Installing and Running Express

In this tutorial we'll create a "Solar System" website using Express. The website will consist of a few HTML web pages with static content. It's important to mention that Express is **not** the best option to serve static HTML web pages (using Apache would be easier and faster for development). However, in future lessons, we'll learn how to make the website dynamic, getting data from APIs and databases.

In this tutorial we will learn:

- Installing Express and related packages
- Creating routes
- Serving static assets such as images, CSS files, etc.
- Creating views
- Installing and using a templating engine
- Creating and applying partial files ("partials")
- Publishing a Node App in Heroku

1. Intro to Express

Express is the most popular Node.js **web application framework**. As such, it's written entirely in JavaScript.
The Express package itself provides just the very basic functionality as a web server, however, there are several additional packages that make it as robust as any other web server (it's able to handle sessions, cookies, interact with databases, etc.)

There are several other **web application frameworks**, each of them use a specific programming language. For example, Django and Flask use Python, Ruby on Rails uses Ruby, Laravel uses PHP, and Spring uses Java. These frameworks use the MVC (Model-View-Controller) architectural pattern to separate the data model with business rules from the user interface.

To learn more visit:
https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs

2. Creating and Running an Express App

1. Create a new Github repository for this lab and integrate it with Heroku, following the naming convention. Don't forget to enable automatic deploys, unless you prefer manual deployment. Clone the repository within the "**labs**" folder of your **cst336 workspace.** Refer to the Workspace Setup document, if needed.

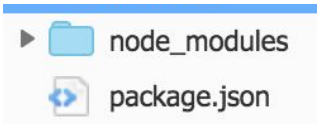
2. Within this new lab folder run the command `npm init` to create the **package.json** file. Use the default values, except for "entry point" (use "**app.js**" instead of "index.js")

```
$ npm init
```

3. Install the Express package by typing:

```
$ npm i express --save
```

"**i**" is short for "install".
"**--save**" makes the package a production dependency by adding it into the **package.json** file.



In the folder structure, you should see a new "**node_modules**" folder with many folders within it (you might need to refresh the folder in your IDE). This folder is created whenever you install a Node package for the first time.

4. Install the EJS package for templating:

```
$ npm i ejs --save
```

Templating Engines

The Express package is so minimalist that it doesn't include its own functions to render HTML files or web pages. It needs a "Templating Engine".

EJS stands for "Embedded JavaScript Templates". It allows generating HTML markup with plain JavaScript.

You can learn more about ejs here: https://ejs.co/

5. Create a new app.js empty file in this lab folder.

6. Type the following lines of code into the app.js file. This is the basic code of any Express app. Each line is explained below.

```
1  const express = require("express");
2  const app = express();
3
4  //routes
5  app.get("/", function(req, res){
6      res.send("it works!");
7  });
8
9  //starting server
10 app.listen(process.env.PORT, process.env.IP, function(){
11 console.log("Express server is running...");
12 });
```

The first line imports the Express library.
The second line is based on the Express package documentation, it requires a variable to access the methods. By convention the variable name is "**app**" but it can be any other name.
"const express = require(express)" exposes a top-level function which we then instantiate with "const app = express()"; the const in line 1 and the function in line 2 are the same.

In line 5, we are creating a "**route**". In this case, we're creating the "**root route**". This means that when opening the browser in the root folder of the application, it should display the message "it works!" We'll create more routes later.

Lines 10 through 12 allow the server to listen for any request. The first parameter is the port number and the second one is the IP address. These parameters are using environment variables (process.env.PORT and process.env.IP) but if you plan to run Express locally you could use "8080" and "127.0.0.1" instead:
app.listen("8080", "127.0.0.1", function() {

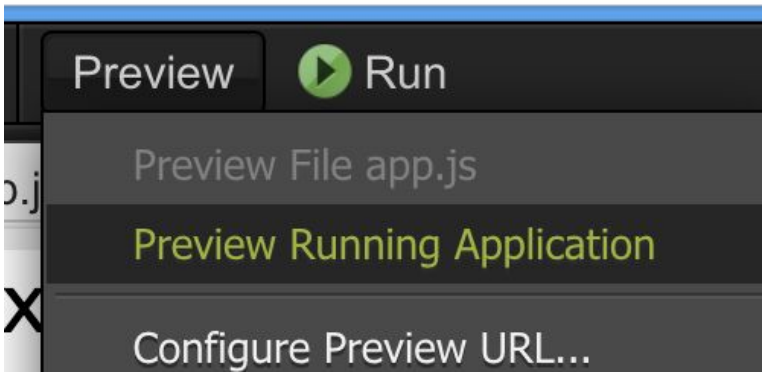
The third parameter is a callback function that displays a message in the terminal indicating that Express is listening for any request.

