Contents

Lab 9: NodeJS and Express

> This is what we'll be building.

In this lab, you'll construct a simple web application. You'll use the Express framework to create a web server, and you'll use `ejs` templates to create templated web pages.

A Quick Recap

Last week, we created some simple JavaScript programs. We ran these programs in the NodeJS environment. For instance, the following program prints "Hello World" to the console:

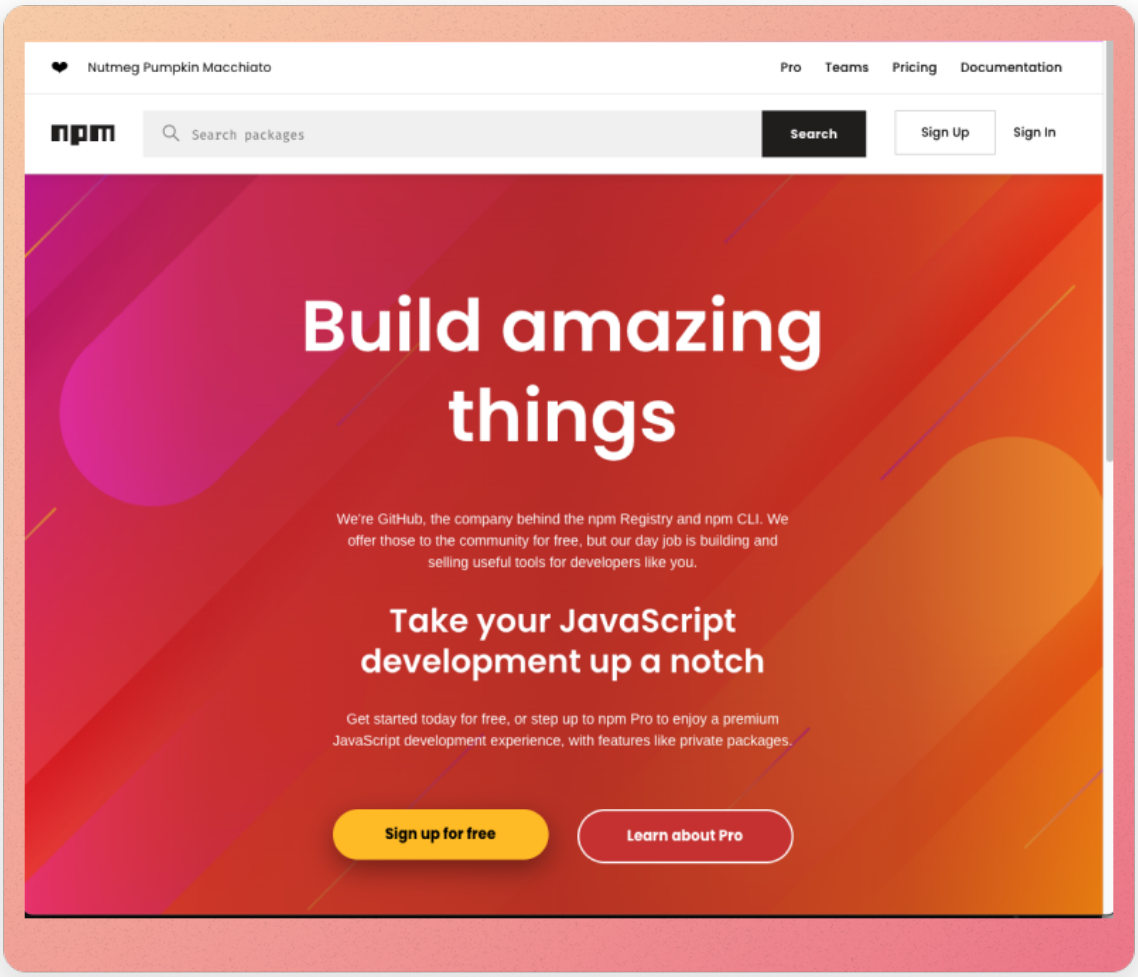
```
1 console.log("Hello World");
```

> example of a simple program called `hello_world.js`

We can run this program by typing `node hello_world.js` in the terminal.

