Assignment: Forms, Validation, Templates, and Express Middleware

The purpose of this assignment is to get practice using some additional features of Express, including middleware and template engines. We'll explore these things by creating a web application for ordering pizza, with form validation.

Task 1: Setup

Create a new Node project and install express. As usual, create a new application script called **app.js** that runs an Express server.

In the starter files, I've provided a video demonstrating how the application should work when complete. As you can see, users will fill a form with some of their personal information and details of what kind of pizza they'd like to order. They can also use a form to change the background color of the page.

I've given you two files—**index.html** and **style.css**—that you can import into your project. Place them in a subfolder called **public**.

Task 2: Express Middleware

Set up the **Express static** middleware to serve the files in **/public.** When visiting the path **"/"** one should see the Langara Pizza homepage with CSS styles applied.

Set up the **Express urlencoded** middleware to parse the **body** of POST requests that contain form data.

Task 3: HTML Form

In **index.html** I've create a basic page for ordering pizzas; you just have to write the code for the form! It should have the following:

- send requests to the **'/orders'** endpoint using the **POST** method;
- a **radio button group** for choosing the sauce; the two options should be **tomato** or **alfredo; tomato** should be chosen by default.
- a **checkbox** group for choosing the toppings. See the video for the list of toppings that should be offered.
- the **name** of the person making the order.
- the **email address** of the person making the order. Normally, one would use **type="email"** for this, but I'd like you to use **type="text"** to prevent browser-based validation (thereby letting us test our own validation script that we will write later).
- A submit button.

Task 4: Route Handler for POST '/orders'

In **app.js,** create a **route** for handling POST requests to **'/orders'.** For now, simply send a response with the **body** of the request encoded as JSON. (we're basically sending the request body right back to where it came from.)

Task 5: Validation Middleware

Create a new script in your Node project called **my-validators.js**. It should export two **Express middlewares: validatePizza** and **validateColor**. Import these middlewares into **app.js**.

The purpose of these middlewares is to sanitize and validate form data submitted by users; this means checking to make sure the form inputs have been filled correctly, with data in the correct format, and modifying the user's input to ensure it conforms with the desired format, applying defaults if no input is given, and producing an error if the form input is unacceptable and cannot be automatically converted to something acceptable.

Let's start with **validatePizza**. Place this validator in the middleware chain to process and modify requests sent to the **POST '/orders'** route. This middleware should do the following:

- Check the **request body** to ensure that all parts of the pizza form have been filled correctly. (More details on this later.)
- If any parts of the form have NOT been filled correctly (or haven't been filled at all), an **error** object should be created and stored in **res.locals.errors.** Initialize **res.locals.errors** to an empty array so that you can push error objects to it when needed. Remember that **res.locals** will be accessible by subsequent middleware in the chain and the route handler.

As you're developing this validator, test it using the following method:

- In **app.js**, you ought to be able to access the array **res.locals.errors** to check if it's empty or not. If it's empty, that means there are no errors, and you can simply send **the request body** back to the client. If it's NOT empty, that means there was a validation error somewhere, and you can send **res.locals.errors** to the client as a JSON object with a status of **422.**

Here's a more detailed breakdown of how the validation should work:

- **Sauce:**
    - check the value to make sure it is one of either **"tomato"** or "**alfredo"**. You may think that if these are the only two options offered on the form, the data submitted by the user *must* be one of the two. However, the browser's inspector or some other tool could be used to modify the HTML before sending the request, meaning they could send a different value or no value at all.
        - Hint: you can quickly check to see if a value is included within an array by using JavaScript's **includes** method on the array.
    - If a sauce type is not submitted or if the value is not one of the two acceptable options, create an error object with the following format and push it into **res.locals.errors:**

      {
      field: the form field for which the error was generated,
      message: a brief message to explain what the problem is
      }

      All errors that occur during validation of any part of the form should have this format.

- **Toppings:**
    - Between 1 and 3 toppings should be selected.
    - The values of all selected toppings should be among the options presented in the form (the user should not be able to modify the HTML and send a different topping)
    - If either of the two above validations fail, push an error object to **res.locals.errors** with the same format you used earlier.
    - Warning: choosing only one topping (which is allowed) may break your validator; test it thoroughly to make sure it handles this case!
- **Your Name:**
    - The customer's name should be between 3 and 30 characters. However...
        - It's unlikely that we're the first ones ever to write this type of validator, and it would be nice if we didn't have to reinvent the wheel. Luckily, many pre-written and open-source validation libraries exist for us to borrow! Install the **validator.js** library using NPM: https://www.npmjs.com/package/validator
          This library has many pre-written validation and sanitization functions that we can use to save time. In the case of validating the customer's name, we don't want **space characters** that appear before or after the name to be counted in its length. In fact, it would be best to remove unnecessary spaces before accepting the input. Figure out how to use one of **validator.js's sanitizer** functions to **trim** excess spaces from **before** and **after** their name.
            - After the spaces have been trimmed off, you can then calculate the length of the name to make sure it's within 3 and 30 characters.
    - As usual, if the name is too long or too short, produce an error object in the usual format.
- **Email:**
    - Use one of the **validator** functions provided by **validator.js** to ensure this field is formatted as a proper email address. If it isn't, produce an error in the usual format.