7. Run the app.js file by typing:

```
$ node app.js
```

You should see the message "Running Express Server…"

8. To access the app in the browser, if you're using Cloud 9, click on "Preview" > "Preview Running Application"



If you're using your local computer the url should be something like: **http://localhost**

3. Adding more Routes

The "routes" are associated with the URLs of our web server. So far, we have defined just the **root route**.

1. Define a couple of more routes in the app.js file (lines 8 to 14 in the screenshot below). Do not define more than these two for now. Don't forget to save the file with the new routes!

```
4  app.get("/", function(req, res){
5      res.send("It works!");
6  });
7
8  app.get("/mercury", function(req, res){
9      res.send("This will be Mercury web page!");
10 });
11
12 app.get("/venus", function(req, res){
13     res.send("This will be Venus web page!");
14 });
```

If you try to access these routes in the browser, you'll get an error. The app.js file must be run every time it is updated. There is another package to avoid restarting the server but for now let's keep practicing stopping and restarting the server.

2. To stop the server, open the terminal window where the server is running and press **Control+C.**
Note: For Mac users, it's still Ctrl+C. You might be used to using Cmd, but Ctrl+C is the universal shortcut for terminating console applications.

3. Run the app.js file once more:

```
$ node app.js
```

Note: If you get any error trying to run the file again, it might be that the process didn't get terminated properly. You might need to run **ps -fea** to see the list of all processes. Identify the ID of the process that is running "node app.js" and kill it by using **kill -9 *processId***, where *processId* is the actual id of that process

3. Go to the browser window where the app is running and add the new routes. You should be able to see the corresponding messages. You need to append the names of the routes on the URL to visit them, such as: localhost:8081/venus

4. Rendering HTML files

'Til now, we've used **res.send()** to display messages on each route. "**res**" stands for "**response**", in other words, it's the information sent by the server to the browser.

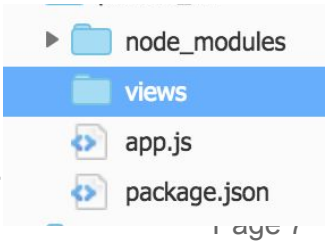
The **send()** method allows for displaying short messages and it's even possible to include HTML tags. However, it is not practical to display a lot of HTML code using the **send()** method. Instead, we'll use the **render()** method to render an HTML page.

There are a few steps that are required to render an HTML page. Let's start by rendering a web page for the **root route**.

1) In the app.js, change the code for the root route to use the render() method. We'll render a file called "index.html". Make sure to save your changes.

```
4  app.get("/", function(req, res){
5      res.render("index.html");
6  });
```

2) According to the Node framework, all html files must be placed within a folder called "**views**" (following the MVC model).



So, create a folder called "**views**" within this lab folder (at the same level as the app.js file)

3) Inside the "**views**" folder, create an **index.html** file. Add any text in it, something like:

```
<!DOCTYPE html>
<html>
    <head>
        <title> The Solar System </title>
    </head>
    <body>
    <h1> Welcome to the Solar System Site!</h1>
    </body>
</html>
```

HTML Basic Structure

Most IDEs provide the feature to generate the HTML basic structure automatically.

For VS Code users, there is an extension called "HTML Snippets" https://marketplace.visualstudio.com/items?itemName=abusaidm.html-snippets

4) By default, Node is unable to render any HTML code. The EJS package that we installed in Lesson 2 will make it possible to render the HTML files. In the app.js file, add the following line right after initializing the "**app**" variable:

```
app.engine('html', require('ejs').renderFile);
```

```
1    const express = require("express");
2    const app = express();
3    app.engine('html', require('ejs').renderFile);
```

5) After saving the app.js file, run it once more.

```
$ node app.js
```

6) Open the application in a browser window.

