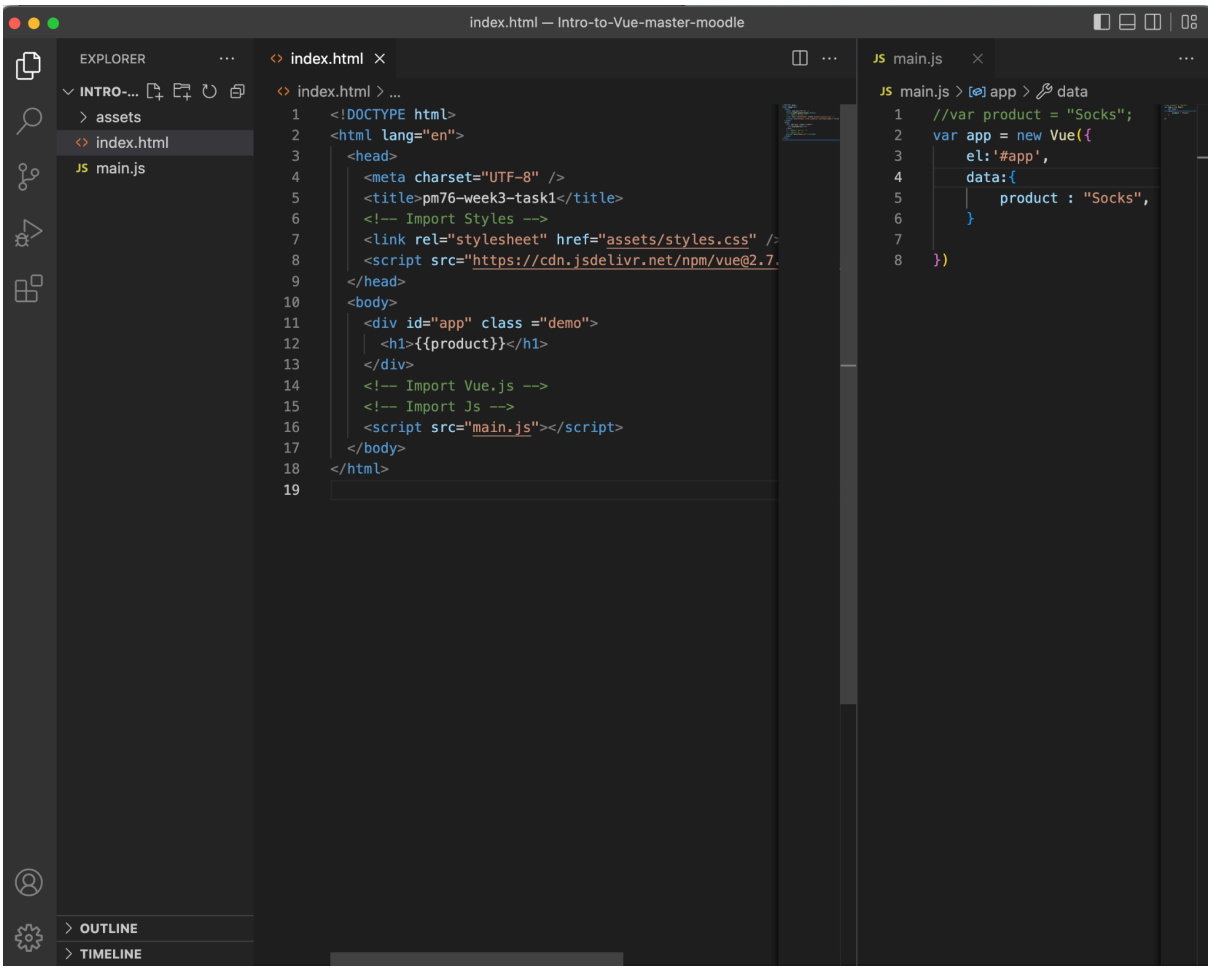
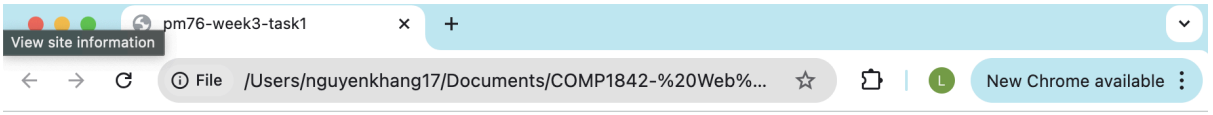
1. Appendix1:

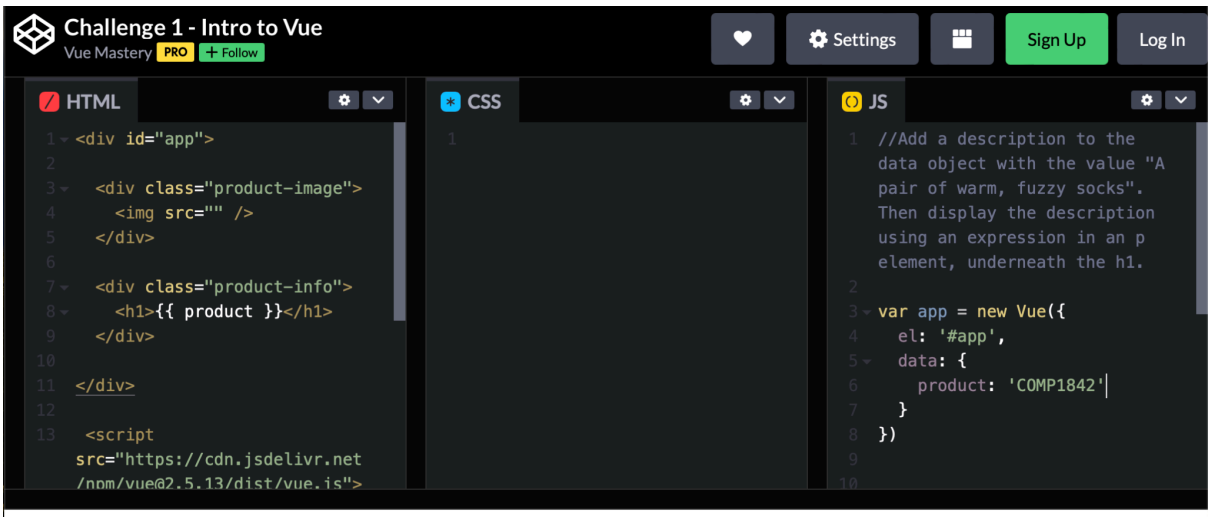
Task 1:





Socks

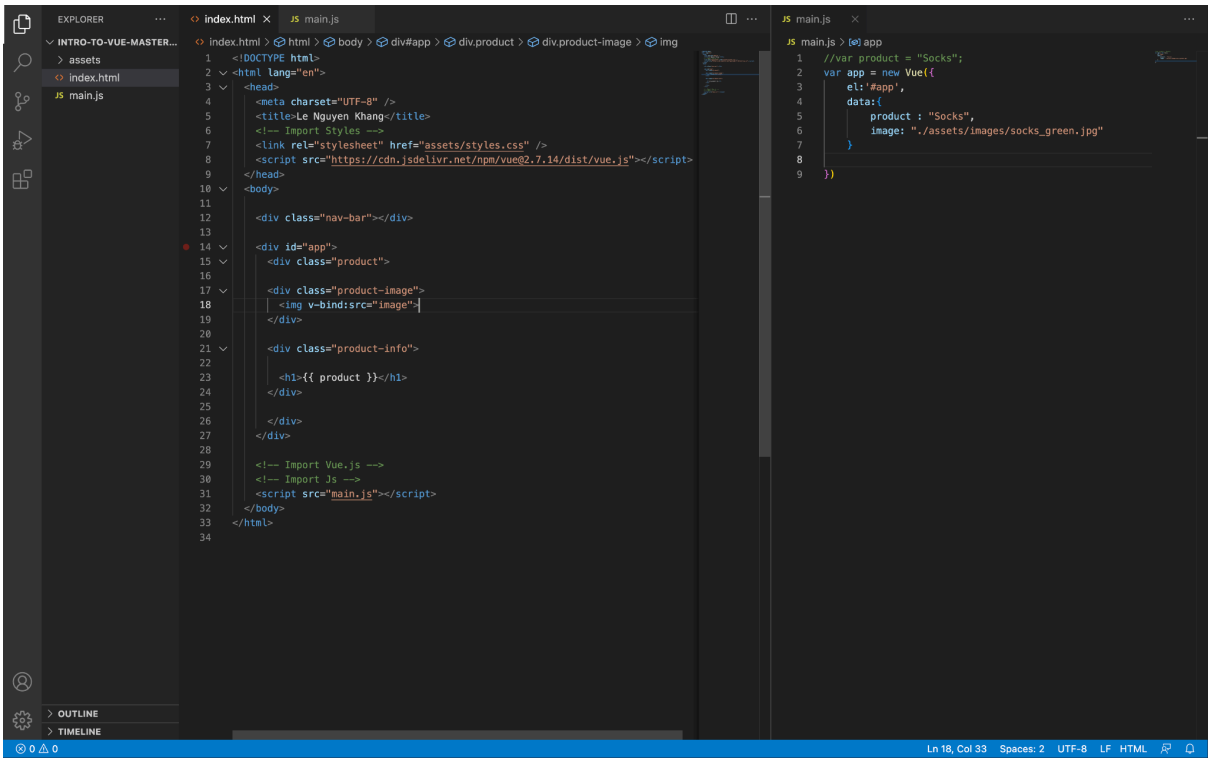
The Code for Task 1



COMP1842

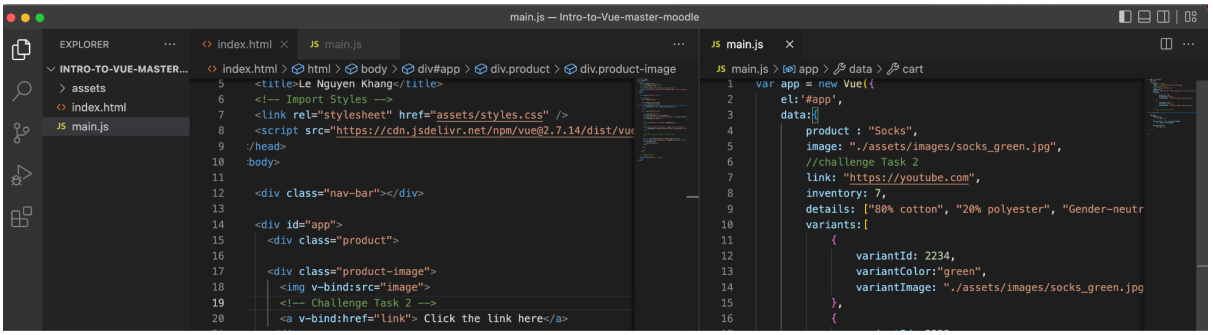
Challenge for task 1

Task 2:



