

# Assignment JS+Node

---

In this assignment you will build the backbone of your application. In the first part of this assignment, you will add **client-side JavaScript** code to your application to make it interactive. In part two, you will write server-side code in **Node.js** and use **WebSockets** to enable clients to communicate with each other via the server.

## 0. Preliminaries

Remember that this is a group assignment! Work efficiently as a team! Both team members **must contribute to the code** and **both team members must understand the code**. The group interview will focus on having the required functionality and showing off your understanding of the code. If you have not programmed as a team before, read up on our introduction to [Visual Studio Code](#).


### Overview of deliverables and upload procedure

Task	Deliverables
1	-
2.1	Plan of action (bullet points are sufficient)
2.2	Use of design patterns (bullet points are sufficient)
2.3	Source code
3.1	-
3.2	-
3.3	WebSocket-based communication pattern between clients and server
3.4	-
3.5	Source code
4	-

Deliverables 2.1, 2.2 and 3.3 must be included in a single PDF file. The first page of this PDF must contain the names and student numbers of the two team members as well as the team name.

Submit your code in the form of a zipped folder. Make sure that your code contains the necessary **package.json** file to install/run the code, i.e. it should be sufficient to run **npm install** and **npm start** to start the server. Any specific configuration parameters your code requires should be described in an accompanying **README** file.

*Note: we expect one zipped code submission, we do **not** want one code submission per task!*

The PDF and code have to be uploaded by one of the team members to  Brightspace under **CSE Web assessment** (find the category your group belongs too) before the assessment session with the teaching assistant and before the ultimate assessment deadline. This means that the outcomes of all web technology assignments are uploaded to the same directory!

**To pass this assignment, you must have completed 2.1/2.2/3.3. Your application needs to include the required client/server components and the client-server communication has to be based on WebSockets. You pass if your app can deal with players executing the game as intended.** This means that you pass, if your app does not (yet) deal with players aborting the game in the middle or making undesired moves.

## 1. Boilerplate code

In this exercise we walk you through how to set up your game project. Once you have completed this exercise you can start coding!

Web applications tend to have specific file and folder structures, which are often generated automatically - you can consider those to be current best practices of how to set up a web application project with your chosen set of libraries and frameworks. This automatically generated code is called *boilerplate code*. In this course we will use [Express](#), a *fast, unopinionated, minimalist web framework for Node.js*. Express comes with its own [application generator tool](#).

We will use it to set up the code structure of our game application.

First of all, create a directory for this course and `cd` into it. If you are not familiar with the `cd` command, take a look at the [10 basic Linux commands you should know](#).

Then, install the necessary npm package `express-generator`. You can use one of two installation types: global or local installation. We recommend the global installation, as `express-generator` will be used from the command line, is useful for many [Node.js](#) projects and should thus be accessible from any directory (in line with [npm guidelines](#)). Packages that are only useful for the current project you are working on should be installed locally.

For a **local** installation (i.e. the package is installed in a folder in the current directory), use:

```
npm install express-generator
```

For a **global** installation of the package (the package is accessible from any directory), use:

```
npm install --global express-generator
```

In the local case (no `--global` option) the package will be installed in the current directory under `node_modules`. If in the global installation case you receive an error of insufficient access rights, you can either use `sudo npm install --global express-generator` to install the package with sufficient privileges or [change the global package installation directory to one that you can access without root access](#).

Think of a name for your application, e.g. `myapp` and then generate the boilerplate. If you have opted for the global installation of `express-generator`, run:

```
express --view=ejs myapp
```

If you have opted for the local installation of `express-generator`, run:

```
node node_modules/express-generator/bin/express-cli.js --view=ejs myapp
```

*Note: if you already created a directory for your app in VSC, let's say 'myapp', and it is empty, you can still generate the boilerplate code as explained above.*

Your terminal should show something like this:

```
create : myapp/  
create : myapp/public/  
create : myapp/public/javascripts/  
create : myapp/public/images/  
create : myapp/public/stylesheets/  
create : myapp/public/stylesheets/style.css  
create : myapp/routes/  
create : myapp/routes/index.js  
create : myapp/routes/users.js  
create : myapp/views/  
create : myapp/views/error.ejs  
create : myapp/views/index.ejs  
create : myapp/app.js  
create : myapp/package.json  
create : myapp/bin/  
create : myapp/bin/www
```

We use `ejs` as our view engine - you will hear more about it in a later lecture. When you now look at your working directory, you should see a folder `myapp` that contains the boilerplate setup. As part of it, a `package.json` file was generated, which contains a list of dependencies that your project requires. You will *not* yet find a `node_modules` folder with the npm packages the project requires in the `myapp` folder.

