Project Requirements Implementation

1. User Login

Requirement:

Allow the customer to enter their login details and have login details validated (via a login screen) before receiving a summary of the order.

Implementation:

- **Username**: user
- **Password**: pass
- **Overview**: The login feature is implemented using HTML5 for front-end validation and Node.js for back-end authentication.
- **Details**:
  - HTML: The login form is created in **login.handlebars**.
  - JavaScript: Authentication is handled by the **handleLogin()** function in **scripts.js**.
  - Node.js: Server-side validation is performed using Express by the **/login** POSTroute in **app.js**.
- **Description**: The user can log in either by clicking on the login link in the navbar or by trying to review their purchase summary, which will redirect them to /login. The Express application serves the login form via a GET request, providing a simple Bootstrap form with fields for username and password. HTML5 handles validation, requiring both fields. Authentication is managed by the Express app through a POST request to the /login route. User data is retrieved from the SQL database, and a boolean value (true or false) is returned based on the authentication result. This boolean flag is then saved in localStorage, so users don't need to authenticate frequently.
- **Improvement**: Implement encryption-based authentication using JWT, or offer additional options with existing credentials (such as Google or Facebook), using the Passport NPM package.

2. Form Validation

Requirement:

Create a form and perform form validation through JavaScript or HTML to ensure:

- a. Text fields are not empty.
- b. A valid email address is entered.

Implementation:

- **Overview**: Form validation is implemented using both HTML5 attributes and JavaScript functions.
- **Details**:
    - HTML: The login form is created in **basket.handlebars**.
    - JavaScript: Validation triggered by the **confirmPurchase()** function in **scripts.js**.
- **Description**: When the user is shown the total of their order, they are prompted to choose a payment method, add a comment, and provide their email address so the order summary can be emailed to them. The confirm purchase button is outside of this form, preventing the use of HTML5 built-in validation. To address this, I added a hidden button of type submit within the form, which I then clicked programmatically if the form's validity check failed.
- **Improvement**: The form validation works well enough as is, but I could implement a real payment method and enable sending the order summary via email in my Express application once the payment is successfully processed.

3. Image Slideshow/Carousel

Requirement:

Include a slideshow or carousel which displays a different image each time the page is loaded.

Implementation:

- **Overview**: A carousel is implemented using Bootstrap 5 framework.
- **Details**:
  - HTML: The carousel is created in **home.handlebars**.
  - JavaScript: Randomization of the first image is handled by the **randomizeCarousel()** function in **scripts.js**.
- **Description**: When the home page loads, I select all the images in the carousel and update their src attributes with different image URLs. I achieve this by shuffling an array of possible URLs using the **shuffleArray**() function. Since I only have three items in my array, it's not always possible to guarantee a different first image with each shuffle. To ensure the first image is different with each page load, I save the current first image URL in localStorage. On the next shuffle, if the new first image matches the previous one, I recursively call the **randomizeCarousel**() function until I get a different first image.
- **Improvement:** The entire randomization logic should be executed within a mutation observer before rendering occurs to prevent the initial image from flashing to the user before the shuffle takes place.

4. Purchase Items

Requirement:

Allow the user to ‘purchase’ items from the site.

Implementation:

- **Overview**: Users can select products and quantities, see the total cost, then click confirm purchase after filling in some details.
- **Details**:
    - HTML: This functionality spans over **home.handlebars** and **basket.handlebars**.
    - JavaScript: The following functions are involved from picking to purchasing items, **updateBasketCount(), displayBasketCount (), addToBasket(), showInfoPopup(), handleRouteToBasket(), showOrderCountAndPrice(), updateBasket(), handleQuantityChange(), removeFromBasket(), handleDeleteFromBasket(), openPurchaseModal(), reopenPurchaseModal(), closePurchaseModal(), confirmPurchase()** in **scripts.js**.
    - Node.js: Data for the homepage and the basket is served using GET requests from **/home, /basketdata for the homepage and the basket is served using GET requests from,** respectively, in **app.js**.
- **Description**: When the user visits the home page, the collection of products is displayed for browsing. JavaScript functions such as **addToBasket()** enable users to add items to their basket. Upon selecting items, **updateBasketCount()** updates the displayed count of items in the basket, ensuring users are informed about their selections.

  Upon navigating to the basket page, users are presented with their chosen items. Here, they can adjust the number of items using **handleQuantityChange()**, triggered by plus and minus buttons. If users wish to remove an item from their basket, **handleDeleteFromBasket()** handles this action upon clicking the delete button. Throughout this process, **showOrderCountAndPrice()** keeps the user informed about the unit price and subtotal of their selections.

  If users attempt to view the purchase summary without being logged in, they get redirected to the login form. Upon successful login, they are navigated back to the purchase summary, offering a seamless user experience.

Once users are satisfied with their selections, they proceed to inspect the purchase summary. Here, they input additional details such as payment method and email. Upon clicking purchase, **confirmPurchase()** handles the “transaction”. Users receive a thank you message and are then redirected back to the home page to continue their browsing experience.

JavaScript functions such as **openPurchaseModal()**, **reopenPurchaseModal()**, and **closePurchaseModal()** enhance user interaction by managing modal displays and interactions.