Socks

The code for Task 2



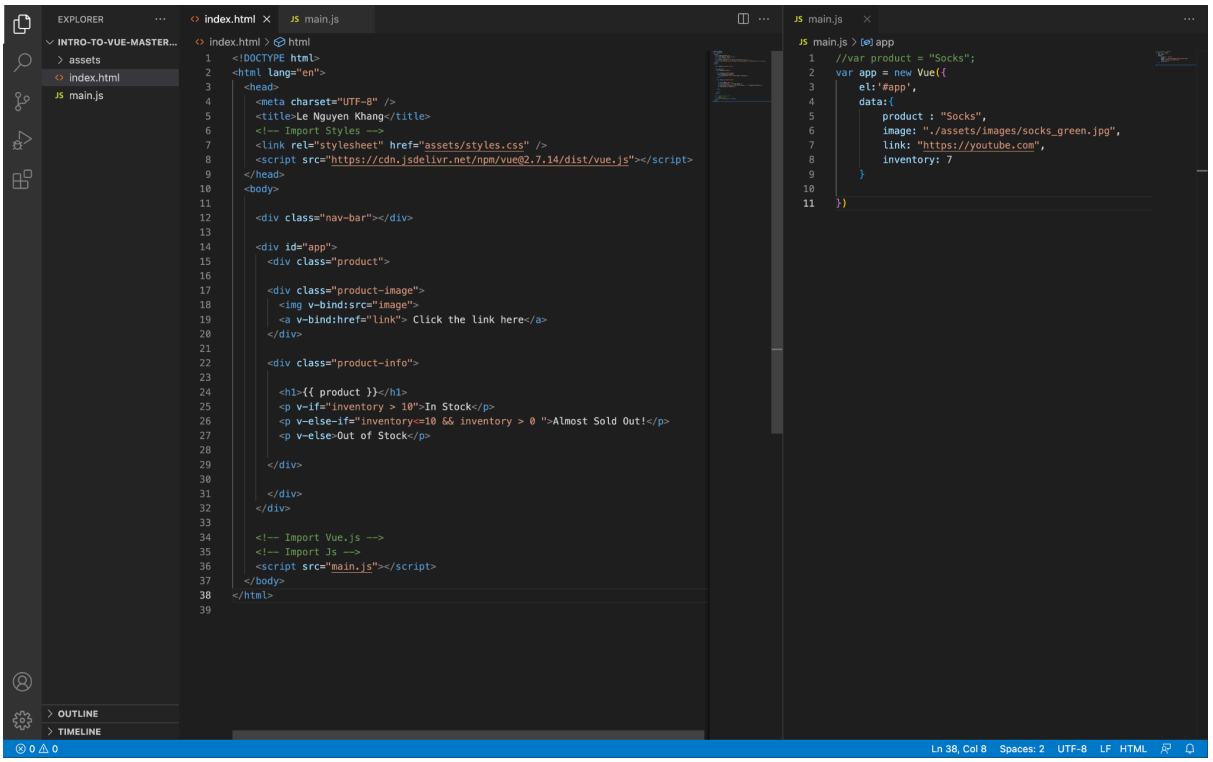
Socks

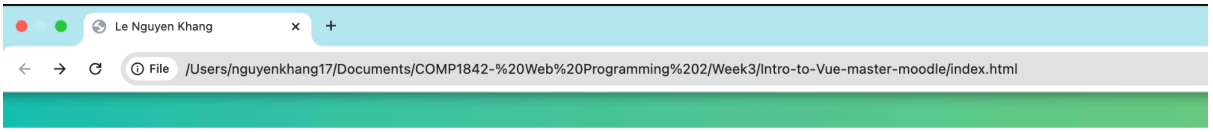
Click the link here

https://youtube.com

Challenge for task 2

Task 3:



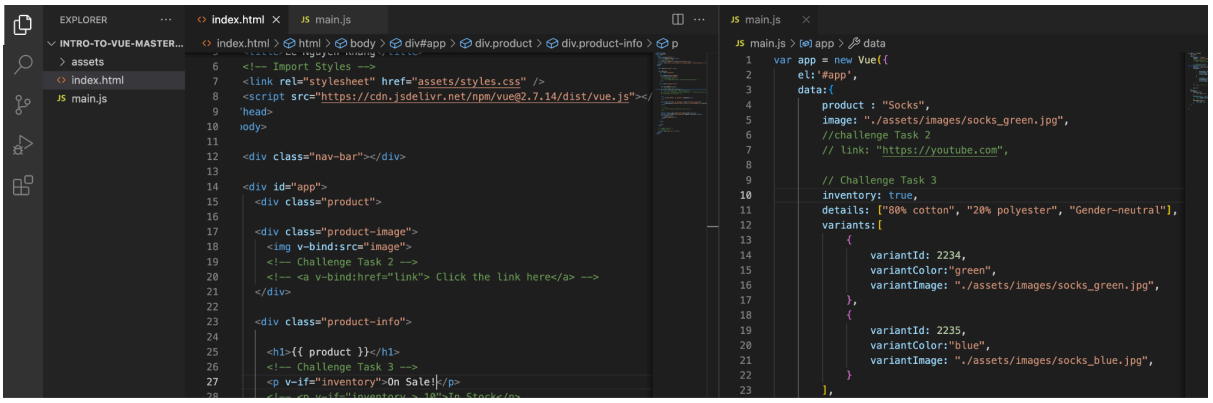


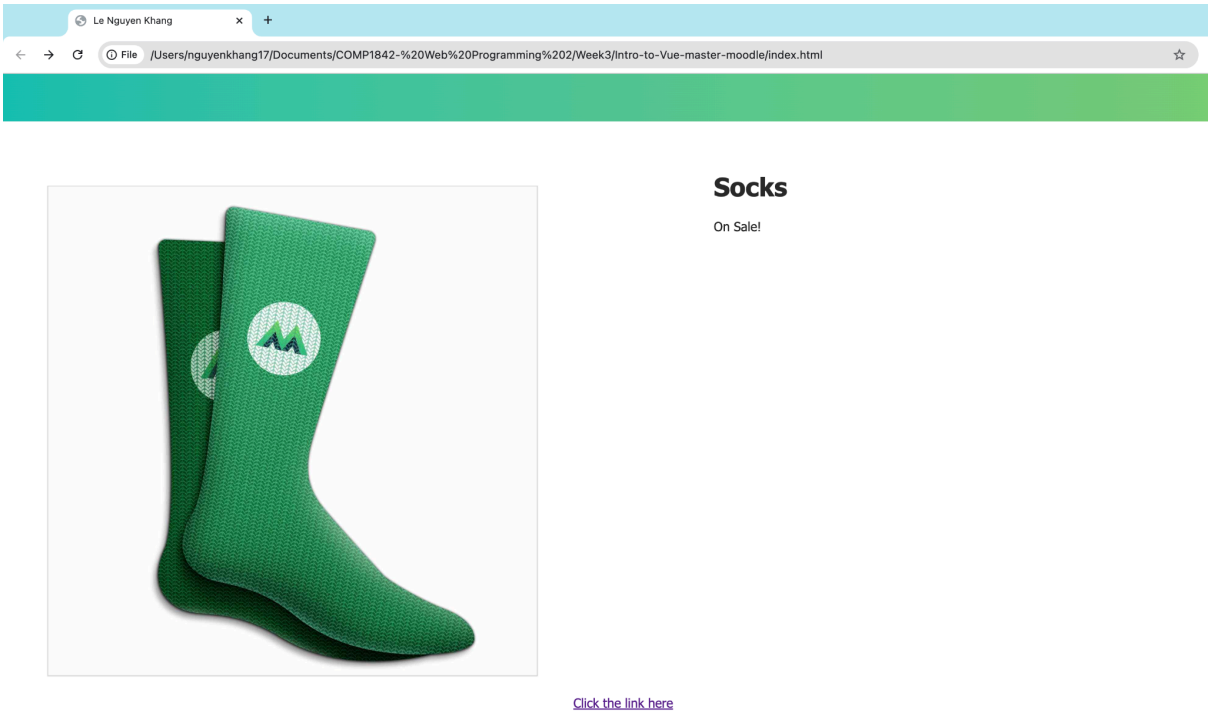
Socks

Almost Sold Out!