In order to install those, `cd` into the `myapp` directory and execute:

```
npm install
```

Now, `node_modules` is created and populated with the required packages. The Node environment will install all packages listed in `package.json`.

Place the `game.html` and `splash.html` documents created in Assignment 4 in the folder `myapp/public/`. This completes the initial setup.

Node.js has become a very popular framework for server-side programming; here is a good overview of [best practices](#). VSC comes with a good debugging support for Node.js by default as outlined in our [VSC](#)

[guide](#). If you are more of a command line person, you find [here](#) a good tutorial of how to debug Node.js applications in the terminal.

## 2. Client-side JavaScript

### 2.1)

Before you start coding, you need to have a *plan* of what needs to be done. Focus on your `game.html` page. We will deal with `splash.html` in the next assignment. **Check the required functionalities of your game listed in Assignment HTTP+Design once again.** Make a list of *all* interactive UI elements you need and their functionality.

//TODO: update demo game link Here is one example item for the [word guesser demo game](#) to help you get started:

- mouse click on a letter in the `div` element with `id=alphabet`: first, check whether the letter is still available (enabled); if so, check whether the letter occurs in the target string (the hidden word); if yes, reveal the letter, if not, remove a balloon; disable the letter; if the guessed letter was wrong, check whether the game is lost.

The plan should include all UI actions possible in the game interface. Check one more time whether you have taken care of all required functionalities.

### 2.2)

Think about the design of your JavaScript code – which aspects of your action plan can you translate into objects? It will make sense to separate the game logic from the game interface. For example, you might want to create different objects for:

- the game state;
- the game board;
- the game items (e.g. an alphabet or a dice);
- and the specific UI elements that correspond to them.

Choose at **least one of the object design patterns introduced in the lecture and implement your objects accordingly**. The basic constructor pattern is the simplest one to implement, the module pattern is more complex but preferable for code maintainability. Feel free to try more than one design pattern. This is your chance to use and learn about the introduced design patterns in more detail!

### 2.3)

Now that you have made your plan and decided on the use of the design patterns, start coding! Be mindful of the following requirements:

- Use *plain* JavaScript (i.e., no TypeScript or any other language compiling into JavaScript).
- The game has to work in two modern browsers.
- Players play the game with the mouse.
- Reduce the redundancy in the code as much as possible (improves code maintainability).
- Create as few global variables as possible (improves code maintainability).

- Achieve a separation between content and interaction: the client-side JavaScript must not be present in `game.html` (besides the obligatory script loading tags) but instead reside in the corresponding `myapp/public/javascripts` folder.

