Computer Science Department
San Francisco State University
CSC 667/867
Fall 2023

Term Project
Milestone 2: Setting up your Development Environment

Due Date
Next week, ***BEFORE CLASS*** (we will be using class time to review)

Grade
Counts towards overall term project grade.

Overview
This milestone will walk students through the process of setting up their development environments for the term project.

Submission
Each student should go through this process to gain familiarity with the different pieces of the application you will be working on throughout the semester. Only one student from your team needs to submit their code to the repository (you will be working on the same codebase after this milestone). Your GitHub repository must be created within our GitHub classroom: https://classroom.github.com/a/MYn2oy0-.

Setting up our Application's Directory Structure
Create the directory that will hold all of the code for the project, and change into that directory. Initialize the directory as a git repository, and initialize a new npm project:

```
mkdir team-alpha-uno
cd team-alpha-uno
git init .
npm init -y
```

(Feel free to update the description, author, and license fields in the resulting package.json file.)

Next, create directories to distinguish between our front-end javascript (the code that will be delivered to the client to be able to implement things like chat functionality, and dynamic page updates) and our back-end javascript (the code that will be used to respond to the client HTTP requests).

```
mkdir frontend
mkdir backend
```

Create the file that will hold the code that will be the starting point for the set up and execution of our server:

```
touch backend/server.js
```

Finally, we want to ensure that we do not commit certain files into our repository (`node_modules/` because they can be installed using `npm install`, and `.env` because it will hold strings that we do not want to make public, and that will be used only for local development):

```
touch .gitignore
```

Add the following contents to the `.gitignore` file:

```
node_modules/*
.env
```

Create a basic Express server

We will be using https://expressjs.com to implement the back end code for the term project - the code that will act as the server for all requests to the web application. Express provides an API that allows us to write javascript code to respond to HTTP requests, as well as providing the ability to host static files, and to dynamically generate HTML response to send to the client. We will be discussing *how* to do this as we continue to work on our projects, but for now, we need to include the express dependency in our project:

```
npm install express
```

A minimal server can be created by adding the following code to `backend/server.js`:

```
const express = require("express");
const app = express();
const PORT = process.env.PORT || 3000;

app.get("/", (request, response) => {
  response.send("Hello World!");
});

app.listen(PORT, () => {
  console.log(`Server started on port ${PORT}`);
});
```

This code requires (another way of saying includes or imports) the express function (that is exported from the express package) into `server.js`, then uses the function to create an instance of an express application. (See https://expressjs.com/en/5x/api.html#app for the express application API.)

We then define a “route” - this specifies an HTTP verb (in this case, get), and a URL (in this case, the root of our site: "/") that the express server will monitor for requests. When an HTTP request is received that matches this verb and URL, the express application will invoke the “handler” function we define, passing in `request` and `response` objects that we can use to create a response to the client. In this case, we use the `response` object’s send method to respond with the text, “Hello World!”.

We can run this using the node javascript runtime from the root of our project directory:

```
node backend/server.js
```

We can verify this is working by visiting http://localhost:3000, where we should see the text that we used the `response` object to send to the client (“Hello World!”).

Organizing the server code

Eventually, we will be adding quite a few additional routes to the server, and we want to avoid creating a single, massive, hard-to-maintain server file. One tool that Express provides us with is the `Router` ”middleware” (more on this later) that allows us to create individual routes in modules (separate files), and then “mount” those routes in our main application instance.

Create a directory where our route logic will go, and an initial route file for our root routes:

```
mkdir backend/routes
touch backend/routes/root.js
```

As we add additional functionality like authentication and authorization, game logic, chat logic, etc., we can break up the routes into easy-to-understand and easy-to-maintain files in this new directory.

Add the following code to import the express `Route` middleware, create the route we previously defined in `server.js`, and then export that route so that another module can import (require) it:

```
const express = require("express");
const router = express.Router();

router.get("/", (request, response) => {
  response.send("Hello world from within a route!");
})
```

```
module.exports = router;
```

Now, we can refactor the code in server.js to include and use this route:

```
const express = require("express");
const app = express();
const PORT = process.env.PORT || 3000;

const rootRoutes = require("./backend/routes/root");

app.use("/", rootRoutes);

app.listen(PORT, () => {
  console.log(`Server started on port ${PORT}`);
});
```

This imports the routes that were exported from the `root.js` file, and then “mounts” all of the routes defined in that `Router` under the "/" URL (appends the URLs from the route to the root URL).

If you left the server running from the previous section and refresh the http://localhost:3000 page, you won’t see a change! In order for node to load the change, it needs to reload the file. To do this stop the server (Ctrl-C on a *nix system), and start it again using the `node server.js` command. (This is tedious, and we will work on a way to automate this shortly.) Visiting http://localhost:3000 should now show us the updated content.