- **Extras:** When a user clicks to add a product to their basket, a useful tooltip is displayed to provide feedback on the success of the operation. This feedback is facilitated by the **showInfoPopup()** function, which informs the user whether the addition was successful or not. If the item is already present in the basket, the tooltip communicates this to the user ensuring clarity.

  Alternatively, if the user clicks on the product card instead of the add button, they are redirected to a product details page. This page, accessible via **getProductDetails()** which routes to **/plants/plantId**, and allows users to learn more about the product, read additional information, inspect user reviews, and view recommended similar products.

5. Use of JavaScript Object/Array

Requirement:

Use an object or an array in JavaScript.

Implementation:

- **Overview**: Basket items are stored in an array of objects in localStorage, and all the methods in **scripts.js** are arranged within a global main object called **WAD**
- **Details**:
  - JavaScript: The following functions are involved in maintaining the basket array in the localStorage, **updateBasketCount(), addToBasket(), updateBasket(), removeFromBasket()** in **scripts.js**.
- **Description**: To keep track of items added to the basket, along with details like quantity and price, I've structured an array of objects stored in localStorage. This array follows this format:

  ```
  [
     {"count":"3","productId":1,"productPrice":25},
     {"count":1,"productId":3,"productPrice":45},
     {"count":1,"productId":6,"productPrice":18}
  ]
  ```

  This stored information is used in various tasks, such as performing GET requests to fetch relevant data in the **/basket** route. Additionally, it's used to calculate subtotal and total costs, as well as display the number of items in the navbar.

  I've organised all my frontend JavaScript functionality within a global object, mimicking a small library format. This approach allows me to call my methods inline in HTML, facilitating scoped functionality and preventing interference with third-party libraries. Additionally, I find that it enhances readability and simplifies file navigation.

6. Custom Node Module

Requirement:

Use at least one custom module in Node.

Implementation:

- **Overview**: A custom module **db.js** is created to handle database connection and queries.
- **Description**: This module uses the **mysql2** library to interact with a MySQL database, and consists mainly of an async **getDBdata()** function that takes a SQL query as input. This function establishes a connection to the MySQL database, then, it executes the provided SQL query to retrieve and return the result.

7. Handling POST and GET Requests

Requirement:

Include capability for handling 'post' and 'get' requests.

Implementation:

- **Overview**: Express is used to handle GET and POST requests.
- **Details**: Routes are defined in **app.js**
- **Description:**
  POST:
  - /login: handles authentication by comparing the data in the request body with the information retrieved from the database.
-

  GET:
  - /home: renders the **home.handlebars** template and returns a list of all the products available in the database
  - /login: renders the **login.handlebars** template
  - /plant/:id: renders the **plant.handlebars** template and returns all data about a given product
  - /basket: renders the **basket.handlebars** templates and returns data about the content of the shopping basket

8. Static and Dynamic Content

Requirement:

Include both static and dynamic content.

Implementation:

- **Overview**: Static content (HTML, CSS, JS) and other assets are served alongside dynamic content generated by Handlebars templates.
- **Details**: Static content is located in the **public** folder and includes:
    - HTML: The **about.html** page, which serves as a static HTML file.
    - CSS: All custom styles are located here.
    - JavaScript: All logic for page interactions is stored here.
    - Images: Since there's no real database, all images are hosted here.
    - SVGs: Various icons are also stored in this folder.

9. Use of Templates in Node

Requirement:

Include the use of Templates in Node.

Implementation:

- **Overview**: Handlebars is used for templating.
- **Details**: Templates are defined in the **views** directory, this includes:
    - main.handlebars: contains logic and elements global for the entire project, such as the navbar and loading of static assets and 3rd party libraries (Bootsrap5 and Font Awesome)
    - login.handlebars: renders the login form
    - home.handlebars: renders the carousel along of a list of all available products
    - basket.handlebars: renders a list of all products added to the basket as well as the checkout form
    - plant.handlebars: renders a detailed view of the product including user reviews and similar products
    - error.handlebars: renders the error page template along with an error messge

10. Error Messages

Requirement:

Include error messages to provide feedback to users in case of any issues or errors.

Implementation:

- **Overview**: Error handling middleware in Express and front-end validation messages.
- **Details**: Error messages are shown in **error.handlebars**, and in **login.handlebars** if authentication fails**.**
- **Description**: A catch-all middleware is implemented to notify users when they attempt to access undefined routes, presenting a 404 error. Additional error messages are displayed in specific scenarios, such as when a user attempts to access an empty shopping basket and the route fails to retrieve items from the database.

  If a user tries to log in with invalid credentials, a message is displayed to inform them of the error.

11. MySQL Database Connection

Requirement:

Connect to a MySQL database that contains relevant site information using Node.

Implementation:

- **Overview**: MySQL database connection is established using the **db.js** module.
- **Description**: This was already covered in requirement Number 6

12. Use of Bootstrap 5 Framework

Requirement:

Use the Bootstrap 5 framework (via CDN).

Implementation:

- **Overview**: Bootstrap 5 is used for styling and layout.
- **Details**: Included via CDN in both **main.handlebars** and **about.html**
- **Description**: Bootstrap 5 was utilized to manage global styles, including the carousel and grid/flex layouts. This provided a consistent layout and responsive design out of the box for the most part. However, custom styles and media queries were incorporated to ensure full responsiveness.