#### 👉 Hints:

- You do not have to incorporate style elements yet (CSS), we will cover styling of HTML elements in the next assignment. If despite this, you choose to incorporate CSS, be aware that we have certain requirements for CSS, so if you want to use CSS already, [check the requirements](#) to avoid duplicate work later on.
- In class we teach few ES6 features. You can, but do not have to use ES6+ features. Check <http://es6-features.org> if you are interested in what ES6 has to offer (although we are now up to ES10, ES6 brought about by far the largest number of complex language feature additions). The two features of ES6 we recommend to use at minimum are `let` and `const`; the [scoping section of the JavaScript lecture](#) explains why.
- Be mindful of the time you have for your implementation. Go for the fast solution if one is available, and move on to the next item to implement instead of being hung up on one feature for too long. For example, the [word guessing demo game](#) requires player 1 to enter a word that player 2 has to guess. There are different ways to ask the player to provide a word, the simplest is the use of `Window.prompt`. Unarguably there are visually more appealing solutions, however, this requires only one line of code, and is sufficient for our project. When you have time left, you can always go back to a feature and improve it.
- When you test your code's functionality, test it in two browsers. If it works in one, but not the other, check the browser's [Web Console](#) output; you will learn quickly whether you used a feature in your code that only one of your chosen browsers supported.
- JavaScript is a dynamic language, keep this in mind when you are debugging. In VSC, if you add `// @ts-check` to the top of your JavaScript file, you get type checking for free.
- The browser development tools are very helpful to debug client-side JavaScript. Use them.
- Do not be afraid to use place-holders (e.g. in our demo game, we start off with a fixed string to guess).
- You will have to refactor/rework your code a few times as the server-side and other client-side components are added; this is expected. For instance, for now you may want to check the validity of players' moves on the client-side (which is fine), but you may later decide to move this functionality to the server-side.
- If you are using `console.log`, familiarize yourself with the other abilities of the `Console` object as well, they are useful for client-side JavaScript debugging in the browser. `console.table` makes the output more readable. `console.assert` is good for sanity checks of your code: e.g. if you have functions that expect an array, you can add an assert statement to check whether the argument is indeed of the expected type. [Some people get angry about the usage of console.log for debugging purposes](#), as there are better debugging tools available in the web development tools of the browser and VSC. Ideally, you make use of those debugging options, however, this takes time to learn.
- You can use ESLint to help you write cleaner code. You can install it as an [extension of VSC](#), resulting in immediate feedback from the linter while coding (underlined in red). In order to use ESLint, you need to create a configuration file for the client-side JavaScript and another one for the Node.js code on the server. In the VSC extension, open `View::Command Palette` and type

Create `ESLint` configuration, once in the `myapp/public/javascript` folder for the client-side JavaScript code, and once in the root folder of the app for the Node.js code.

---

## 3. Node.js

### 3.1) Your first server

Now it is time to start your server. To get you started we walk you through the setup of a basic web server. Make sure you have completed task 1 and:

- generated the boilerplate code;
- installed the packages required to run the generated boilerplate (`npm install`);
- copied `game.html` and `splash.html` in the `myapp/public` directory.

Now, open the generated `app.js` file, delete its current content (which is an elaborate version of what we will add here) and add the following:

```
var express = require("express");
var http = require("http");

var port = process.argv[2];
var app = express();

app.use(express.static(__dirname + "/public"));
http.createServer(app).listen(port);
```

This is all the code you need for your server-side for now. Save the file, and start the server in the terminal (your current directory should be `myapp`):

```
node app.js 3000
```

Here, `3000` is the port number, you can safely use integers above 1024. Just make sure that no other application binds to the same port. Then, open your browser and use the following URLs (change the port number if you used a different one):

- `http://localhost:3000/game.html`
- `http://localhost:3000/splash.html`

If you see the two HTML files, then, congratulations, you have successfully implemented your first Node.js server!

You can also use `npm start`. This command runs an arbitrary command specified in `package.json`'s `start` property. By default this property contains `"start": "node ./bin/www"`. Replace this line by `"start": "node app.js"` in order to execute your program entry point which, in this case, is `app.js`. Since we require your submission to be runnable with `npm start`, please make sure you have followed these steps!

Instead of using the command line, you can also start your server from within VSC. Use the debug panel; a `launch.json` file will be generated for you, which needs to be updated with program specific property values. As an example, for the demo game, my `launch.json` file looks follows:

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/demo-code/balloons-
game/app.js",
      "cwd": "${workspaceFolder}/demo-code/balloons-game",
      "args": [
        "3000"
      ]
    }
  ]
}
```

### 3.2) Your first routes

In the next step, let's make our two HTML files available with modern web tech, i.e. routes:

//TODO: add example of Jekyll site setup URLs ending in `*.html` are considered old-fashioned, modern web frameworks avoid this and use *routes* instead. Add a route (i.e. `app.get("/", ...)`) so that `splash.html` is served for the URL `http://localhost:3000/`. You can make use of `res.sendFile("splash.html", {root: "./public"})`; A click on the **Play** button (or your equivalent) in the splash screen will return the `game.html` content. If you are using the HTML `<button>` element here, you can simply enclose it in an HTML `<form>` with an appropriate `action` attribute. Together with the `app.use(express.static(__dirname + "/public"))`; line in your server, this is sufficient to serve your HTML and client-side JavaScript.

In our boilerplate setup, a folder `routes` was generated; you can store your routes in files within this folder. The `demo game` shows you how to do this and how to import the routes in `app.js`. This is typically done to make the code more maintainable. You can also write out all routes directly in `app.js` (as we often do in the toy code examples presented in the lectures), however once you are working on a larger project this quickly becomes tedious.

### 3.3) WebSockets: communication between client and server

Before you are implementing the client-server communication via WebSockets, it is important to work out which entity communicates what.

For example, your game may have different types of players (in the demo game, we have two types of players, a *word creator* and a *word guesser*) and the type of player a client corresponds to, should be communicated to each client. The moves need to be communicated between clients, facilitated by the server (the **word guesser** wants to send the character guessed to the **word creator**). Who won the



game may be important for the server to log and here one player type may be responsible for communicating this to the server; and so on.

//TODO: update link to demo game Create a list of message types (e.g. game-start, game-move, player-type, abort-game, ...) and work out who (server, client-A, client-B) communicates it to whom. How many and what types of messages you need, depends on your chosen game to implement. As a concrete example, in the demo game, all message types are listed in [messages.js](#).

### 3.4) WebSockets: a minimum viable examples

Let us now connect our gamers to each other: time for the [WebScket API](#). We use [ws](#), a popular WebSocket implementations for Node.js.

First, let's install it (the `--save` option ensures that the dependency gets added to your project's `package.json` file). We assume once more that your terminal's current directory is `myapp`:

```
npm install --save ws
```

//TODO: update example link Before you implement anything for your board game, we suggest you take a look at this [example](#) of a WebSocket-based app - **it is completely independent of your board game application code**. In this minimum viable example, a client establishes a WebSocket connection with a WebSocket handshake. It sends a *Hello from the client* message to the server, which responds with a *Hello to you too!* and logs the client's message. WebSocket programming thus requires both changes in the client-side and server-side code.

### 3.5) WebSockets: implementing your game

Having seen and run the minimal WebSocket example, it is now time to implement client-server communication in your game application via WebSockets. Here is what we want to achieve:

- Upon clicking the *Play* button, the user will enter the **game screen** and wait for a sufficient number of other gamers to start playing. It is clear for the player that s/he is waiting for more players to enter the game.
- Once there are sufficiently many players, the game automatically starts and the players play against each other. Multiple games can take place at the same time. Some games require a setup phase which may differ between players (e.g., in [Mastermind](#) one player is the codemaker and one is the codebreaker).
- Once a player drops out of an ongoing game, the other players are alerted and the game is aborted.
- Once a game is finished, the winner(s) and loser(s) receive a status update.
- The server keeps track of certain game statistics (pick three statistics you want to report).
- The players communicate with each other (i.e. their moves) via WebSockets.

---

#### 👉 Hints:

- You may want to share code between the client and server - the message types described above are a good candidate for code you do not want to duplicate. The solution is a particular design



pattern, [described here](#). If you add for instance `messages.js` to `myapp/public/javascripts` following that particular design pattern, you can add the following line to `app.js` to access it in your Node.js script as well: `var messages = require("../public/javascripts/messages");`.

- The server has to keep track of all ongoing games in order to facilitate the broadcasting of messages to the correct clients. There are many ways to do this, one option is to create a `game` object that keeps track of the WebSocket objects belonging to the game's players. Each WebSocket object receives an `id` and a `Map` object with WebSocket `id` as key and `game` object as value to ensure that the server can determine quickly for which WebSockets the received messages are meant.
- The game status on the server can be implemented as an in-memory object; we do not require you to store the game status in a database or back it up to file.

---

## 4. Cleaning up

Ensure that your app works as intended in two major browsers.

We suggest you check the ESLint (or any other linter) output: it should help you to spot easy-to-make mistakes, which in turn should help you write better code.

Ensure that your code contains the necessary `package.json` file content to install/run the code, i.e. it should be sufficient to take your `myapp` folder, and run `npm install` and `npm start` to start the server. Any specific configuration parameters your code requires, should be described in an accompanying `README` file.