Click the link here

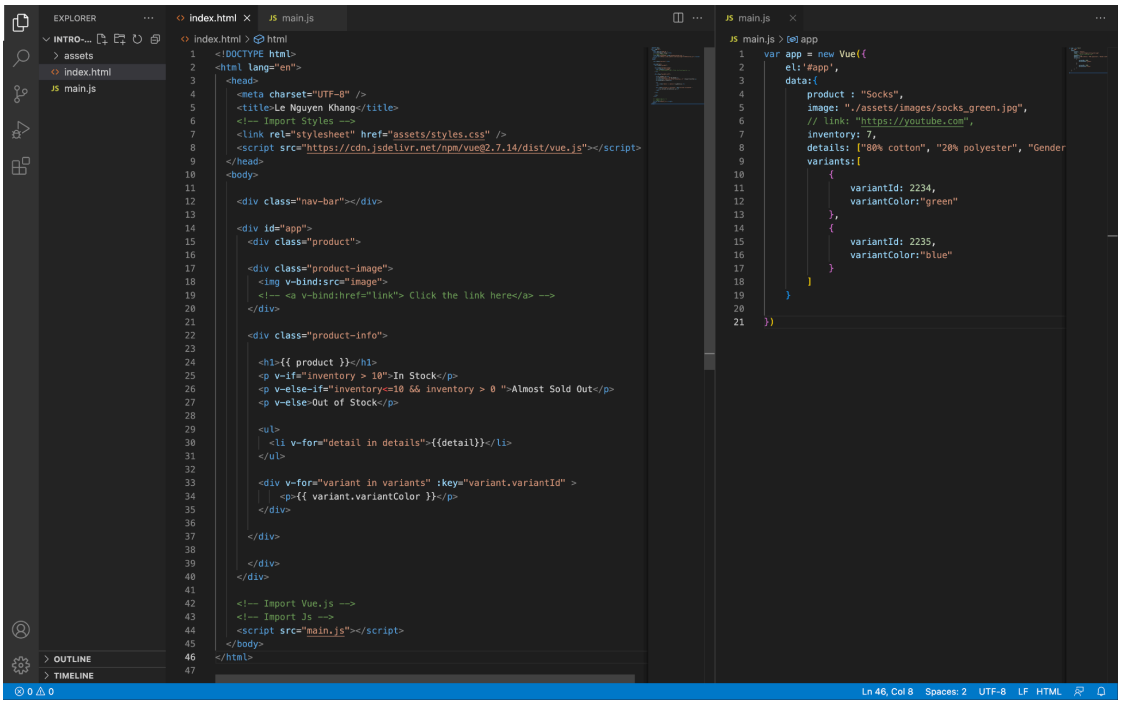
The code for Task 3

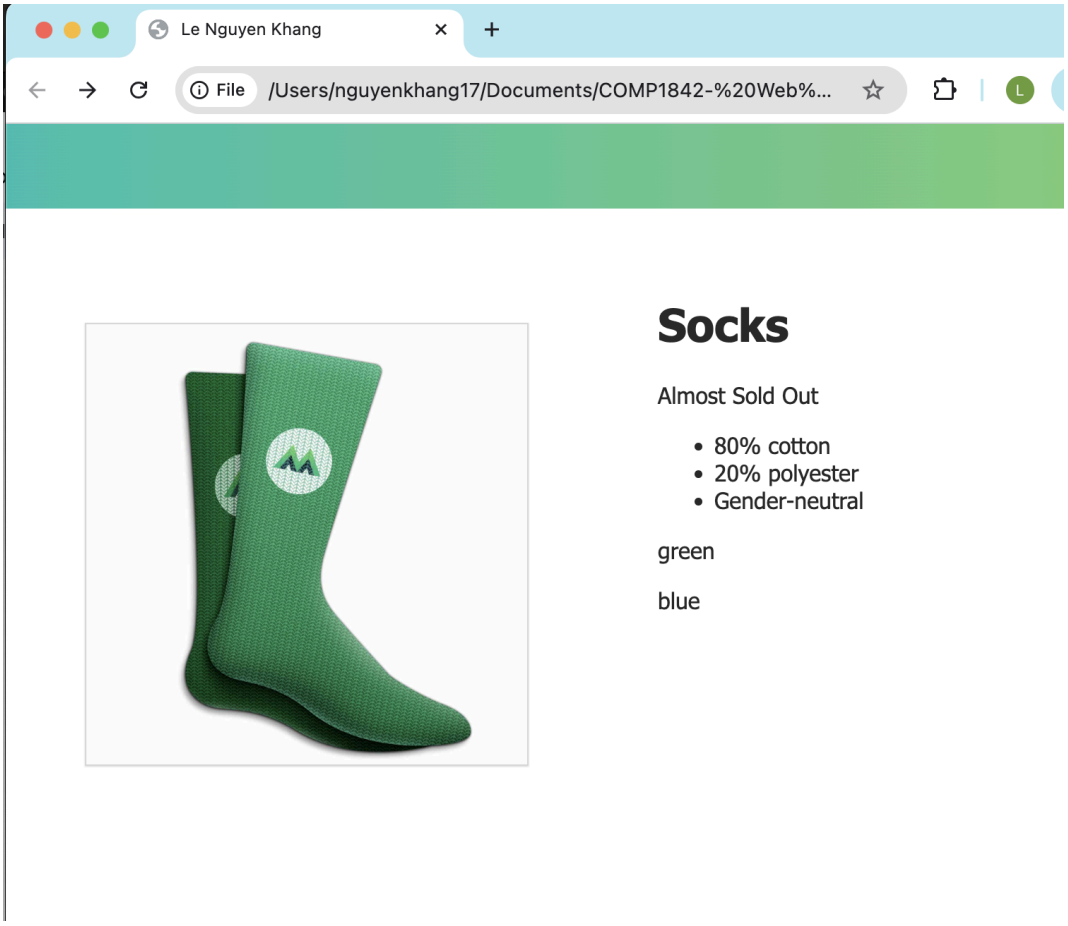




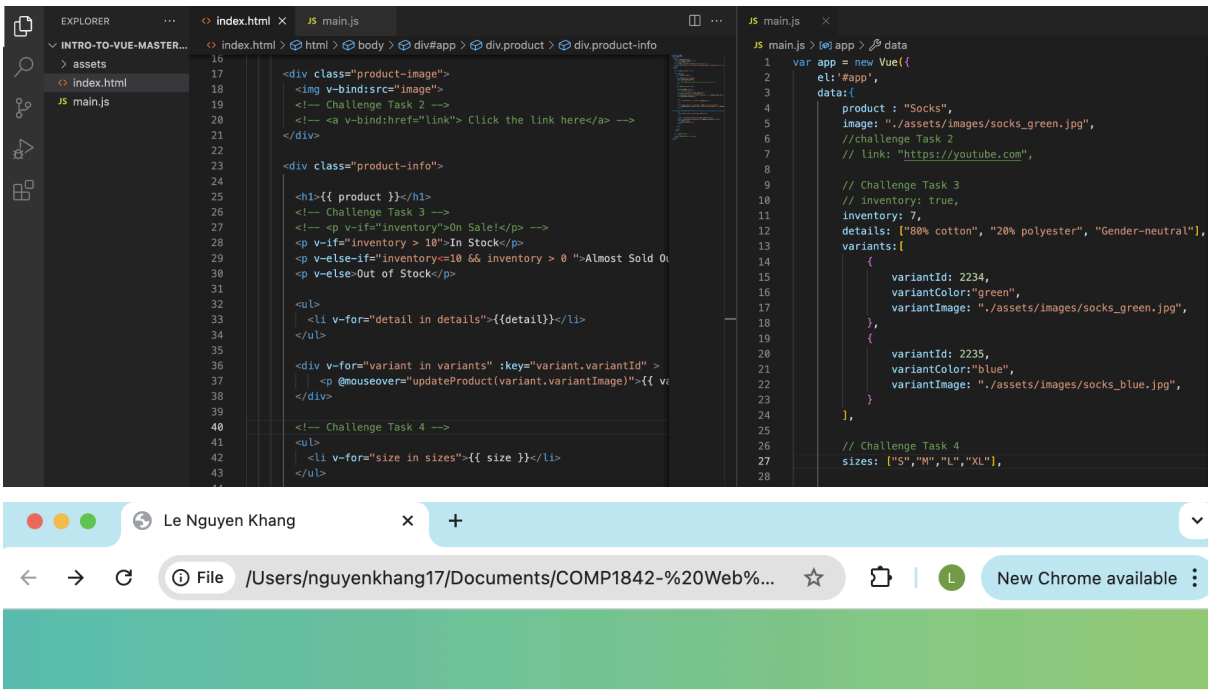
Challenge for Task 3

Task 4:





The code for Task 4



Socks

Almost Sold Out

- 80% cotton
- 20% polyester
- Gender-neutral

green

blue

- S
- M
- L
- XL

Challenge for Task 4

Understanding for Appendix 1:

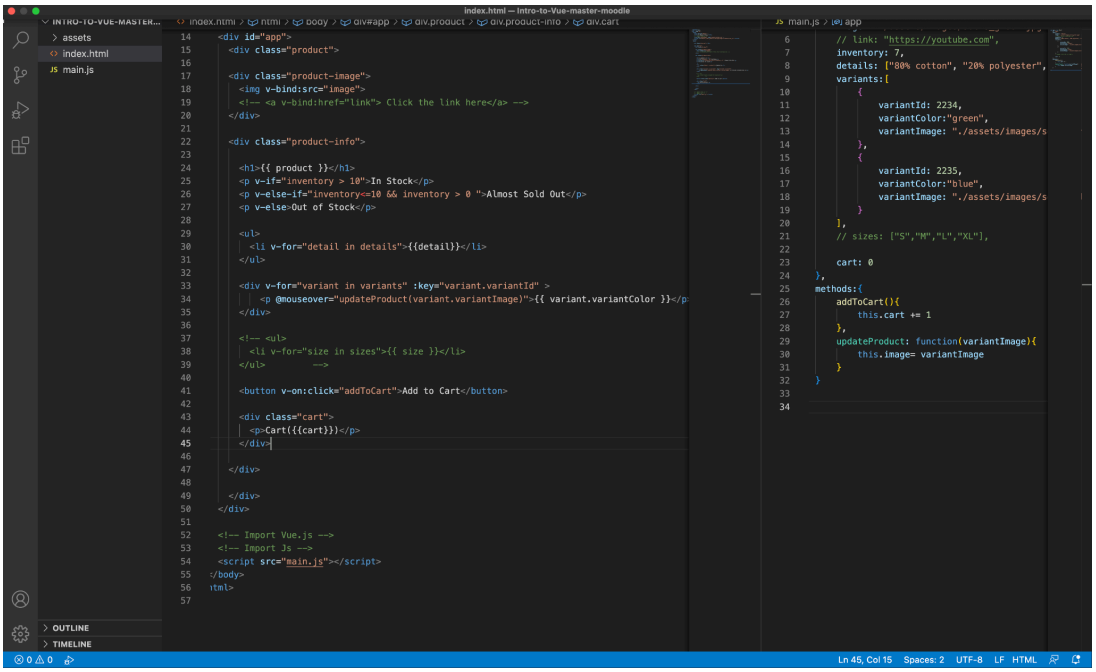
In Vue.js, creating a Vue instance connects data to the DOM, making the interface reactive. This means any changes in the JavaScript data are instantly reflected in the UI without manual updates. Vue's data binding with `{{ }}` allows dynamic data display in HTML, while v-bind enables binding attributes like `src` and `alt` in an image tag to Vue's data. Using shorthand like `:src="image"` simplifies the process, ensuring that when the image source or data changes, the UI automatically updates without reloading the page. This reactivity makes Vue ideal for building interactive, responsive web applications.

For conditional rendering, Vue uses directives like `v-if`, `v-else`, and `v-else-if` to control element display based on conditions. For example, displaying "In Stock" when `inStock` is true or showing an alternative when false. Additionally, `v-show` efficiently toggles element visibility by altering the CSS `display` property without removing elements from the DOM.