0.0 Upping our NodeJS Development Game

Package Management and the Package.json file



> The NPM package manager website (https://www.npmjs.com/)

NodeJS has a package manager called `npm`. It is similar to `pip` in Python or Maven in Java. We can use `npm` to install packages that we can use in our programs.

One of the reasons I love JavaScript is it has a rich package ecosystem. We can think of packages as open-source, free bolt-ons to our programs. At the time of writing, there are over 1.3 million packages. You can explore packages by visiting the NPM repository.

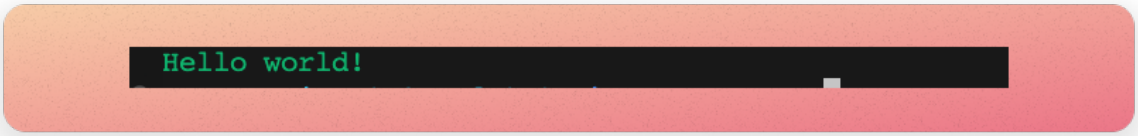
This week, we are going to use the packages (`express`, `ejs`, `chalkjs` and `nodemon`) to build a simple web application. We'll learn more about these package, and how to include them in our projects,

as we go along.

Exercise 0.1 : Getting Started

1. As always, start and connect to your Azure Labs' virtual machine (VM) by visiting this link: https://labs.azure.com/virtualmachines.
2. Connect to the VM:
    1. Toggle the button to start – it might take a while.
    2. Once it changes to Running, click on the monitor icon.
    3. A file will be downloaded – click on it to run it, and you will be prompted to enter the password you created last time. IMPORTANT: the username must be `labuser` (remove the ~/).
    4. Remember to click on the following icon to make the window resize appropriately
3. Within your VM, create a folder in your `C:\code` directory called `lab_9`.
4. Open VS Code and open the `lab_9` folder.

Exercise 0.2 : Creating a Package.json file



> The hello world program running in the VS Code terminal. Notice how it's green because we are using the `chalk` package.

1. Open the terminal in VS Code.
2. Type `npm init` and press enter. You'll see some prompts keep pressing enter to use the defaults.
3. You should now have a **`package.json`** file in your `lab_9` folder. This file contains information about your project and the packages that it depends on. It also contains a list of scripts that you can run. We'll look at this in more detail later.

Exercise 0.3 : Installing Packages

1. We are going to install the `chalk` package.

2. In the terminal ensure it is pointing to the root of the `lab_9` folder, type `npm install chalk@4` and press enter. This command installs the `chalk` package and saves it as a dependency in the **`package.json`** file. The `@4` part of the command tells `npm` to install version 4 of the `chalk` package. This is important because packages are updated regularly and we want to make sure that we are all using the same version.
3. If all has gone well you should see that a `node_modules` folder has been created. This folder contains the `chalk` package and any other packages that it depends on. You should also see that the **`package.json`** file has been updated to include `chalk` in the list of dependencies. Finally, you should see a **`package`**`-lock.json` file has been created.
4. We will now use chalk. In the `lab_9` folder, create a file called `exercise_0_3.js`. Add the following code to the file:

```
const chalk = require("chalk"); // Import the chalk package from the
    node modules folder, notice how we don't need to use an absolute
    path.

console.log(chalk.blue("Hello world!")); // Use the chalk package to
    print a blue message to the console.
```

5. In the terminal, type `node index.js` and press enter. You should see a blue message printed to the console. Try and change the message to green!

Click Here to See the Solution

Exercise 0.4: Installing Nodemon

Nodemon is a package that we can use to automatically restart our NodeJS programs when we make changes to them. This is useful because it means we don't have to keep restarting our programs manually.

