ZombieGame Documentation

Contents

[Installation 2](#_Toc374386577)

[Getting started 3](#_Toc374386578)

[Adding static files 3](#_Toc374386579)

[Adding a controller 3](#_Toc374386580)

[Using the MySQL connector 4](#_Toc374386581)

# Installation

1. Install NodeJS  
   <http://nodejs.org/download/>  
   If the installation asks to add Node to your PATH, say yes, it’s very handy
2. Install the Github GUI tool  
   <http://windows.github.com/>
3. Clone the zombiegame repository to a local folder.
4. Open a command line terminal  
   On Windows : search for cmd  
   On Mac: search for Terminal  
   On Linux: don’t search, you’re already there
5. Navigate to your local zombiegame repository  
   On Windows: cd C:\Users\Guillaume\Desktop\zombiegame  
   On Unix: same but different path probably
6. Start the Node server by typing following command :   
   node server.js 8080  
   Where 8080 is the server port
7. Install Google Chrome, or update it if you already have it. You can visit the application at <http://localhost:8080>  
   In order to get it to emulate a mobile device, hit Ctrl+Shift+I (uppercase i) to show the DevTools bar. Open the DevTools settings by clicking the http://michal.karzynski.pl/images/illustrations/har2grinder/gear_icon.png icon. In the “Overrides” tab, activate the “Show 'Emulation' view in console drawer” option. Close the settings. If the emulation view doesn’t show up, click on any tab other than Console, and hit Esc. Now you can play with the emulation parameters.

# Getting started

The server application is written in JavaScript and built on top of NodeJS.

It uses the MySQL database to store data, with the node-mysql module to communicate. More information and examples on <https://npmjs.org/package/mysql>

The application also uses Express, which is a framework for NodeJS making development easier and faster. Some of the improvements are:

* Parsers for cookies, GET and POST parameters.
* Session handling (similar to PHP)
* Static file server (adds a HTML, CSS and image server in one line of code)
* Easier routing and basic HTTP level abstraction

Documentation is available at: <http://expressjs.com/guide.html>

## Adding static files

Whenever you want to add a static file (images, html or css or any other file) to the server/application, just add it in the public folder. Express will automatically find it and serve it online. If you put the file under public/test.jpg, the file will be available at <http://serverhost:port/test.jpg>. Be careful WHAT you put in there; all these files are truly public and there is no access control.

## Adding a controller

A controller is just a web URL (for example <http://serverhost:port/register-ajax>) which will be used by the server-side javascript to communicate with the server, but only for one specific “**action**”. (In this case, to perform registration). Because we plan to implement a lot of “actions”, we will have a lot of controllers. We use Express to help us make this easier.

Any action has a **method** (GET or POST) which is straightforward: GET to get data, POST to send data. A POST action can also get data back. A combination of method and URL is called a **route**. By adding a function to the route, we now have a controller!

Please refer to the documentation for more information: <http://expressjs.com/api.html#app.VERB>

app.get('/', function(req, res){  
 res.send('hello world');  
});

This is an example of a GET routing (see app.get). The req and res are respectively the request and response objects. If the user navigates to http://serverhost:port/, he will get a blank page containing “hello world”. Note there is no HTML or CSS formatting by default.

Because we use the controllers as API requests, we probably want to get more than just a string or a plain data back from the server. We can use serialized javascript (JSON) to pass multiple data (types) to the client and the client can easily deserialize it. (JQuery does it automatically but modern browsers also support it natively). XML is another option but is much more complex and is not the standard anymore.

To send a JSON response, we use Express’ function res.json(data)

JSON can easily return complex data structures to the client. Example: an object containing an array (list) of users and their associated data and another array with spatial data. If you just returned a string with all that data, you would have to process it with splits yourself.

## Using the MySQL connector

A global variable sqlConnection has been defined and assigned with mysql.createConnection. You can use it to make requests to the database. It’s very easy to use.

sqlConnection.query('SELECT \* FROM `users` WHERE `username`=?', 'myusername', function(err, rows) {  
 console.log('mysql result:', rows);  
});

The function query performs a query on the database. It takes 3 parameters.

The first parameter is the SQL query, you can use ? placeholders to indicate where the connector should fill in data, this is safer than putting it in there ourselves (see <http://en.wikipedia.org/wiki/SQL_injection>).

The second parameter is the data to replace the placeholders. If it’s just a scalar (string, number, Boolean, …) it replaces the placeholder as expected. If you pass an array, you can use multiple placeholders. If you pass an object, the placeholder will be replace by key-value pairs. (Example: {username: 'myusername'} will fill in `username`='myusername' at the placeholder. )

The third parameter is the function which will be called when the query is finished or failed. You can also INSERT or UPDATE data in the database, in that case the rows parameter of the callback function is just a Boolean indicating the result. For more information, please refer to the documentation.