oh, you are also invited to build **gameplay definitions**; see next chapter!

Definition files

Gameplay definitions are hosted under `data/<gameplayName>/definition.js`. This file must be a **node.js module**: it means that it must return a javascript object. These two structures are valid:

```
module.exports = {

  element: "value",
  another: "bar"

}
```

```
module.exports = function() {

  this.element = "value";
  this.another = "bar";

}
```

This file is the **entry point** of your gameplay definition: you can create many javascript files in the gameplay directory if you need it! There is no restriction of size or compute time.

Note: You must provide valid keys in your definition file: it will be parsed at startup and invalid files will be rejected.

You can view a list of checked keys here.

Mandatory keys

`'name'`
String

The name of the gameplay. It will be displayed in room properties.

`'minPlayers'`
Number

The minimum/default number of players per room.

`'maxPlayers'`
Number

`'start'`
function (room, callback) {}

Called when the start command is emitted by the room creator. The callback must be called with **null** to confirm the room start. If `callback` is called with another value, a message would be printed in game chat.

Optional keys

'version'
String

The current version of this gameplay.

'opVersion'
String

The required version of OpenParty. Must be in **semver** format.

Examples:

```
"0.1.*"
">=0.1"
"<1.0.0"
```

'description'
String

A short description of the gameplay, displayed in room list.

'stages'
Object

An object containing all available stages for this gameplay. This object can be dynamically updated by your gameplay. A **stage** is just a period of time and contains only two functions: `start` and `end`.

The key used to define a stage is saved in `room.currentStage` variable, and a room is **always** in a specific stage.

'start'
function (room, cb) {}

Called by the engine when a stage is started. `cb` must be called with two parameters: the first one is an error indicator, and the second is the **duration** of the stage (seconds). A duration can be -1 for infinite.

'end'
function (room) {}

Called when a stage ends.

Example of stages object

```
stages: {
  "default": {
    start: function(room, cb) {
        cb(null, 5);
    },
    end: function(room) {}
  }
}
```

'firstStage'
String

The first stage to start.

'css'
Array[String]

You can include custom css files in the web browser. Just place your css files in a `data/<gameplayName>/css` folder and specify their names in the `css` array.

```
css: ["file1.css", "file2.css"]
```

'parameters'
Array[Object]

Used to define specific parameters for room. Players can interract with these parameters to customize their gameplay experience.

> **Warning:** This feature is currently in development.

Example of parameter:

```
{
  name: "The name of the parameter",
  type: Number, // the type
  value: 1,     // default value
  help: "An help text for this parameter"
}
```

'init'
function (room) {}

Called just after room creation.

'processMessage'
function (channel, message, player) {}

Called for each message sent by players.

You **must** return the message to broadcast it, modified or not. Return `false` to ignore the message.

'onDisconnect'
function (room, player) {}

'onReconnect'
function (room, player) {}

'reconnectDelay'
Integer

After a disconnection, a player has this delay (in second) to reconnect. Use -1 to disable reconnection. Defaults to one minute.

'sounds'
Array[Sound]

An array of Sound objects to be preloaded automatically at start. Please provide only tiny sounds to avoid large bandwidth usage.

Public API

You can access and modify these objects through your `definition.js` file.

Room

class Room()

This object is frequently an argument for `definition.js` files. You can check a full list of features in `lib/rooms.js`.

Rooms are created (and destroyed) for you by the framework. You just have to interract with it.

`Room.`**`id`**

String

A unique identifier for the room.

`Room.`**`players`**

Array[Socket]

Contains players that is in this room.

> **Warning:** This is an array of `socket` objects, not `player`! It's very important: to access the player objects, write something like this:

```
var player0 = room.players[0].player;
```

The first element of this array is the room creator.

`Room.`**`name`**

String

The current name of the room.

`Room.`**`size`**

Number

The number of available seats. If the room is started, it should be the total number of players.

`Room.`**`started`**

Boolean

If the room is started (not in waiting stage).

`Room.`**`currentStage`**

String

The name of the active stage, or `null` if not started.

`Room.`**`broadcast`**(*[channel]*, *event*[, *data*])

Arguments

- **`event`** (*`string`*) – Send an event to players browsers
- **`channel`** (*`string`*) – Send the event to this channel. If null, send to everyone in the room.
- **`data`** – Data to send through the event

Here is some examples of available events:

- `chatMessage({sender: String, message: String})`: print a new message in game log
- `clearChat()`: clear the game log
- `setGameInfo(String)`: change the content of the box in the left-top on the game-screen
- `preloadSound(Sound)`: preload a sound in player browsers to avoid further latency

- `playSound(Sound)`
- `stopSound(Sound)`

`Room.`**`playerInfo`**`(`[*channel*], *player*, *value*`)`
Send more information about a player. Client-side, it will be displayed in players list.

Arguments

- **`player`** (*`Player|Socket`*) – The player to update
- **`value`** (*`string`*) – The new value to display (html allowed)
- **`channel`** (*`string`*) – Send the event to this channel. If null, send to everyone in the room.

`Room.`**`message`**`(`[*channel*], *message*`)`
Send a chat message to players (system message)

Arguments

- **`message`** (*`string`*) – The message to send (html allowed)
- **`channel`** (*`string`*) – Send the event to this channel. If null, send to everyone in the room.