Let's create a webserver and try and understand why we need to install Nodemon.

1. Create a file called `exercise_0_4.js` in the `lab_9` folder. Add the following code to the file:

```
const http = require('http'); // Import the http package, this is a core NodeJS package so it doesn't need to be installed.
const server = http.createServer(); // Create a server object.
const chalk = require('chalk'); // Import the chalk package from the node modules folder; we installed this in the previous exercise.

server.on('request', (request, response) => { // Listen for requests and respond with a message.
    response.end("hello world"));
});

server.listen(8080, function () {
      console.log(chalk.blue('Server is listening on port 8080')); // Print a message to the console when the server starts.
});
```

Do not worry if the code above seems somewhat alien to you; much of it may be new. In short, we are setting up web server that is listening on local host port 8080. When a client connects, we simply return hello world.

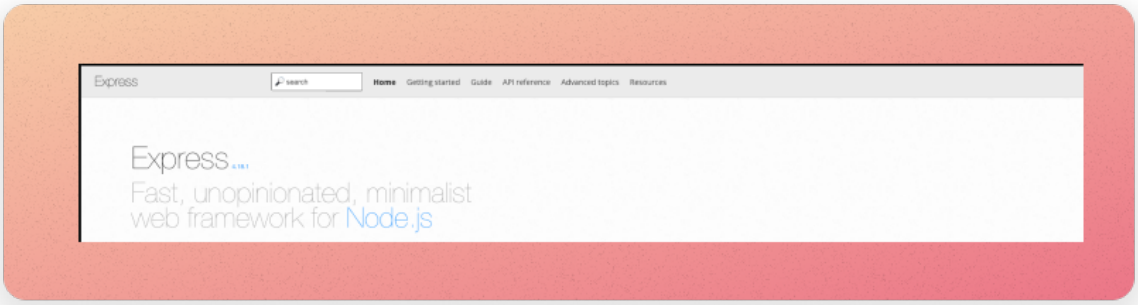
1. In the terminal, type `node exercise_0_4.js` and press enter. You should see a message printed to the console saying that the server is listening on port 8080. This means that the server is running and waiting for requests. If you open a web browser and navigate to `http://localhost:8080` you should see a message saying "hello world".
2. Notice how, in the terminal, your program is still running. Make a change to your code, for instance update "hello world" to "Hello, World!". If you refresh or revisit the localhost URL, you will see that the change has not been applied. This is because the server is not restarting when we make changes to the code. We have to stop the server and restart it manually. This is where Nodemon comes in.
3. To install Nodemon, stop the existing program from running: press `ctrl + c` in the terminal. Then type `npm install -D nodemon` and press enter. The `-D` flag tells `npm` to install the package as a development dependency. If you check the **`package.json`** file you should see that Nodemon has been added to the list of development dependencies.
4. Next, we need to update the **`package.json`** file to tell it to use Nodemon to run our program. Open the **`package.json`** file and add the following line to the `scripts` section (there is already a `test` script in there; you only need to add the `start` script):

```
 "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon exercise_0_4.js"
  },
```

7. In the terminal, type `npm run start` and press enter. You should see a message printed to the console saying that the server is listening on port 8080. Now, ff you make a change to the code and refresh the page, you should see that the change has been applied. This is because Nodemon is automatically restarting the server when we make changes to the code.

Click Here to See the Solution

1.0 Introduction to Express



> The Express website (https://expressjs.com/)

Express is a NodeJS package that we can use to build web applications. It provides a set of tools that make it easier to handle requests and responses.

According to the the express documentation, “express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications”. In other words, it provides a lightweight framework to assist in the development of web applications.