   You should see the content of the "**index.html**" page in the root route.

7) Update the "**index.html**" file to include links to the two other routes ("/mercury", and "/venus"). You don't need to restart the app.js file to apply the changes, just refresh the web page. Restarting isn't required because "render()" utilizes the client and not the server.

```
<body>
<h1> Welcome to the Solar System Site!</h1>
<p>Click on a planet to know more about it:</p>

    <a href="/mercury"> Mercury </a> <br>
    <a href="/venus">   Venus   </a> <br>
    Earth  <br>
```

Practice!

Now, try creating the "mercury.html" file. For now, it'd be fine if it just display a "Welcome to Mercury" message. Don't forget to put it within the "**views**" folder and update the corresponding route. You'll need to restart the app.js file too.
Every time a new directory is added, the server must restart in order to accommodate it.

5. Displaying Images and Styles

According to the Node framework all static files (images, external CSS files, external JavaScript files, etc.) must be included in a separate folder (they won't be embedded if they're in the **"views"** folder). By convention, all static files are included in a folder called "**public**" but you can use a different folder name. This folder name must be specified in the app.js file

1) In the app.js file, add the following line before the root route:

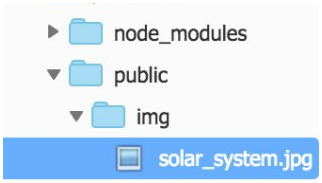
```
app.use(express.static("public"));
```

```
1    const express = require("express");
2    const app = express();
3    app.engine('html', require('ejs').renderFile);
4    app.use(express.static("public"));
```

2) Create a folder called "**public**" at the same level as the "**views**" folder. (In other words, in the root directory of our project)

3) Within the "**public**" folder, create another folder called "**img**"

4) Download any free image of the Solar System and put it within the "**img**" folder.



5) Open the "**index.html**" file and add the following HTML tag anywhere in the body to display the image:

```
<img src="/img/solar_system.jpg" alt="The Solar System" />
```

When the app.js runs, all of the content of the "**public**" folder is placed within the root folder. For this reason, the path to the image must not include "**public**" and must start with a forward slash: **/img**

Note: It doesn't matter where you place the image within the web page, the most important is that you are able to display it. If the image is too big, please resize it or use the "width" attribute within the <img> tag.

Save all of your changes, run the app.js once more and open the root route in a browser. You should be able to see the image!

6) Let's add now a couple of CSS rules in an external CSS file.

Within the "**public**" folder, create a "**css**" folder.

Within the "**css**" folder create a new empty file called "**styles.css**"

7) Type the following CSS rule in the "**styles.css**" file. You can add any other styles you want or use a different background color (colors can be selected using colorpicker.com)

```css
body {
    background-color: #CBE2F6;
    text-align: center;
}
```

8) Embed the external CSS file in the "**index.html**" file by typing the following "link" tag within the <head> section. Observe that the path to the file starts with a forward slash (the root folder): /css/styles.css.

```html
<head>
    <title> The Solar System </title>
    <link rel="stylesheet" href="/css/styles.css" />
</head>
```

9) Save the **index.html** file and preview it in a browser. Don't forget to save and run the app.js file. Your file should display the image and the corresponding CSS rules.

6. Using “Partials”

Before creating any other web page, let’s start taking advantage of some features that Express provides, such as the use of “partials”. Partials refer to partial content that can be reused and embed it in multiple files.

For instance, if the copyright year is included across all pages as part of the footer, it’d be easier and faster to change the year in just one single file than in all of them. So, the footer could be included as a partial.

1) Create a folder called “**partials**” within the “**views**” folder.

2) Create a new empty file called “**header.ejs**” within the “**partials**” folder.