Vue's v-for directive makes list rendering straightforward by looping over data arrays and dynamically generating HTML for each item. This reduces the need for hardcoding and makes the application more scalable and maintainable. As data changes, Vue automatically updates the rendered lists, ensuring that the UI stays synchronised with the data model. Overall, Vue simplifies the development of dynamic and interactive interfaces while improving code maintainability and efficiency.

2. Appendix 2:

Task 5:



Socks

Almost Sold Out

- 80% cotton
- 20% polyester
- Gender-neutral

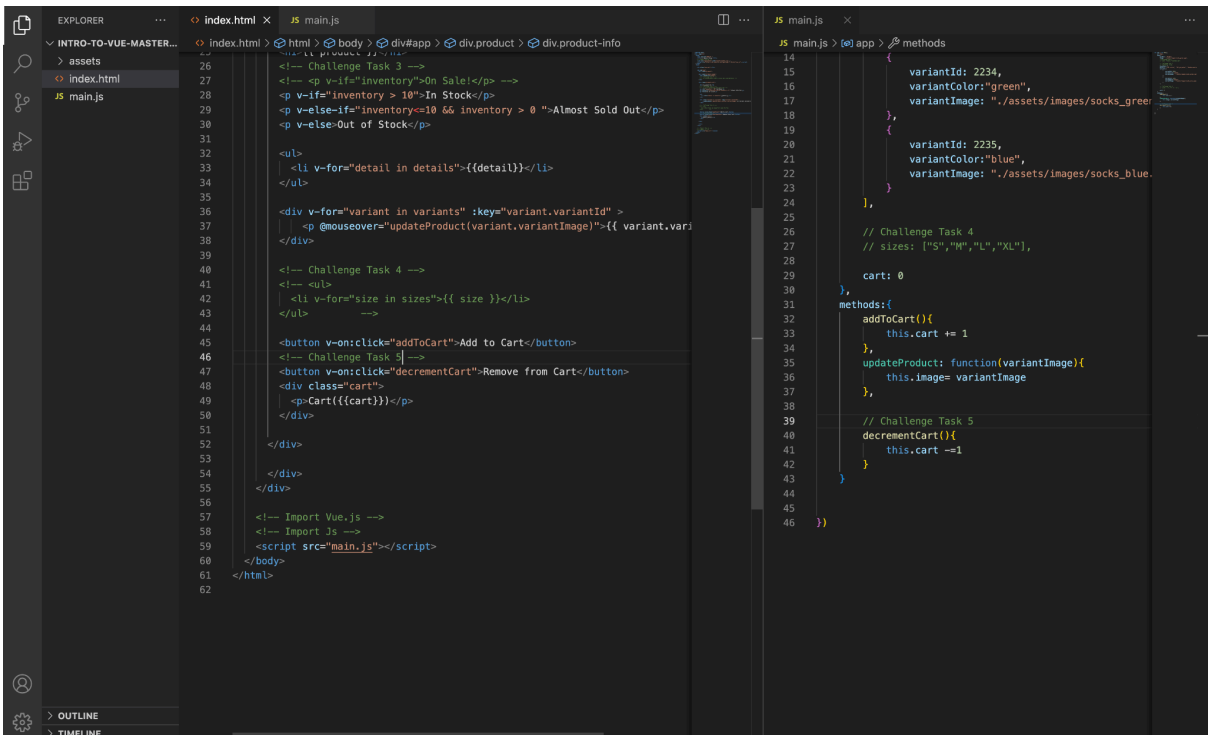
green

blue

Add to Cart

Cart(9)

The code for Task 5



Socks

Almost Sold Out

- 80% cotton
- 20% polyester
- Gender-neutral

green

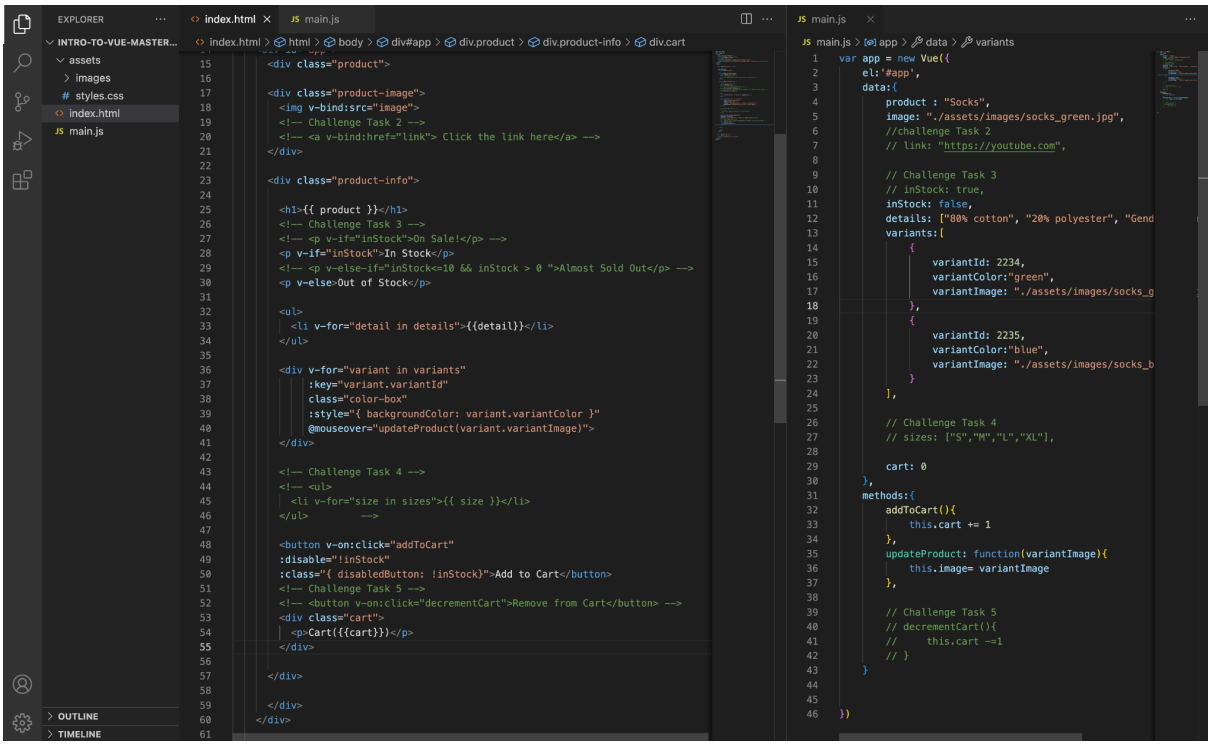
blue

Add to Cart | Remove from Cart

Cart(2)

Challenge for Task 5

Task 6:



Socks

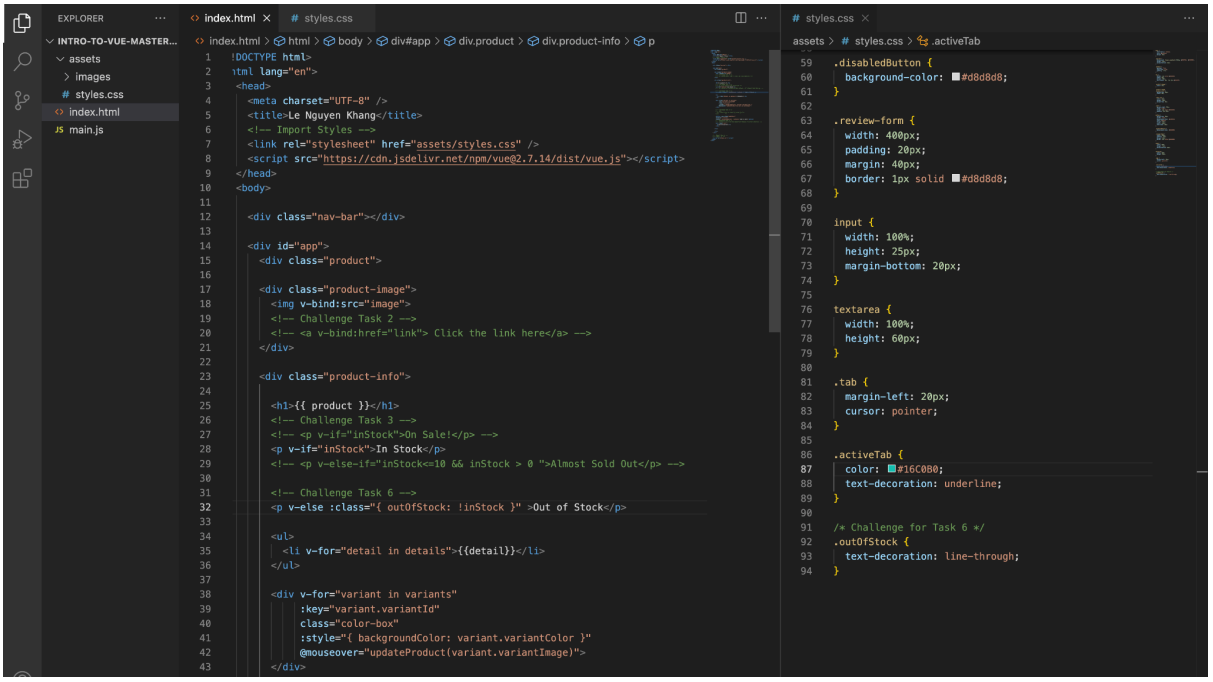
Out of Stock

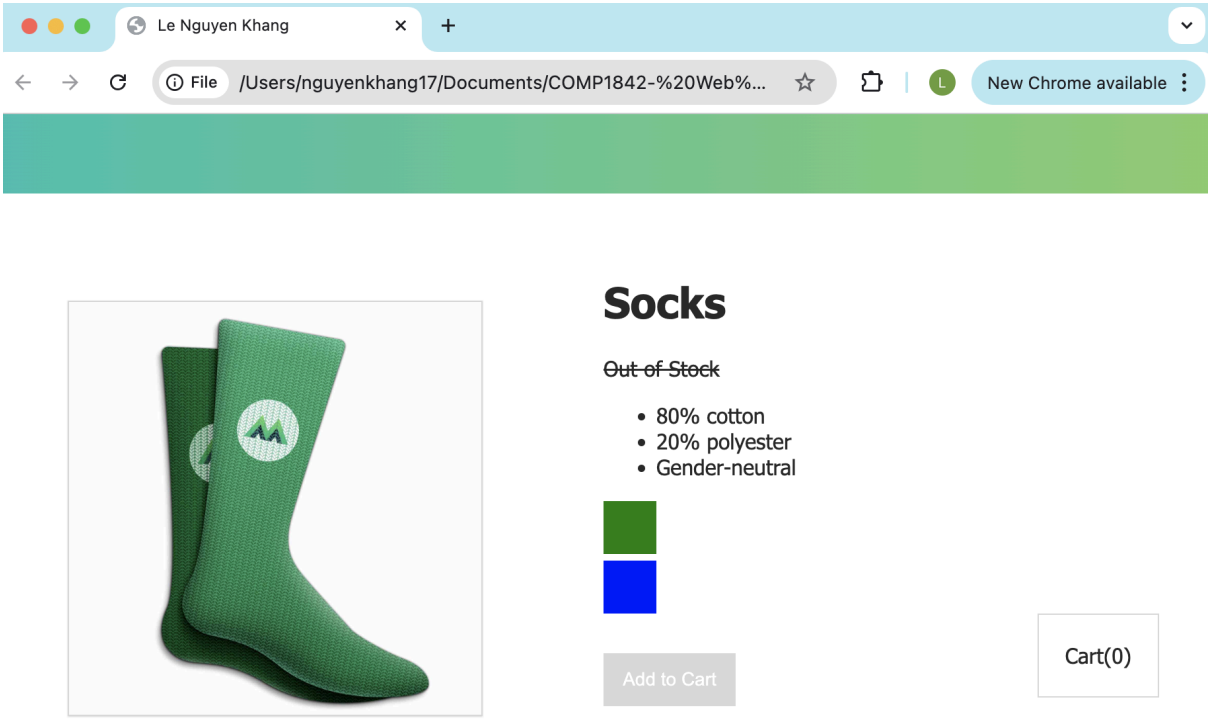
- 80% cotton
- 20% polyester
- Gender-neutral

Add to Cart

Cart(0)

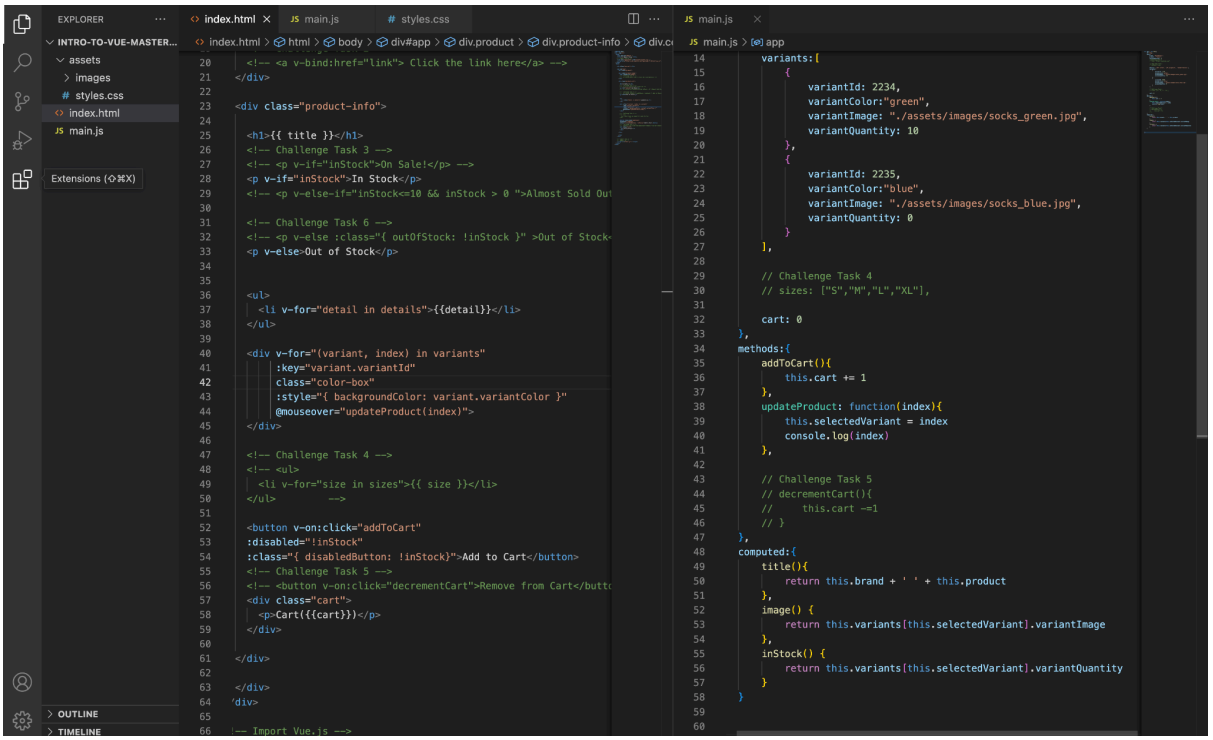
The code for Task 6

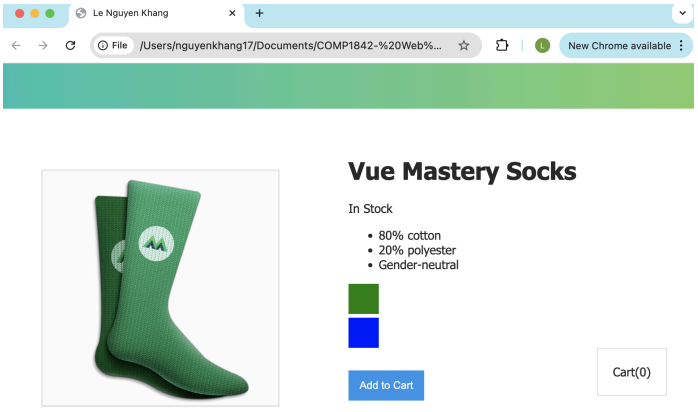


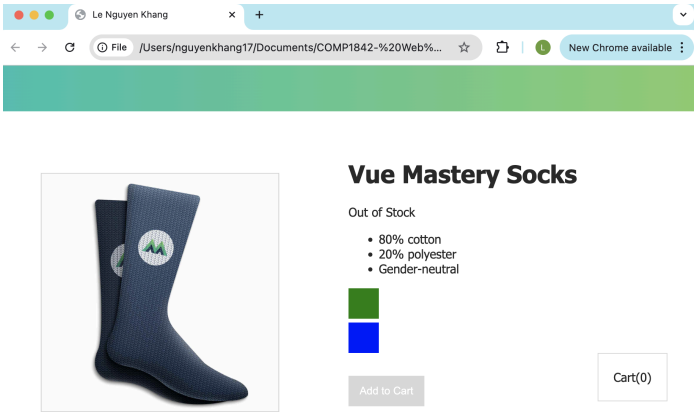


Challenge for Task 6

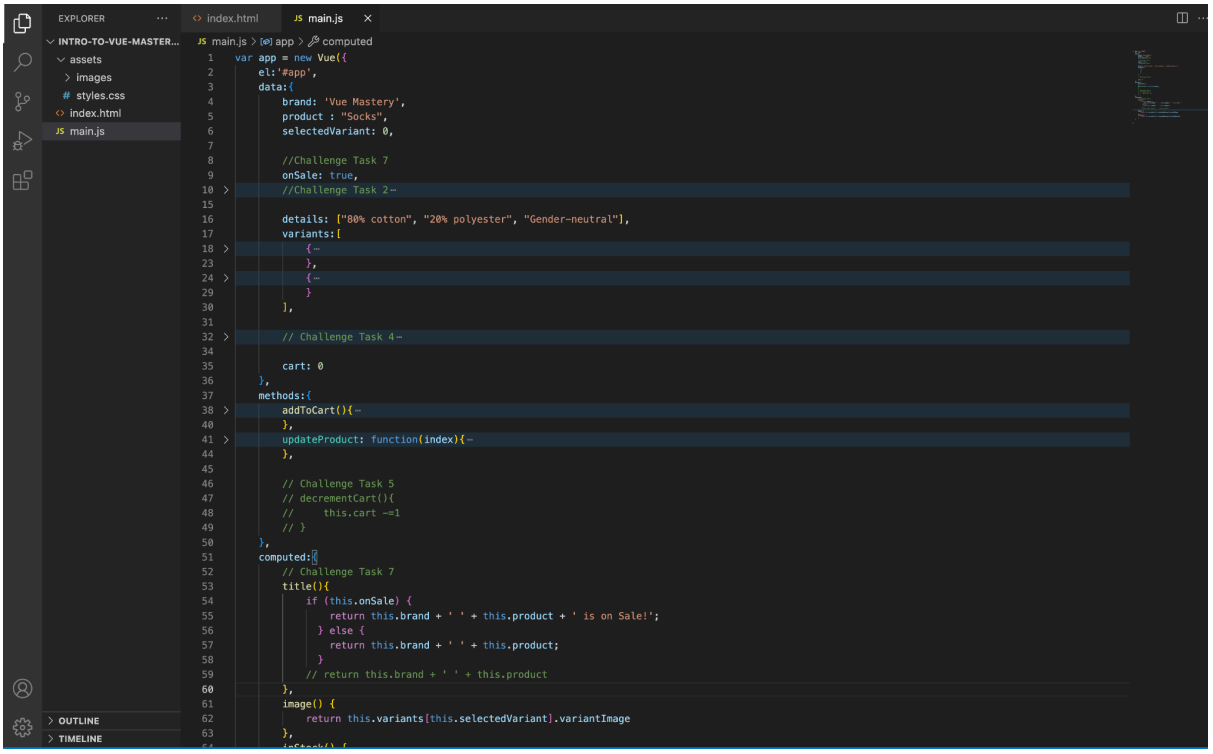
Task 7:

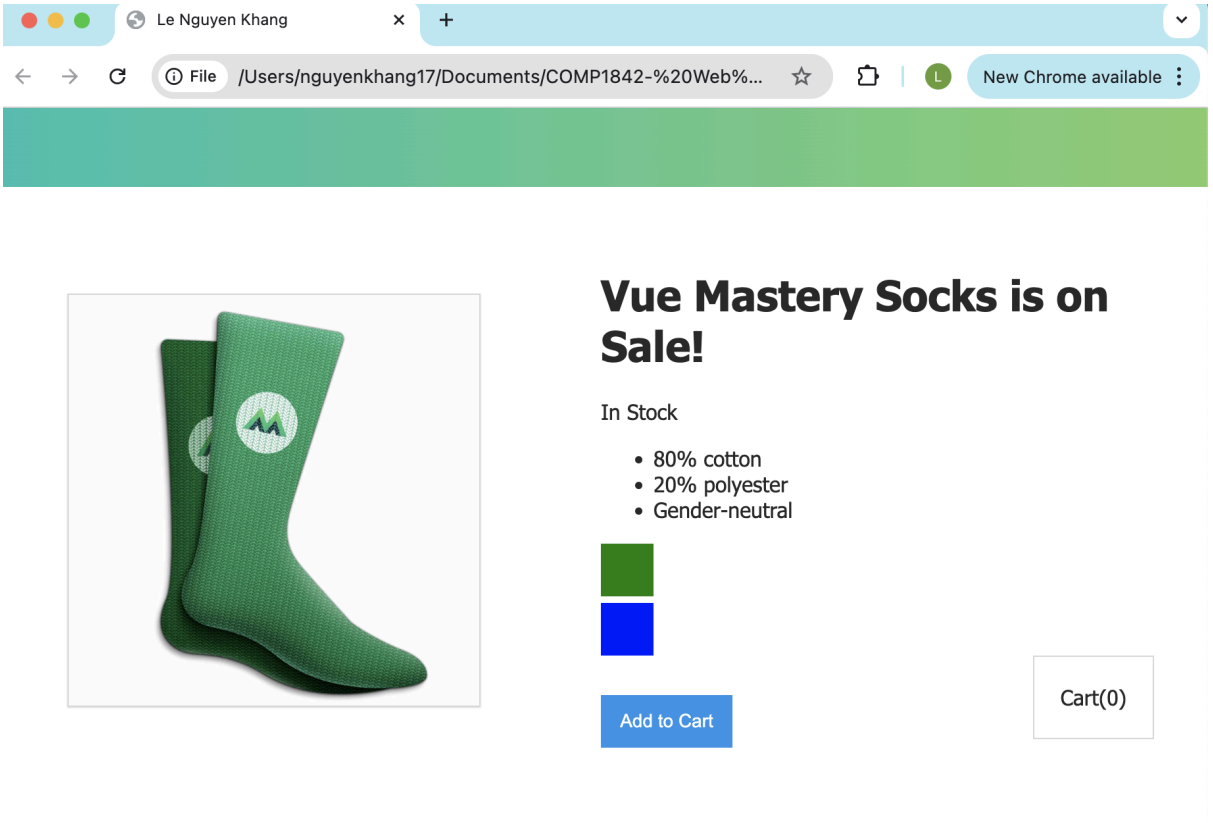






The code for Task 7





Challenge for Task 7

Understanding for Appendix 2:

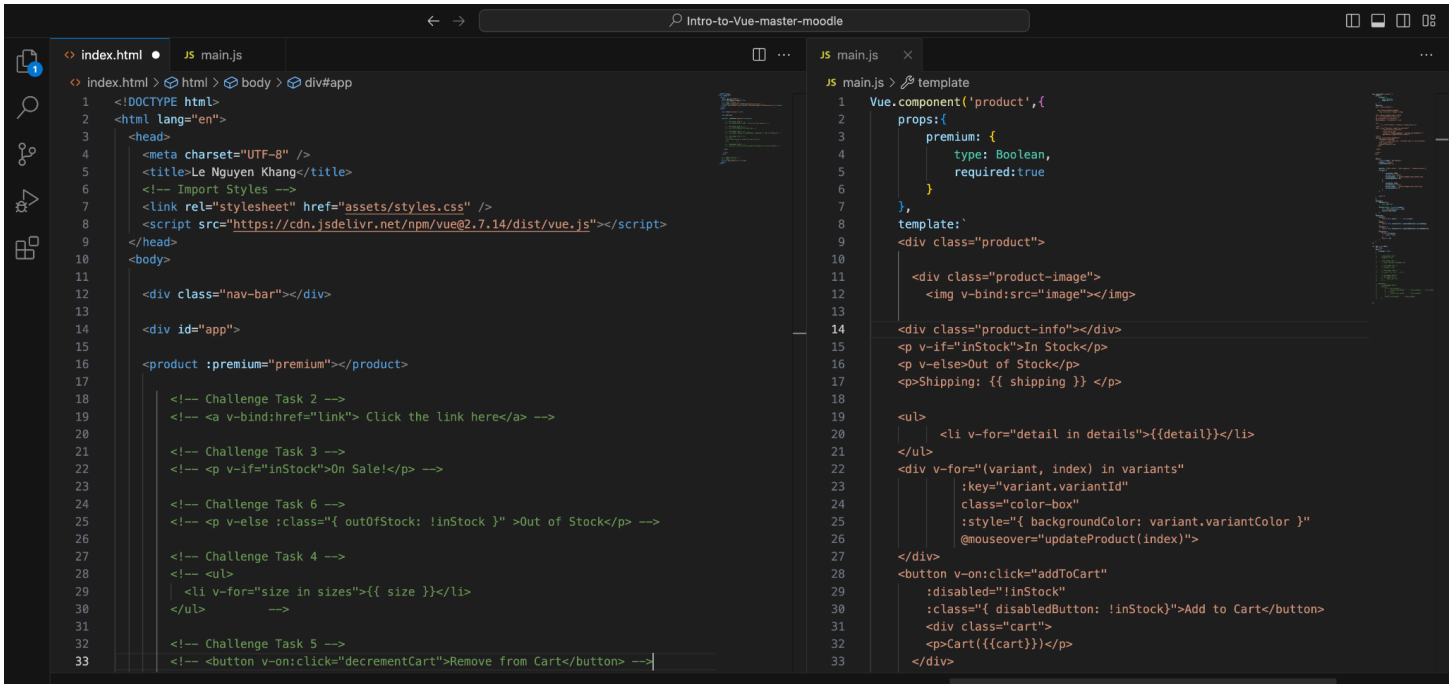
Vue's `v-on` directive is a powerful tool for handling DOM events such as clicks or mouseovers, enabling developers to trigger methods that enhance the interactivity of applications. Using the shorthand `@`, event types like `@click` can easily trigger methods such as `addToCart`, which dynamically updates the data model by referencing properties like `this.cart`. This integration keeps the application reactive and ensures that user interactions are instantly reflected in the UI.

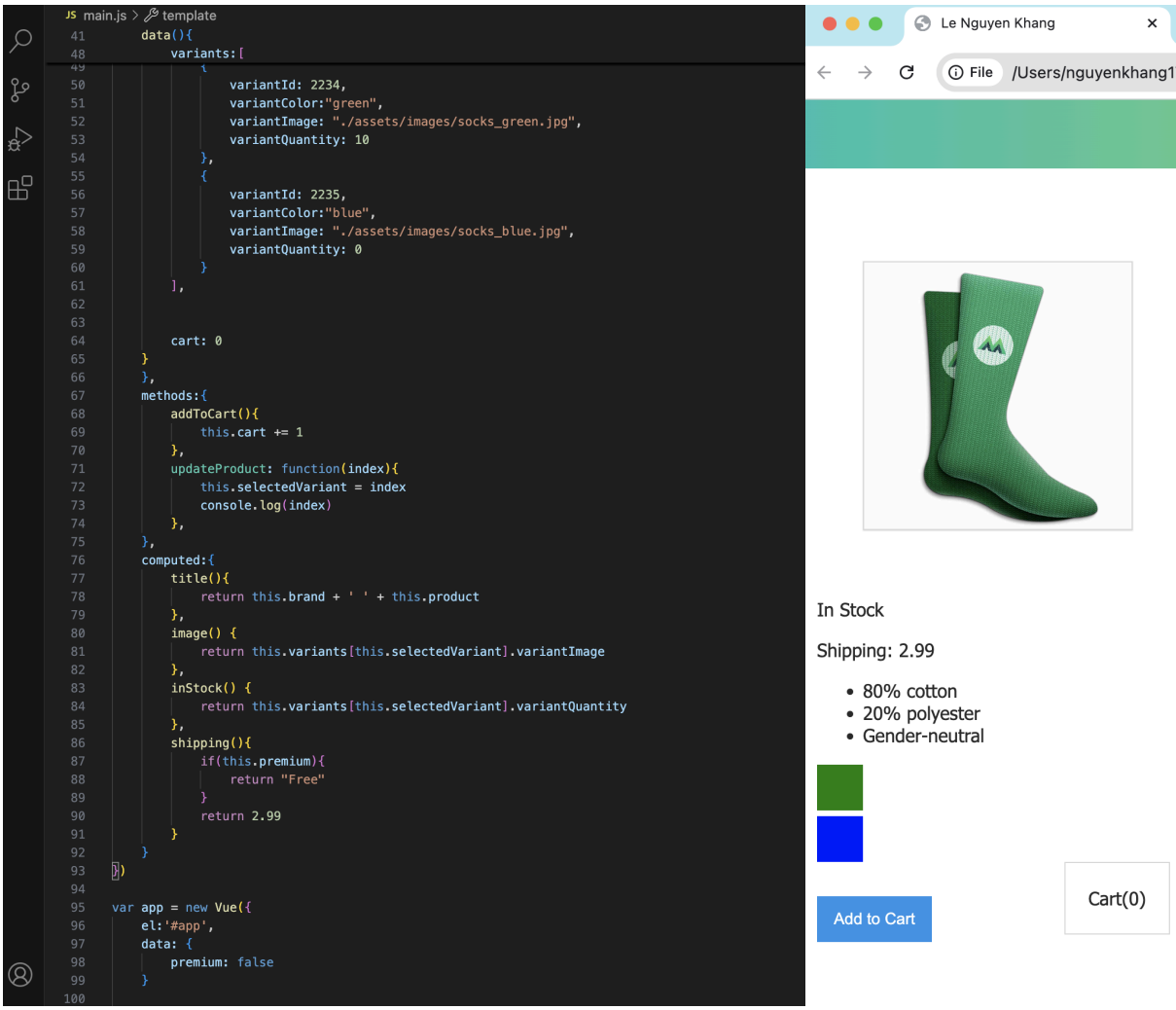
In working with class and style binding, I learned how to dynamically link data-driven styles and classes to HTML elements. For instance, using `v-bind:style` allowed me to change a `div`'s background color based on product variants, while class binding greyed out buttons when a product was out of stock. These bindings ensure that the UI adapts automatically to changes in data, offering a seamless user experience without manual DOM manipulation.

Additionally, Vue's computed properties calculate values dynamically, such as combining product brand and name into a single string, streamlining the logic and reducing redundancy in the data model. By leveraging Vue's reactivity and directives like `v-for` for list rendering, I saw how Vue simplifies complex UI updates, providing cleaner, more maintainable code for scalable applications.