Simplify Application Startup

We can use the `package.json` file to help simplify start up by adding a “script” entry that runs the startup command for us. (We will be adding additional scripts here as we continue development.) In `package.json`, in the `scripts` section, add the following entry:

```
"start": "node ./server.js”
```

Now we can use the following command to start the server:

```
npm run start
```

This runs the `node server.js` command, starting the server as we have seen in previous steps.

Now, we can add another `scripts` entry to help reload the server as we make changes, so that we do not have to manually stop and start the server whenever a file is updated in order to see that change reflected in our site. To do this, we will use another library, `nodemon`, that will automatically restart the node process whenever a change is detected in a directory.

First, install the `nodemon` dependency:

```
npm install nodemon –save-dev
```

Note that we will only be using `nodemon` in our development environments - we do not want the server to reload in production if a file changes! For this reason, we use the `--save-dev` flag when installing nodemon to tell npm that the module should only be installed in a development environment.

Now, we can add a new script to package.json:

```
"start:dev": “nodemon ./backend/server.js”
```

We create this separate script so that we can add developer tools when we are in a development environment, and so that we can omit those tools in the production environment (that's why we keep the `start` script).

Now start the server:

```
npm run start:dev
```

We don't see anything different since we haven't made any changes, but now we can make a change to our route file (for example, updating the text), and just press the refresh button in the browser to see those changes reflected (without having to stop and start the server). If you look at the shell, you will see that `nodemon` causes the server to reload when it detects the file change.

Adding Error Handling to the Server

With the current implementation, if we try to visit a URL (route) that the express application does not understand, it responds with a simple text message. For example, with your server running, browse to http://localhost:3000/nothing. You will see the message:

```
Cannot GET /nothing
```

In order to create a better and more consistent user experience when an error is encountered, we will use the `http-errors` library:

```
npm install http-errors
```

Add the following code to the `server.js` file by importing the library at the top of the file, and then adding a middleware function at the bottom of the file. Adding this function at the bottom of the file is important - express attempts to match routes in the order they are defined, so if a valid route is defined, we want Express to use that route logic instead of executing this function. Placing the function at the bottom of the file means it should only be reached (and therefore executed) if the requested URL does not get matched to any route.

```
const createError = require("http-errors");

/** Existing server.js content **/

app.use((request, response, next) => {
  next(createError(404));
});
```

Visiting the page, we now see a very noisy (and informative) error message being displayed.

So, What is Middleware?

Middleware is a fancy term for "a function that will always be executed by the express application" (following the order rules discussed above). These functions have access to the `request` and `response` objects, as well as a special function named `next` that tells Express to execute the next middleware function in a chain of middleware functions.

The routes that we've seen to this point are just a special type of middleware - functions that have access to the `request` and `response` objects, and that only execute for a *specific path (or URL)*. The routes we have written also end the request-response cycle, so they do not need to use the `next` function to continue executing middleware; after we respond to a client request, there's nothing else we want to do.

Since we will likely be writing additional middleware to support our project, let's create a `middleware` directory in our `backend` folder, and create a simple middleware to exercise our understanding:

```
mkdir backend/middleware
touch backend/middleware/request-time.js
```

In `request-time.js`, add the code:

```
const requestTime = (request, response, next) => {
  console.log(
    `Request received at ${Date.now()}: ${request.method}`
  );

  next();
};

module.exports = requestTime;
```

Now, tell the express application to use this middleware in `server.js` (remember that the placement of this is important!):

```
const createError = require("http-errors");
const requestTime = require("./backend/middleware/request-time");

const express = require("express");
const app = express();
app.use(requestTime);

const PORT = process.env.PORT || 3000;

/* Rest of server.js */
```

Since this middleware was placed before all other middleware (including routes and the error handler), it always gets executed, printing out a timestamp and the HTTP verb used in the HTTP request sent to the express application.

Now that we know a little more about middleware, we can remove this from `server.js`.

Serving Static Files

Sometimes, we want to serve a static file - one that is not created by javascript logic in a route (images, stylesheets, etc.). To configure our express application to do this, create a directory to hold our static assets:

```
mkdir backend/static
```

Now, set up this directory so that Express knows to serve static files from it. In `server.js`, add the following:

```
const path = require("path");
const createError = require("http-errors");

const express = require("express");
const app = express();

const PORT = process.env.PORT || 3000;

app.use(express.static(path.join(__dirname, "backend", "static")));

const rootRoutes = require("./backend/routes/root");

/* Rest of server.js */
```

To test this functionality, I added a `favicon.ico` file in the `static` directory, and used chrome (which annoyingly always asks for a favicon file, but its useful in this instance). When I refresh the root URL of my application, I can see in the network tab that the favicon is being returned, and that the route is still loading.

