Lab Exercise: Create a Node.js and Express.js Web Application with Pug.js Template Engine

Overview of the lab exercise:

- Install Node.js, npm, express-generator, and nodemon on your system.
- Generate a new Express.js app with Pug as the view engine.
- Add a development startup script to your package.json file.
- Start the web server in development mode and preview the web app.
- Understand the default structure and functionality of the Express app.
- Use Pug syntax to create HTML templates.

Part 1: Node Installation

Step 1: Check for Node.js
Before you begin, check if Node.js is already installed on your computer. If it is, you can skip the installation step and move on to installing express-generator and creating a new Express app.

Step 2: Download Node.js Installer
If Node.js is not installed, you'll need to download the installer. Visit the official Node.js downloads website (https://nodejs.org/en/download) and select the LTS (Long Term Support) version that's compatible with your operating system.

Step 3: Install Node.js
The installation process varies depending on your operating system:

1. Open the Node.js installer (it will be a .msi file for Windows and a .pkg file for macOS).
2. Follow the prompts in the installer. When you reach the screen labeled "Tools for Native Modules", make sure to check the box that says "Automatically install the necessary tools".

Step 4: Open Terminal or Command Prompt
After you've successfully installed Node.js, open a terminal (on macOS) or command prompt (on Windows).

Step 5: Update npm
Next, you'll need to update npm, which is the package manager for Node.js. This can be done by running a specific command. However, it's important to note that the `-g` switch in this command means that npm will be updated system-wide and not just for the current Node.js application. Here's the command you need to run:

```
npm install npm@latest -g
```

This command tells npm to install the latest version of itself globally on your system. Once this process is complete, you're all set and ready to start using Node.js.

Step 6: Install express-generator
Next, you'll need to globally install the `express-generator` application. This can be done by running the following command:

```
npm install -g express-generator
```

This command tells npm (Node Package Manager) to install the `express-generator` application globally on your system. The `-g` switch specifies that the software is installed system-wide, not just for the current Node.js application.

Once this process is complete, you're all set and ready to start using `express-generator`.

Part 2: Make a new Express app

Step 1: Generate a New Express.js App
Open a terminal or command prompt window and run the following command to generate a new Express.js application. In this case, the application is named `myapp` and the view engine is specified as `pug`.

```
express myapp --view="pug"
```

Step 2: Change Directory
Next, navigate into the newly created Express app directory using the following command:

```
cd myapp
```

Step 3: Install Dependencies
Once inside the Express app directory, run the following command to download and install the required dependencies as listed in the `package.json` file:

```
npm install
```

Step 4: Apply Security Updates
If any security updates are available for the installed dependencies, a notification will be displayed. If so, apply these updates using the following command:

```
npm audit fix --force
```

Step 5: Install nodemon
Finally, while still in the Express app directory, install `nodemon` using the following command. The `--save-dev` option indicates that `nodemon` is a development dependency and not used in the application itself, but as a tool during development.

```
npm install --save-dev nodemon
```

Once these steps are complete, you're all set and ready to start developing with your new Express.js application.

Part 3: Add a Development Startup Script

Step 1: Open package.json
Open the `package.json` file in your app directory using a text editor. This JSON file specifies the dependencies used by your Node app and contains named startup scripts that start the application in different ways.

Step 2: Locate the "scripts" Entry
In `package.json`, find the "scripts" entry. By default, it contains only one script ("start"):

```
"scripts": {
  "start": "node ./bin/www"
}
```

Step 3: Add a Development Startup Script
Add a new line that defines a script `devstart`. The way you define this script depends on your operating system:

- For macOS:

```
"scripts": {
  "start": "node ./bin/www",
  "devstart": "DEBUG=myapp:* nodemon ./bin/www"
}
```

- For Windows:

```
"scripts": {
  "start": "node ./bin/www",
  "devstart": "SET DEBUG=myapp:* & nodemon ./bin/www"
}
```

These scripts ("start" and "devstart") can be executed by running the command `npm run <scriptname>`.