Exercise 1.0 : Installing Express

1. In the VS Code terminal window, type `npm install express` and press enter.
2. Next, create the file `index.js` this will be the entry point to our application.
3. Add the following code:

```
1 const express = require("express");
2 const app = express();
3 const port = 8000;
4
5 app.get("/", (req, res) => {
6   res.send("Hello, World!");
7 });
```

```
8
9  app.listen(port, () => {
10   console.log(`Example app listening at http://localhost:${port}`);
11 });
```

The above example uses the express package to create a new web-server that listens on port 8000 (l.9). We can then set up listeners to respond to any given HTTP request. Above, we listen for a get request to the base URL of our server (l.5).

3. Before we can run the application, we need to update the script tag in the package.json, so that it runs the index.js file. Open the **package.json** file and update the `scripts` section so that it looks like this:

```
1  "scripts": {
2     "test": "echo \"Error: no test specified\" && exit 1",
3     "start": "nodemon index.js"
4   },
```

4. In the VS code terminal, type `npm run start` and press enter. You should see a message printed to the console saying that the server is listening on port 8000. If you open a web browser and navigate to `http://localhost:8000` you should see a message saying "Hello, World!". If you change the message to something else and refresh the page, you should see that the change has been applied.
5. If you want you can add a new route to the application. For example, you could add the following code to the `index.js` file:

```
1  app.get("/about", (req, res) => {
2    res.send("This is the about page");
3  });
```

5. To finish the task, create some further routes (e.g., `/contact`, `/about`).

Click here to see the solution

Exercise 1.1 : Serving HTML files

1. Let's update the application so that it returns a HTML page instead of a plain text message. Express makes serving static files easy. For instance, let's assume that we are making simple website. When we receive an HTTP get request to the root path of our website we would want to return an index.html file (see the sample below).

/break

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

const app = express();
const port = 8000;

app.get("/", (req, res) => {
  res.sendFile(path.resolve(__dirname, "index.html")); // serve the
      index.html file from the root of the project. Notice how we use
      the path module to resolve the path to the file - this is imported
       at the top of the file. Using the path module is a good practice
      as it ensures that the path is correct across different operating
      systems.
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

> example of a simple express application that serves a HTML file

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First Web Page</title>
  </head>
  <body>
    <h1>Contact</h1>
  </body>
</html>
```

> a simple html page served from the root directory of our application

1. Using the above example as a guide, within your `lab_9` folder set up and serve html an html page, in the root of your `lab_9` folder for each of the routes you created early (e.g., `/about` and `/contact`). Place an h1 in each page that identifies the page (see above). Work out how serve these pages. For instance, when we visit `http://locahost:8000/contact` contact.html should be served. You will need to create a HTML file for each route you created earlier (about, contact, and index). Place these files in the root of your project.
2. Currently we cannot serve static assets such as CSS and images in our HTML pages. In oder to configure this functionality we need to use some middleware. Middleware can be considered functionality that runs between an HTTP request being received and a response being sent. Express has a number of built-in middleware functions, and we can use them by calling `app.use()` and passing the middleware function as an argument. Update your program so it uses

the express.static() middleware function to serve static files from the public folder. Below, is an example of how to use the express.static() middleware function:

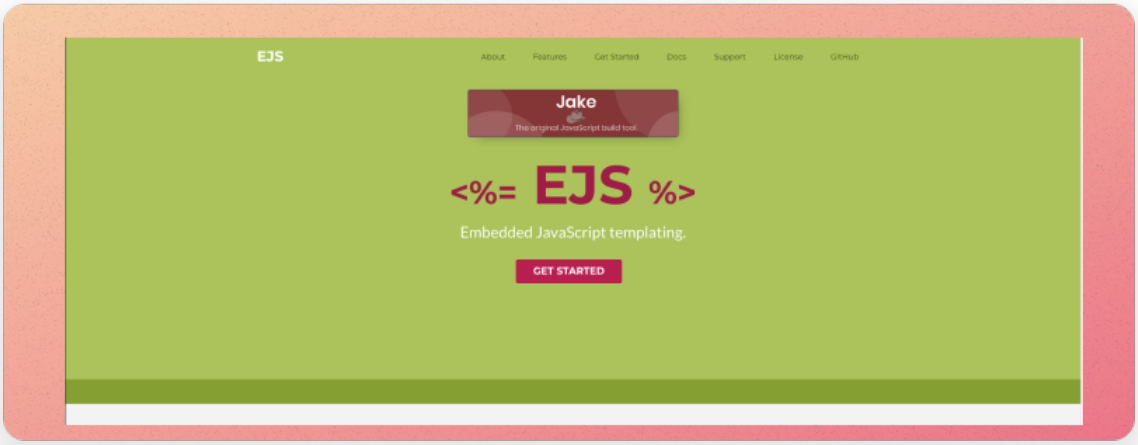
```
const express = require("express");
const app = express();
const port = 8000;