Creating Content

We can use `response.send()` to send html content to our client, for example:

```
  router.get("/", (request, response) => {
    const name = "person";

    response.send(
      `<html><head><title>Hello</title><body><p>Hello ${name} html!</
p></html>`
    );
  });
```

As you can imagine, it could get really tedious generating these HTML strings by hand, especially when we want to insert dynamic information into the HTML (like the variable `name`, above). Express allows us to integrate a "template engine" that removes the tedium from dynamically generating HTML. A templating engine allows us to specify a template, provide it with values we want to plug into that template (sometimes called "locals"), and then use the template to generate the HTML.

There are a few different template engines supported by Express. I prefer `ejs`, so that is what I will use in this example. Feel free to research different template engines that are supported by Express if `ejs` isn't your cup of tea: https://expressjs.com/en/resources/template-engines.html.

First, install the template engine library:

```
  npm install ejs
```

In `server.js`, tell the express application to use the template engine (doing this immediately after the application instance is created so that the engine is available for use in any route), and *where* to find the templates:

```
  const app = express();

  app.set("views", path.join(__dirname, "backend", "views"));
  app.set("view engine", "pug");
  app.use(express.static(path.join(__dirname, "backend", "static")));
```

Create the directory to store our templates (also called views), and our first template file:

```
  mkdir backend/views
  touch backend/views/root.ejs
```

In `root.js`, add the following code:

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello <%= name %></p>
  </body>
</html>
```

Now we can update the `views/root.ejs` file to take advantage of this template:

```
router.get("/", (_request, response) => {
  const name = "My name";

  response.render('root', { name });
});
```

Switching `response.send` to `response.render` tells the express application to find a template, in this case, `root`, and to provide the "locals" object containing the `name` key and value to be used to fill out the template. Refresh the root page of our app, and you should see an HTML page with the dynamic content that was specified. Updating the variable values in the route will update the information displayed in the HTML.

Setting up the Frontend

Eventually, we will be adding code specifically to handle the front-end logic for our game (this will go in the `frontend` directory). In our front-end code, we want to be able to utilize modern javascript, and to be able to organize our code into discrete modules (individual files) to make it easier to maintain. Unfortunately, the browser is not able to find this code unless we manually insert `<link>` and `<script>` tags for all of the different files.

To avoid this manual process, we will use a modern build tool, `webpack` in order to aggregate all of our files into a single file, and place that file in the `static/` folder we created earlier. This way, we only have to add one `<script>` tag to our site, and that will include all of the front-end code we have written.

We need a few libraries to do this:

```
npm install --save-dev webpack webpack-cli babel-loader
```

Webpack can be run manually, passing in configuration flags, or we can specify configuration in a special file named `webpack.config.js` at the root of our project:

```
touch webpack.config.js
```

In this file, we will define the configuration that the `webpack` tool will use to determine how to build our javascript, where to find the javascript we want built, and where to place the final build product. Add this code to the `webpack.config.js` file:

```
const path = require("path");

module.exports = {
  entry: "./frontend/index.js",
  output: {
    path: path.join(__dirname, "backend", "static", "scripts"),
    publicPath: "/backend/static/scripts",
    filename: "bundle.js",
  },
  mode: "production",
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: { loader: "babel-loader" },
      },
    ],
  },
};
```

Here, we're telling `webpack` to start at `./frontend/index.js`, and to create a bundle of all of the javascript referenced from that file and all files it references (i.e. traversing the dependency tree of `import` statements, and ensuring that all code that is imported is added to resulting build artifact). The `module rules` section tells `webpack` to handle any file that ends with the extension `.js`. (We can use additional loaders with different extensions; for example, if we wanted to use typescript - a strongly typed superset of javascript - we could test for the `.ts` extension, and use an appropriate loader to convert that into javascript that the browser could understand.) Finally, we tell `webpack` to output the resulting file in our `backend/static/js` directory, naming it `bundle.js`.

With this configuration done, we will add another script into the `scripts` section of our `package.json` file in order to automate this build process. (The `build` script will be used to

ensure a new package is created for production every time we deploy, and the `build:dev` script will be used while we are developing, to watch for changes, and re-run `webpack` when a change is detected.)