Step 4: Start the Web Server in Development Mode
The command `npm run devstart` starts the app with two additional development features enabled. The DEBUG environment variable is set, specifying that the console log and error pages, such as HTTP 404, display additional information, like a stack trace. In addition, `nodemon` monitors certain important website files. If you modify these files, such as redesigning a page or modifying static content, `nodemon` automatically restarts the server to reflect the changes.

To start the web server in development mode, run:

```
npm run devstart
```

Note: If the Windows Firewall blocks the web server application, click Allow Access.

Step 5: Preview the Web App
When the application is running, your computer acts as a web server, serving HTTP on port 3000. To preview the website, open a web browser and navigate to `localhost:3000`.

Overview of the default Express app

The default structure of the Express app is listed here, with descriptions of each file and directory.
myapp/ (Contains the entire Express app)

```
├ app.js                 The core logic of the Express app.
├ bin/                   (Contains the app's executable scripts)
   └ www                 A wrapper that runs app.js.
├ node_modules/          (Contains dependencies installed by npm)
├ public/                (Files downloaded by the user's web browser)
  └ images/              (Contains client-accessible image files)
  └ javascripts/         (Contains client-accessible JavaScript files)
  └ stylesheets/         (Contains client-accessible CSS)
     └ style.css         The site's CSS stylesheet.
├ routes/                (Contains logic for individual site routes)
  └ index.js             Logic of the "index" route (/).
  └ users.js             Logic of the "users" route (/users).
├ views/                 (Contains HTML templates)
   └ error.pug           View displayed for error pages, such as HTML 404.
   └ index.pug           View displayed for the site root (/).
   └ layout.pug          View template of layout shared by all pages.
├ app.js                 Define the core functionality of the website.
├ package-lock.json      JSON manifest of installed dependencies.
├ package.json           JSON of dependencies and config specific to your app.
```

In the app.js file, we define and name different sections of the website, which we refer to as ‘routes’. Each route corresponds to a unique path in the website’s URL. For instance, www.example.com/search and www.example.com/login are examples of routes.

Each route is associated with a specific ‘route logic script’, which is stored in the ‘routes’ folder. When a user navigates to a particular route, the corresponding route logic script is triggered. This script processes the incoming HTTP request data and subsequently sends a response back to the user.

The ‘views’ folder contains HTML templates, also known as ‘views’. These views are processed by a view engine, in this case, Pug. The view engine takes the data provided by the route logic script and combines it with the appropriate view to generate the final HTML that is sent to the user’s browser.

Introduction to Pug.js

Pug is a template engine for Node.js and for the browser. It compiles to HTML and has a simplified syntax, which can make you more productive and your code more readable1. Pug makes it easy both to write reusable HTML, as well as to render data pulled from a database or API.

Pug syntax is designed to be a simpler and more readable way to write HTML. Here are some of the basic elements of Pug syntax:

1. DOCTYPE: You can use Pug to generate a number of document type declarations.
2. Tags: Pug doesn't have any closing tags and relies on indentation for nesting.
3. Classes, IDs and Attributes: Classes and IDs are expressed using a `.className` and `#IDname` notation.
4. Plain Text and Text Blocks: You can write plain text in your Pug templates.

Here's an example of how you might write a basic HTML structure in Pug:

```
doctype html
html(lang="en")
 head
  title Your Page Title
  meta(charset="UTF-8")
  meta(name="viewport", content="width=device-width, initial-scale=1")
 body
  header
   h1 Your Header
  main
   p Your content here...
```

In this example, each indentation level represents a nested HTML element. The `doctype html` at the top specifies the doctype, and the `html(lang="en")` specifies the root `html` element with a language attribute of "en". The `head` element contains nested `title` and `meta` elements, and so on.

Creating multiple pages in Pug with Node.js

1. Create your Pug templates: In your project directory, create a new directory for your Pug templates (e.g., views). Inside this directory, create a new Pug file for each page you want to render. For example, about.pug.

   ```
   // about.pug
   doctype html
   html
    head
     title About Page
    body
     h1 Welcome to the About Page
   ```

2. Create routes for each of your pages:

```
app.get('/about', (req, res) => {
   res.render('about');
});
```

3. Start your server: Run your server file with Node.js. If you navigate to http://localhost:3000/about, you should see your about page.