3) Cut, not copy, and paste the first seven lines of HTML code from the “**index.html**” file into the “**header.ejs**” file. Save both files.

```
<!DOCTYPE html>
<html>
    <head>
        <title> The Solar System </title>
        <link rel="stylesheet" href="/css/styles.css" />
    </head>
    <body>
```

4) In the “**index.html**” file, include the “**header.ejs**” file by typing this line of code as the first line:

```
<%- include('partials/header.ejs') %>
```

```
1 <%- include('partials/header.ejs') %>
2
3    <h1> Welcome to the Solar System Site! </h1>
```

Save the "**index.html**" file and refresh it. The header should still be displayed (try changing the content of the <title> tags in the "**header.ejs**" file to confirm that it's been included properly). The lines we pasted into "**header.ejs**" are injected directly into "**index.html**" as though they never left.

> **EJS Tags**
>
> The EJS templating engine uses the tags <% and %>
> All code included within these tags will be executed using EJS.

5) Within the "**partials**" folder, create a new empty file and called it "**footer.ejs**"

6) Cut, not copy, and paste the last two lines of HTML code from the "**index.html**" file into the "**footer.ejs**" file (the closing body and html tags). Add a copyright notice and a disclaimer about the information not being accurate. Save both files.

```
1  <hr>
2  &copy; 2019. <br/>
3  Disclaimer: The information in this page might be innacurate.
4  </body>
5 </html>
```

7) As the last line of code of the “**index.html**” file, include the “**footer.ejs**” file. Save the file. You can change your header inclusion to not have the ".ejs" extension:

```
13    <%- include('partials/footer.ejs') %>
```

8) Refresh the “**index.html**” file in the browser and you should see the new footer.

> **Practice!**
>
> Try adding partials to the mercury.html page. Add an image too!

7. Publishing to Heroku

Follow these steps once you’re ready to deploy to Heroku.

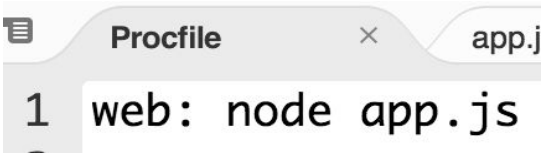
1) Make sure that the **app.js** file is using environment variables to start the server listener:

```
//listener
app.listen(process.env.PORT, process.env.IP, function(){
    console.log("Running Express Server...");
});
```

2) Open the package.json file and make sure that the value of the “main” property is “app.js”. By default, it assigns the name “index.js”, it should say app.js

```
4   "description": "",
5   "main": "app.js",
6   "scripts": {
```

3) Within this lab folder, at the same level of the **app.js** file (in the root directory of our app) create a file called **Procfile**, (check the spelling!). Heroku will use this file to handle this app as Node.



The content of this file must be:

```
web: node app.js
```

4) Within this lab folder, at the same level of the **app.js** file, create a **.gitignore** file (don't forget the period at the beginning of the name!). Open the file and type:

```
node_modules/
```

In this way, the "node_modules" folder won't be pushed to github. Heroku will install all dependencies based on the package.json file the first time it runs. This is the way that node apps work, all dependencies must be installed in the new system. The package.json ensures dependency versioning to avoid conflicts caused by the user's global installs and by any packages updated by their creators.

5) Add the files to Git by typing in the terminal the following line (don't forget the dot at the end)

```
git add .
```

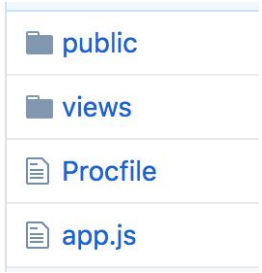
```
(master) $ git add .
```

6) Commit the files to your local repository:

```
git commit -m "Initial commit of Planets site"
```

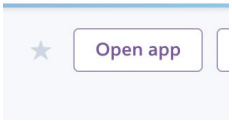
7) Push the files to Github by typing the following command. Then enter your Github username and password.

```
git push
```



8) Open a new tab in the browser and go to your Github repository. Double check that the files have been uploaded properly.

9) Login into Heroku and access this app. Click on the "Open App" button located at the top-right corner of the page. You should be able to see your Planets site!



It's Your Turn!

1) Add Bootstrap to all pages (include the Bootstrap css file in header.ejs)
2) Add content (between 50 and 100 words would suffice) and at least one image for Mercury, Venus, and Earth pages.
3) Add a navigation menu across all pages (consider including it in the header.ejs file)