```
{
  "scripts": {
    "build": "webpack",
    "build:dev": "webpack --watch"
  }
}
```

Let's add the entry file into the `frontend` directory:

```
touch frontend/index.js
```

And add some basic code so we can check that this works as expected:

```
console.log("Hello from a bundled asset.");
```

At the command line, use the `build` script:

```
npm run build
```

A new folder named `js` will be created in `/backend/static` containing the file `bundle.js`.

We will include this in the `root.ejs` file we created earlier by adding the following `script` tag in the `head` section:

```
<script src="/js/bundle.js">
```

Make sure your server is running (`npm run start:dev`), and point your browser at http://localhost:3000. Check the console in the browser, and you should now see the `console.log` statement we added in the `frontend/index.js` file.

Development is Hard

With our current scripts, we would have to run both `npm run start:dev` and `npm run build:dev` in order to have our front-end code changes automatically bundled, and to have our back-end changes cause the server to reload. In addition, when either of these changes happen, we have to manually reload the browser. Let's make it easier to manage all of this in our development environment with a few more changes to our project.

First, let's add some libraries to help:

```
npm install --save-dev livereload connect-livereload
```

We need to integrate these with our server, but we only want this to happen while we're in the development environment. Change the `start:dev` script to add an environment variable named `NODE_ENV`:

```
{
  "scripts": {
    "start:dev":
      "NODE_ENV=development nodemon --watch backend ./server.js"
  }
}
```

We will use this environment variable in our code to only run development code if this variable's value is `development` (we will set the value to `production` when we look at how to deploy our application). To use this environment variable, update `server.js` to include the following, right after we create the app instance:

```
if (process.env.NODE_ENV === "development") {
  const livereload = require("livereload");
  const connectLiveReload = require("connect-livereload");

  const liveReloadServer = livereload.createServer();
  liveReloadServer.watch(path.join(__dirname, "backend", "static"));
  liveReloadServer.server.once("connection", () => {
    setTimeout(() => {
      liveReloadServer.refresh("/");
    }, 100);
  });

  app.use(connectLiveReload());
}
```

We also want to reload the server whenever we make a change to an `.ejs` file, and since `livereload` is monitoring the static folder, we can omit that from `nodemon`'s set of files. Create a configuration file for `nodemon`:

```
touch nodemon.json
```

```
```

In this file, add the following:

```
{
  "ext": "js,hbs",
  "ignore": [
    "backend/static"
  ]
}
```

Now we want to run *both* the `nodemon` process and the `webpack` process `concurrently`:

```
npm install --save-dev concurrently
```

We will change the `start:dev` script to use `concurrently`, and move the existing `start:dev` script to `server:dev` in `package.json`:

```
{
  "scripts": {
    "start:dev":
      "concurrently \"npm:server:dev\" \"npm:build:dev\"",
    "server:dev": "NODE_ENV=development nodemon ./backend/server.js”
  }
}
```

Final Touches

To facilitate development (and eventually debugging), we will add a logging library called `morgan` to our application. Install the dependency:

```
npm install morgan
```

And tell the express application about it in `server.js` (immediately after the application object is created):

```
const express = require("express");
const app = express();

app.use(morgan(“dev"));
```

Additional information will now be displayed in the shell as requests are made:

```
[nodemon] starting `node ./server.js`
Server started on port 3000
GET / 304 175.073 ms - -
GET /stylesheets/home.css 304 1.138 ms - -
GET /favicon.ico 304 0.500 ms - -
```

We want to add some additional utility to our server, namely the ability to support url encoded request bodies, json request bodies, and cookies. Supporting cookies requires a library, `cookie-parser`:

```
npm install cookie-parser
```

JSON and url encoded bodies are supported by express, with some minor setup. To setup all three of these, I made some additions to `server.js`, with the final version of my file looking like this:

```
const path = require("path");
const createError = require("http-errors");
const cookieParser = require("cookie-parser");
const morgan = require("morgan");

const express = require("express");
const app = express();

app.use(morgan("dev"));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.set("views", path.join(__dirname, "backend", "views"));
app.set("view engine", "pug");
app.use(express.static(path.join(__dirname, "backend", "static")));

const rootRoutes = require("./backend/routes/root");
app.use("/", rootRoutes);

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server started on port ${PORT}`);
```

```
  });

  app.use((request, response, next) => {
    next(createError(404));
  });
```

Keeping Code Consistent

It can be difficult to keep code consistent when working on a team of multiple developers. For that reason, we are going to automate code formatting, and use some tools to apply the code formatting every time we commit a change. First install the libraries that will manage this for us (since we only use these during development, we are using the `--save-dev` flag):

```
  npm install --save-dev husky lint-staged
  npm pkg set scripts.prepare="husky install"
  npm run prepare
  npx husky add ./.husky/pre-commit "npx lint-staged"
  # Make sure the hook is included in your repo
  git add .husky/pre-commit
```

Then, add the following to `package.json` (the outer braces in this snippet are the outer braces in `package.json`):

```
  {
    "lint-staged": {
      "**/*": "prettier --write --ignore-unknown"
    }
  }
```

Now that we have some automation happening on every commit, we can also use it to ensure that our frontend code gets built for the production environment. One way to do this is to just have webpack run on every commit. We can update the `lint-staged` entry in `package.json` to do this for us:

```
{
  "lint-staged": {
    "**/*": "prettier --write --ignore-unknown",
    "frontend/**/*.js": "webpack"
  }
}
```