`Room.`**`nextStage`**`(`*stage*[, *callback*]`)`
End the current stage and start another one

Arguments

- **`stage`** (*`string`*) – The new stage name
- **`callback`** (*`function`*) –

`Room.`**`endStage`**`()`
End the current stage, without starting another one

`Room.`**`setStageDuration`**`(`*duration*`)`
Change current stage duration

Arguments

- **`duration`** (*`number`*) – duration in seconds

`Room.`**`getRemainingTime`**`()`

Returns Milliseconds before next stage. Can be "Infinity"

`Room.`**`resolveUsername`**`(`*username*`)`

Get player object from username

Arguments

- **`username`** (*`string`*) –

Returns Socket associated to this username, or null

Player

class `Player``()`
One player object is created and associated for each user at room startup.

`Player.`**`roles`**
`Object[Role]`

Roles of this user. **You should not modify this object directly.**

`Player.`**`channels`**
`Object[Channels]`

Subscribed channels for this user, overrides `Player.roles` ones. **You should not modify this object directly.**

`Player.`**`actions`**
`Object[Actions]`

Subscribed actions for this user, overrides `Player.roles` ones. **You should not modify this object directly.**

`Player.`**`socket`**
`Socket`

`Player.`**`room`**
`Room`

`Player.`**`username`**
`String`

`Player.`**`setRole`**(*role*, *value*)
Add, update or remove a role for a player. Actions and channels attached to the role are silently added for the player.

Arguments

- **`role`** (*`String`*) – The name of the role (should be consistent)
- **`value`** (`Role`) – Role data, or `null` to remove the role

`Player.`**`setAction`**(*name*, *value*)
Add, update or remove an action for a player

Arguments

- **`name`** (*`String`*) – The name of the action (should be consistent)
- **`value`** (`Action`) – Action data, or `null` to remove the action

`Player.`**`setChannel`**(*name*, *value*)
Add, update or remove a channel for a player

Arguments

- **`role`** (*`String`*) – The name of the channel (should be consistent)
- **`value`** (`Channel`) – Channel data, or `null` to remove the player

`Player.`**`sendAvailableActions`**()
Call this function to update one's available actions (after updating some properties for instance).

`Player.`**`emit`**(*event*, *data*)
Emit an event for one player only

`Player.`**`message`**(*m*)
Send a chat message for one player only

Action

class `Action`()
This object contains all mandatory data to build dynamic forms for players ingame.

`Action.`**`isAvailable`**
`function(player) {}`

Must return `true` if the action is available for the player.

`Action.`**`type`**
`String`

- `button`
- `select`

`Action.`**`options`**
`Object`

Contains additionnal information for specific actions.

- `submit: String` (for all): the submit message printed on the button
- `choices: String | Function | Array` (for select): the list of available choices for select actions. If the value is `players`, default choices is players' usernames.

`Action.`**`execute`**
`function(player[, choice])`

Called during action execution by a player. You don't need to check the availability, OpenParty does it for you :)

Examples:

```
var action1 = {
  isAvailable: function(player) {
    return true;
  },
  type: "button",
  options: {
    submit: "BOUM",
  },
  execute: function(player) {
    player.room.message("EVERYTHING IS EXPLODED!");
  }
};

var action2 = {
  isAvailable: function(player) {
    return true;
  },
  type: "select",
  options: {
    choices: ["One", "Two"],
    submit: "Choose",
  },
  execute: function(player, choice) {
    player.room.message(choice);
  }
};

var action3 = {
  isAvailable: function(player) {
    return player.room.currentStage === "stageA";
  },
  type: "select",
```

```
  options: {
    choices: function() { return [1,2,3]; },
    submit: "Choose",
  },
  execute: function(player, choice) {
    player.room.message("general", choice);
  }
};
```

Channel

class Channel()

A very simple object for channel management. A channel is a virtual chat room: players can read and/or speak in that channel.

By default, each player is in `general` channel (read and write accesses). You can remove this behavior by executing the following code:

```
room.players.forEach(function(p) {
  p.player.setChannel("general", null);
});
```

Each player is also in a private channel (read-only). The name of the channel is

```
player-<username>
```

with <username> replaced by the effective username of the user. This feature is just an helper for gamemaster features or private messages (for instance).

`Channel.`**r**

`Boolean`

Determines read access

`Channel.`**w**

`Boolean`

Determines write access

`Channel.`**n**

`String`

The channel name. Players will see this name on their game screens.

`Channel.`**p**

`Number`

The channel priority. Highest priority element is in the top in channels list, and selected by default. It is an optional parameter.

Example of read-only channel:

```
var channel = {r: true, w: false, n: "My Channel", p: 10};
```

Role

class Role()

A role is a combination of some **channels** and some **actions**. Because in roleplay games, some players could

share the same channels and actions...

`Role.`**`channels`**
`Object[Channel]`

`Role.`**`actiond`**
`Object[Action]`

Example:

```
var role = {

  channels: {
    "channelA": {...},
    "channelB": {...}
  },

  actions: {
    "actionA": {...},
    "actionB": {...}
  }

}
```

Sound

class `Sound`()
A sound is a minimal object containing several information for browsers.

`Sound.`**`id`**
`String`

A unique identifier for the sound.

`Sound.`**`path`**
`String`

Relative, or absolute path of the sound. You should store your sounds in **/public** directory.

`Sound.`**`distant`**
`Boolean`

Is it an absolute path, or not ? Defaults to false.

`Sound.`**`loop`**
`Boolean`

Define if the sound should be restarted at the end or not.

`Sound.`**`volume`**
`Number`

A number between 0 and 1 for setting sound volume.

Global objects

Some usefull objects are loaded as global variables by OpenParty.

`GET_RANDOM`(*from*, *to*)

Arguments

- **from** (*number*) –
- **to** (*number*) –

Returns A random integer between from (included) and to (included).

__app
The sockpress app for OpenParty. You can use it to add custom routes if required. Check the documentation.

Index

Symbols

A

C

G

P

R

S