Task 6: Templates

At this point, your pizza form should be working properly with validation. However, the application is currently sending JSON responses to the client, which is good for testing but not what we want for production. Our application should be sending HTML pages to the client: a **success page** for a properly-filled form, or an **error page** if validation fails.

In the last assignment, we used String **template literals** to embed application data into HTML responses. However, there's a better way to do it: using **Express template engines!** Template engines allow us to write template files with slots to insert data from our application. Express then converts the template to a regular HTML file and sends it to the client. This is a more sophisticated form of server-side rendering.

In **app.js,** add the following code before your routes:

```
app.set('view engine', 'ejs');
```

We're using Express's **set** method to configure our server application to use the **EJS** template engine. There are many different template engines to choose from (Express uses **Pug** by default). The benefit of EJS is that it allows regular JavaScript code to be used for templating logic. Go ahead and install the **EJS** package using NPM: https://www.npmjs.com/package/ejs

Create a folder in your project called **views**. Inside this folder, create two files: **success.ejs** and **error.ejs**.

I've provided a set of slides called **Express Templates** on Brightspace. At this point, please review these slides to get an idea of how template engines and EJS work. You do NOT have to use **includes** for this assignment.

The template **success.ejs** should simply display the name and tagline of the site, a success message, the **name of the customer,** and their **email address**. CSS styles should be applied using the same stylesheet as before in **/public/style.css**

The template **error.ejs** should display the name and tagline of the site and a **description list <dl>** with each error, both **field** and **message**. Use EJS's programatic logic capabilities to iterate through the **res.locals.errors** array with a **loop**.

Task 7: Validating Query Parameters

It's time to turn our attention to the form for choosing a background color! As you can see from the HTML code, this form actually submits a GET request. This makes sense, as the form is NOT meant to create something on the server (like a new pizza order) but to simply GET a home page with a different color.

With GET requests, we typically send data in the **URL** of the request instead of the **body**, and this data is typically meant to specify options for what kind of resource we want to get or how the resource should be formatted. For example, when you submit a Google search query, your query parameter is encoded in the URL like this:

https://www.google.ca/search?q=cats

As you can see, at the end of the URL there's a question mark **?** followed by a **key=value** pair, where the key is **q** (presumably meaning "query") and the value is **cats.** This is called a **query parameter.** When we send a form with the GET method, the form data is encoded in the URL as a series of **query parameters.**

In our server-side application, we can access query parameters using **req.query.keyOfParameter.**

Your next task is to do the following:

- In **my-validators.js,** write the code for **colorValidator** to do the following:
    - Check **req.query.color** to see if it's a valid **hex color.** Figure out which function from the **validators.js** library you can use to do this.
    - If the color is NOT a valid hex color, or if no color is submitted, set a **failover/default** color with the following value: **#fffeed**
- Create a new route in app.js for **GET '/'.** Figure out how to stop the **Express static** middleware from taking over requests to this endpoint. The route handler should render a new template file called **index.ejs.** This template should have exactly the same HTML code as **index.html,** except that it should style the **<body>** to have the **background color** chosen by the user (or the default color if the page is loaded without using the color picker form).
- To test that the color validator/sanitizer is working, try modifying the query parameter in the URL of the page and give it an invalid color. For example: ***https://your -unique-url.app.github.dev/?color=%23not-a-valid-hex-color*** Notice that the hash # symbol shows up in the URL as **%23**, which is the special code for that character.

Hand In

Download your project and rename the folder to **a4-firstname-lastname.** Delete the **node_modules** folder and any other unnecessary files (such as these instructions, the demo video, etc.) Zip the folder and hand it in to Brightspace.

Grading

- **Task 2**
  - **[2 marks]** static and urlencoded middleware set up
- **Task 3**
  - **[1 mark]** Form: action and method set correctly
  - **[4 marks]** Form: Inputs for sauce, toppings, name, email
- **Task 4**
  - **[1 mark]** route hander for POST '/orders' set up
- **Task 5**
  - **[1 mark]** middleware functions set up in separate script and exported/imported properly
  - **[4 marks]** validation and sanitization with errors (1 mark for each type of input)
  - **[1 mark]** route handler receives errors from middleware and chooses response appropriately
- **Task 6**
  - **[2 marks]** templates set up for success and error pages
- **Task 7**
  - **[1 mark]** middleware for validating color query parameter
  - **[1 mark]** EJS template used for index, with correct color displayed

Total: 18
As usual, -25% may be deducted for poor organization and improper hand in.