app.use(express.static("public")); // Serve static files from the
    public folder.

....
```

> Above, express looks up the files relative to the static directory, so the name of the static directory is not part of the URL. For instance, if we place the image foo.jpeg in the public folder we would reference it in index.html like this, `<img src="foo.jpeg">`.

3. Can you find and insert a picture of Rick Astley or a cute cat into one of your HTML pages. Remember to place the image in the public folder, and you should be able to reference it in your HTML page using the `<img src="filename.extension">` tag. Remember to use the correct file extension (e.g., .jpeg, .png, .gif).

Click here for the solution

2.0 EJS Templates



> The EJS website (https://ejs.co/)

While serving plain HTML files provides us with a means to present a website, we do not have much flexibility and sophistication. For instance, how do we inject data into our html pages? Furthermore,

how can we reuse sections of our HTML pages (e.g., the header and the footer)? Templating languages to the rescue! We will explore one such language, (EJS) https://ejs.co/#about.

Exercise 2.0 : Installing EJS

1. In the VS Code terminal window, type `npm install ejs` and press enter.

Now you've installed EJS, We can now tell express to render our html pages using ejs. Below is a full example:

```
const express = require("express");
const path = require("path");
const app = express();
const port = 20000;
app.set("view engine", "ejs");
app.use(express.static(path.join(__dirname, "public")));

app.get("/", (req, res) => {
  res.render("index");
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

In the above example, express will assume that we have an `index.ejs` file in a views folder which lives in the root directory of your project.

EJS, is a superset of HTML. This means, to get the above example to work, we can simply rename our index.html file to `index.ejs` and move it to a views folder.

Using EJS, we now have some dynamic capabilities within our HTML views. The first thing you might want to do is extract your header into a shared folder such as `views/common/header.ejs` we could then share it amongst our pages as follows:

```
<!-- views/index.ejs -->
...
<body>
  <%- include('common/header'); %>
  <!-- notice how we emit the .ejs extension jk -->
  <h1>Home Page</h1>
  <img src="images/test.jpeg" alt="wtf" />
</body>
...
```

```
<!-- views/common/header.ejs -->

<nav>
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="about"></a>About</li>
    <li><a href="contact">Contact</a></li>
  </ul>
</nav>
```

2. Using the above example as guidance can you update your application to use EJS templates. You will need to create a views folder and move your HTML files into it. You will also need to update your routes to render the correct EJS template. For instance, when we visit `http://locahost:8000/contact` contact.ejs should be rendered. Further, can you add a shared header and footer to your application. The header should contain links to each of the pages in your application, and the footer should contain your name and the year (e.g., `&copy; Joe Appleton 2020`). Place the header and footer in a common folder within the views folder: `views/common/header.ejs` and `views/common/footer.ejs`.

Click here for the solution

Stretch Goal

Can you add some CSS and create some basic styles for your application (e.g., a background colour, a font, and some padding, make the links vertical). You will need to create a **`public`** folder and place your CSS file in it. You will also need to update your HTML files to reference the CSS file. For instance, you could add the following line to the head of your HTML files:

```
<link rel="stylesheet" href="style.css" />
```