Software Engineering (C-T-303-HUGB), Week 08 Fall Semester 2018

Javascript and dependecies

Main purpose of this exercise is for students to get comfortable in the environment we will be working in over the next weeks. We are focused on running things "bare-boned" (i.e. no fancy IDEs).

Javascript is an interpreted language, not a compiled one such as Java or C#. So for today's exercise there is no need to setup any compiler, but to manage dependencies that we will use (additional libraries) we will use NPM (https://www.npmjs.com/) (Node (https://nodejs.org/en/) package manager). In addition to install NPM (https://www.npmjs.com/), Node (https://nodejs.org/en/) also gives us a Javascript runtime, so we can run our code outside of a browser.

Setup Node.js

1. Download the Node.js (https://nodejs.org/en/) installer and run it.
2. Verify everything is working in Git Bash, by typing `node --version`
    - You should see either "v8.12.0" or "v10.11.0"
3. Check for NPM (https://www.npmjs.com/), by typing `npm --version`

Part 01: Getting started with Javascript

1. Using an editor of your choice (not an IDE), create a simple Javascript Hello World script (*greeting.js*)

```
// greeting.js
function greeting() {
    console.log("Hello, World!");
}

greeting();
```

2. Run it using Node (https://nodejs.org/en/): `node greeting.js`
3. Once you have this initial structure setup, it's time to create a repository for this application and pushing it to Github. See last week's lab exercise for instructions if you need to jolt your memory.

Part 02: Creating a module and using it

In order to use our greeting function elsewhere, we need to turn it into a *module*. Do the following:

1. Change the function to accept a name as an input.

```
function greeting(name) {
    ...
}
```

2. Change the output of the function so that it returns a string with a greeting using the name given.

```
...
    return "Hello, " + name + "!";
...
```

3. Instead of calling the function at the end of the execution, we want to *export* it, so that the function can be used elsewhere.
    - We do this by adding `module.exports = greeting;` to the end of the file,
    - and remove the line where we call the function `greeting();`.
4. Now create a new file, `app.js`, that imports the function `greeting` and calls it with some name. After creating the file, you should now have two files in the folder, `greeting.js` and `app.js`.

```
// app.js
const greeting = require('./greeting');

console.log(greeting('Totoro'));
```

5. When you run `node app` or `node app.js` (Node assumes `.js` by default), you should see the message: *Hello, Totoro!* and you have created a nice modular module and use the module in another file, congratulations!
6. Make sure your changes are commited in your git repository. Try to add a tag as well.

This pattern of exporting a function, or a set of functions and variables from a file and *requiring* them in others is an essential step in helping us avoid bundling together different functionality and mixing together logic that doesn't belong together. And we will be doing plenty of it.

Part 03: Introduce a dependency - Jest (testing framework)

Using NPM (https://www.npmjs.com/), let's add the Jest (https://jestjs.io) testing framework. In order to do so, we must initialize our repository with `npm init`. This step will ask several questions about your project and finally output a `package.json` file. This file holds information about your project, such as name, version number, author, and *dependencies*. Dependencies are *packages* that the project requires to run.

1. Run `npm init` in your project directory.
    - Make sure you specify `jest` as the test command (if you don't this can be changed in the `package.json` file later).
2. Install Jest as a development dependency, `npm install --save-dev jest`
3. Add a new file, `greeting.test.js`.

```
// greeting.test.js
const greeting = require('./greeting');

test("returns greeting with custom name", () => {
  expect(greeting("Bei")).toBe("Hello, Mei!");
});
```

4. Run the test!
    - `npm test`
    - Bonus step: If you want your tests to run every time a change is made, Jest (https://jestjs.io) offers a `--watchAll` parameter, that will monitor your project for changes and run all tests when a file is saved. To be able to do this you can either:
        - Add Jest globally on your computer (`npm install --global jest`), and run `jest --watchAll`.
        - Or, add a custom script into your `package.json` file, called `watch` and run it with `npm run watch`. Try to figure out how you add a custom script into your `package.json` file.
5. You should have a failing test. Make changes to the test so that is passes, and try again.

NPM basic commands

- `npm init` initializes an NPM package in the current directory.
- `npm install` will look for an existing `package.json` file and install all dependencies specified in the file.
- `npm install --save NAME_OF_PACKAGE` installs a NPM package and adds it as a dependency to the existing packages.json file in the current directory.
- `npm install --save-dev NAME_OF_PACKAGE` installs a NPM package and adds it as a development dependency to the existing packages.json file in the current directory.

- `npm install NAME_OF_PACKAGE` installs a npm package locally, without saving it to the project.
- `npm install --global NAME_OF_PACKAGE` installs a npm package globally on your computer, without saving it to the project.

Part 04: Project structure

Let's clean the project up a bit, following some good Javascript practises.

1. Create the following folder structure:

```
Week08
  |
  | - app.js
  | - node_modules/
  |   | ... (all the NPM packages)
  | - package.json
  | - src/
  |   | - greeting.js
  |   | - greeting.test.js
  |
```

2. Note, node_modules should never be a part of your git repository, please make sure you have ignored that folder using a .gitignore file.
3. Refactor app.js to import the greeting function from the `./src/greeting` path.
4. Make sure everything works;
    - `node app`
    - `npm test`
5. If it works, make a git commit.

Part 05: Custom build scripts

With the more complicated structure and when adding more dependencies, compiling and running our Project will grow more and more complex. We can add build commands to make our lives easier. When developing an application using NPM, we can add *scripts* to our package.json file.

NPM has standard commands, like `init`, `install`, `start`, `test` and more, these can be left as is, or overwritten if needed. Notice that we have overwritten the `test` command already, as we wanted to use the Jest (https://jestjs.io) testing framework.

Create two scripts in the package.json file: `clean` and `run`.

Clean

The clean script should remove the `node_modules` folder.

```
{
  "name": "hugb-week08-2018",
  ...
  "main": "app.js",
  "scripts": {
    "test": "jest",
    "watch": "jest --watchAll",
    "clean": "rm -rf node_modules"
  },
  ...
  "devDependencies": {
    "jest": "^23.6.0"
  }
}
```

Run

The run file should run the application once, this is left as an exercise for you.