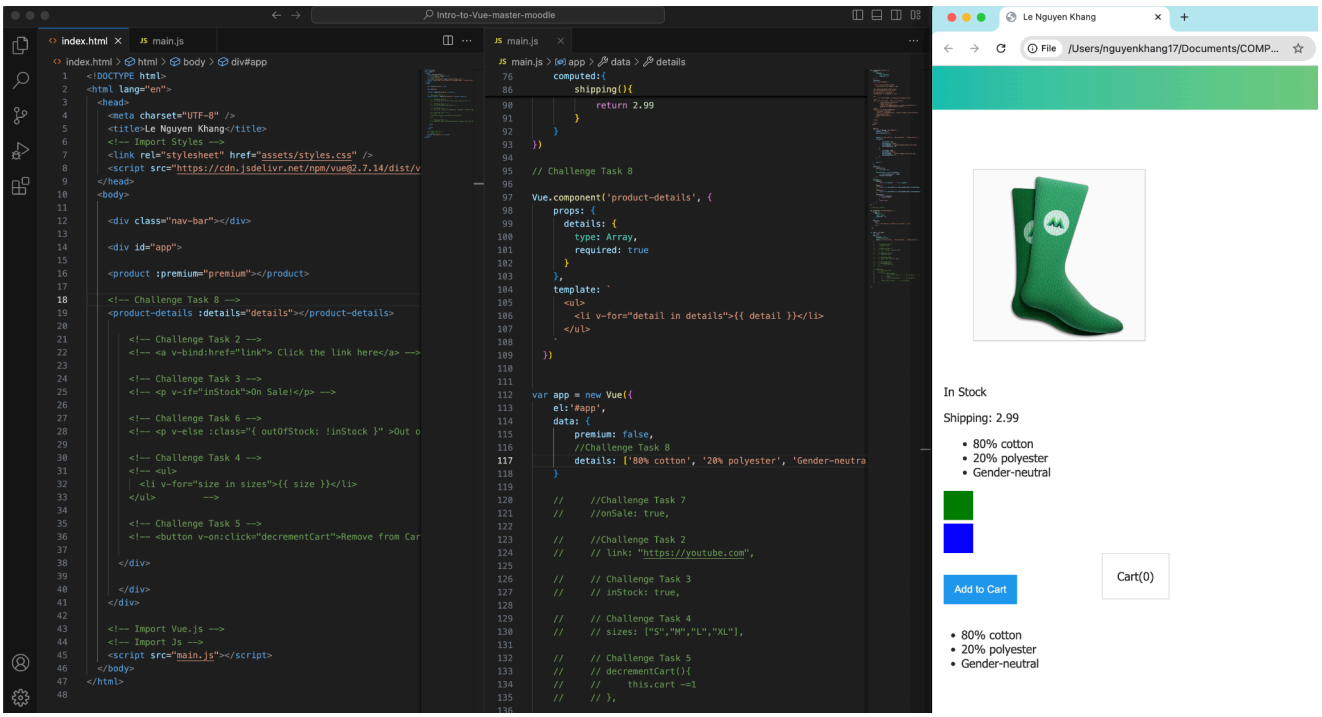
3. Appendix 3:

Task 8:



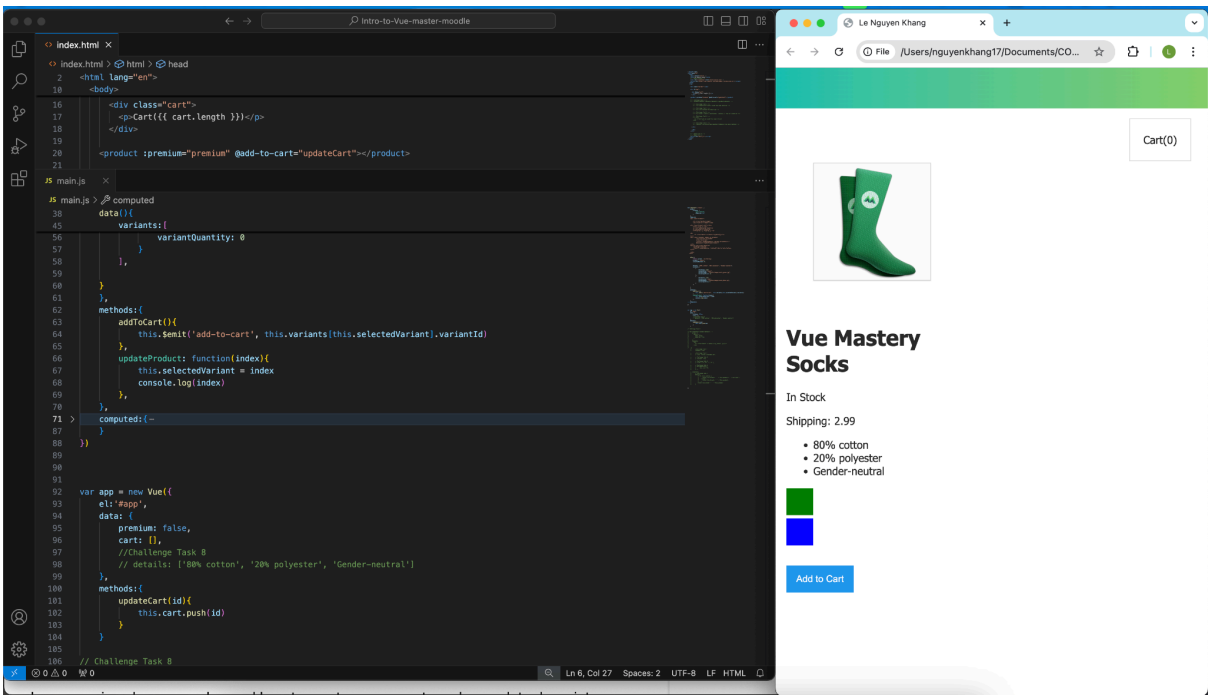


The code for Task 8

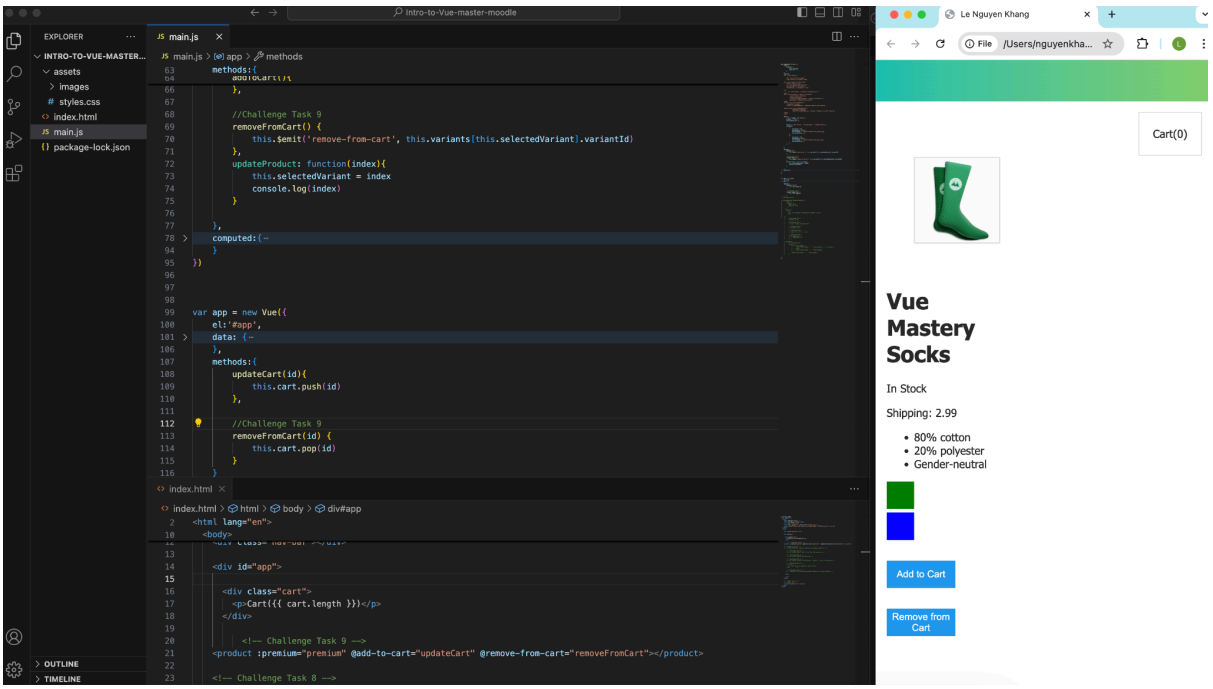


Challenge for Task 8

Task 9:

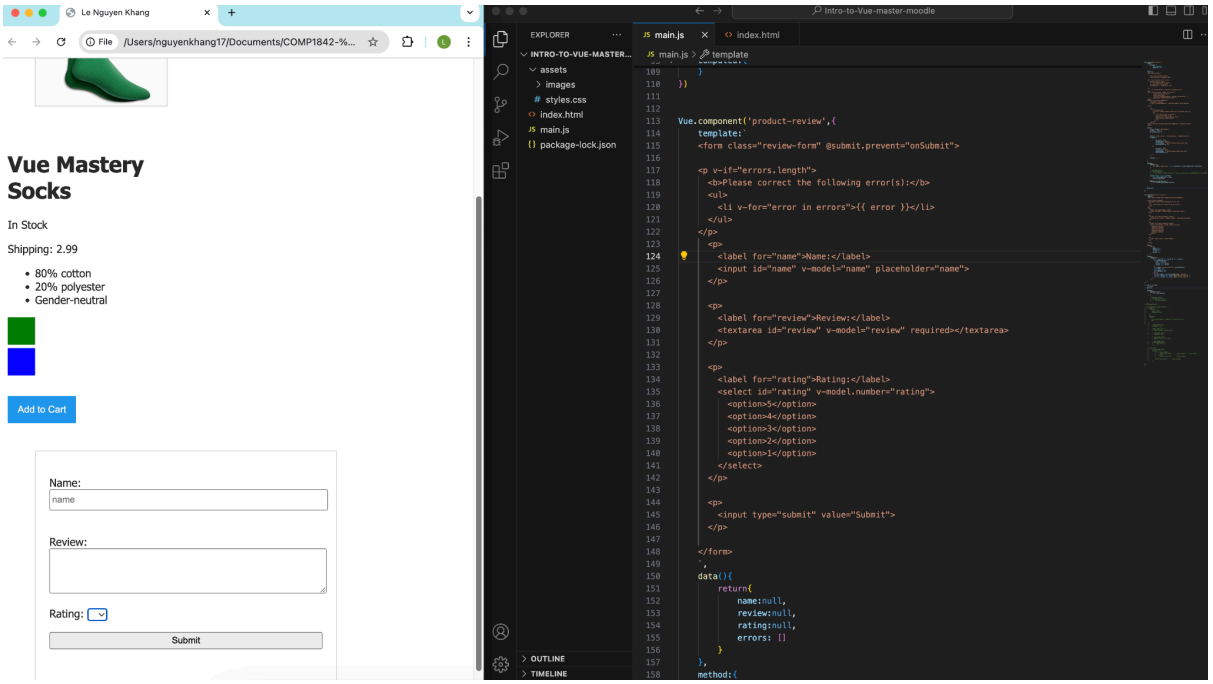


The code for Task 9

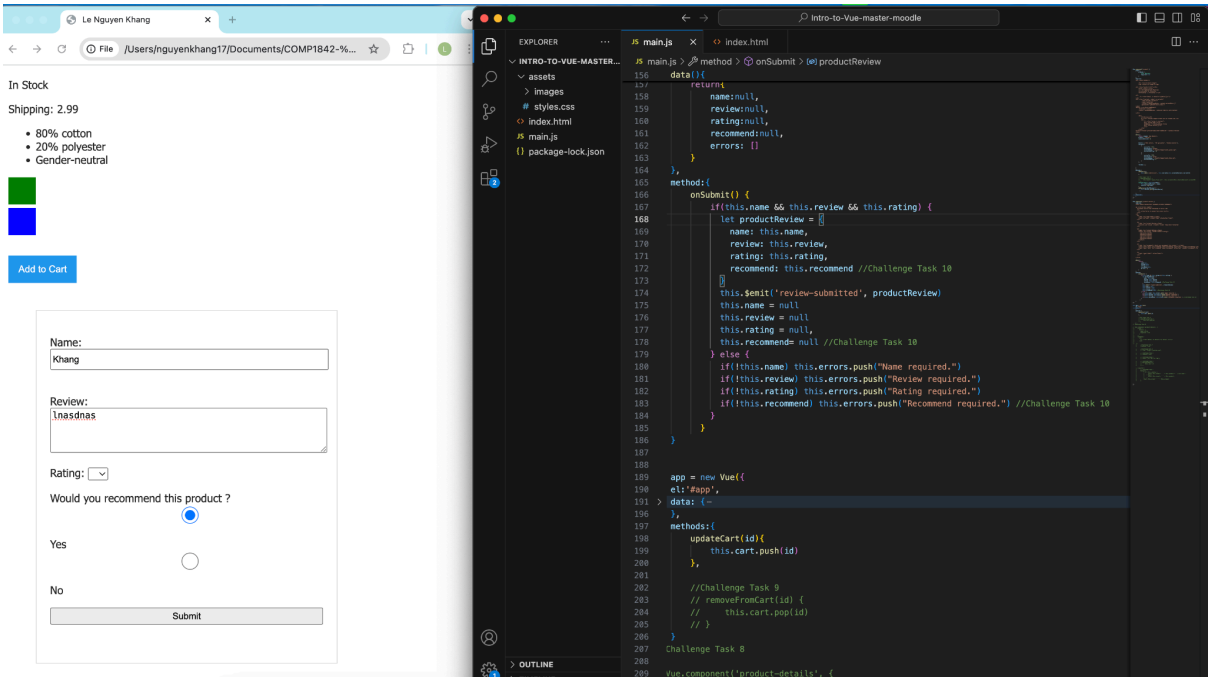


Challenge code for Task 9

Task 10:



The code for Task 10



Challenge code for Task 10

Understanding for Appendix 3:

This task focused on the use of Vue.js components to create a modular, reusable, and maintainable application structure. By encapsulating product-related functionality within a `product` component, the application became more efficient, with a cleaner structure and enhanced reusability. This approach minimises code duplication and facilitates easier updates across multiple instances.

A key aspect of this work involved managing data flow between parent and child components using "props" and "events". The parent component passed a `premium` value to the `product` component, which used it to dynamically adjust shipping costs. Computed properties were also employed to manage conditional logic, ensuring dynamic updates to the UI based on user status.

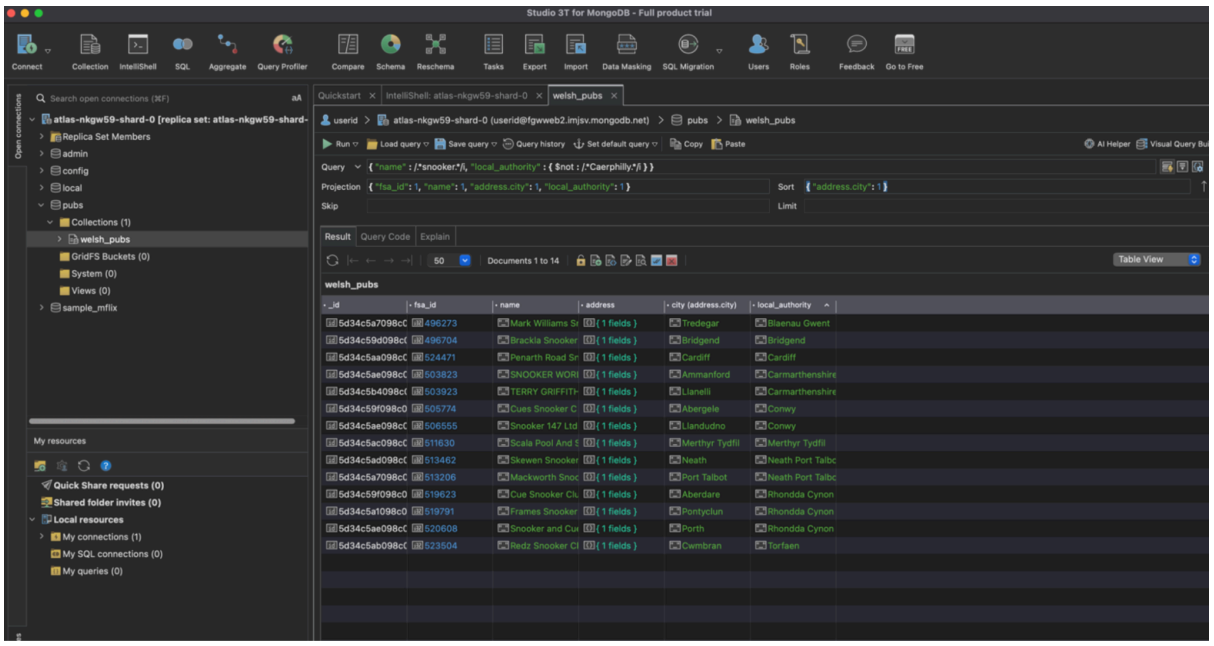
In addition to downward data flow via props, the child component communicated events like "add-to-cart" back to the parent using `$emit`. This event-driven model allowed the parent to respond to user actions while maintaining component independence and modularity.

Furthermore, two-way data binding was implemented through Vue's `v-model`, enabling user input in forms to directly modify component data. Custom form validation logic ensured required fields were filled before submission, further

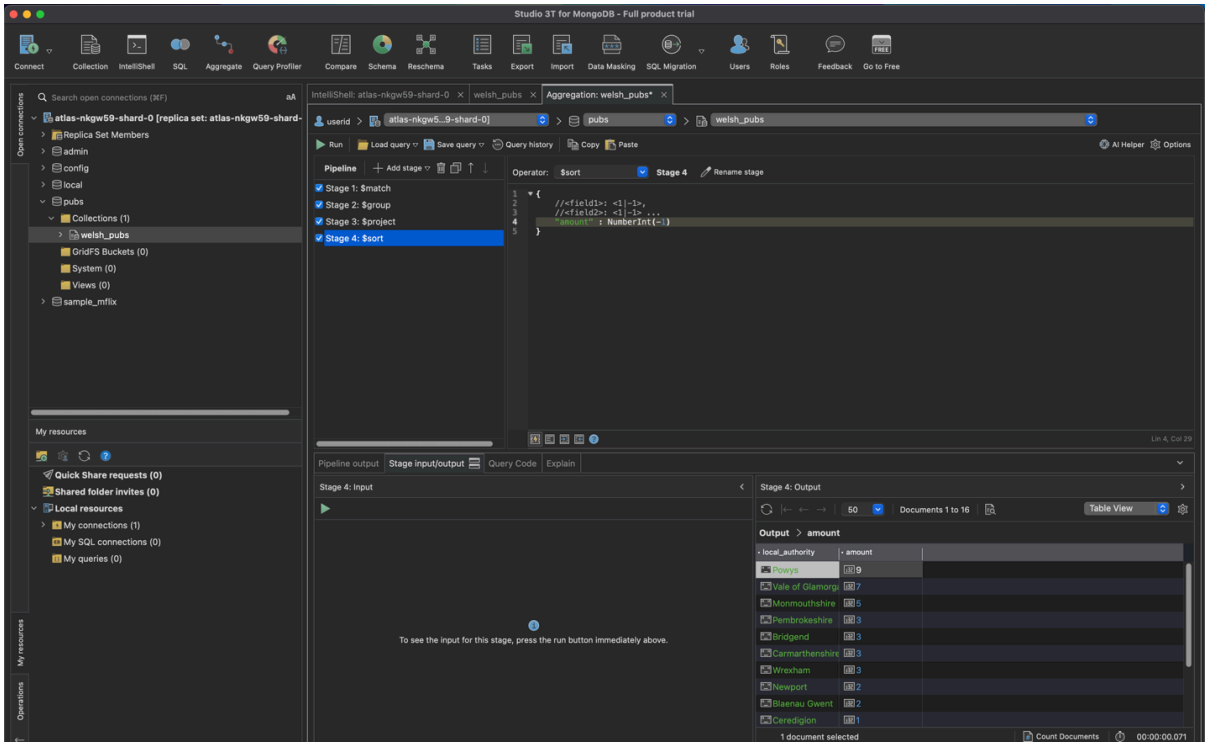
enhancing the application's interactivity and responsiveness. Overall, this work demonstrates the efficient use of Vue's features to create dynamic, interactive applications with clear separation of concerns and streamlined data flow between components.

4. MongoDB

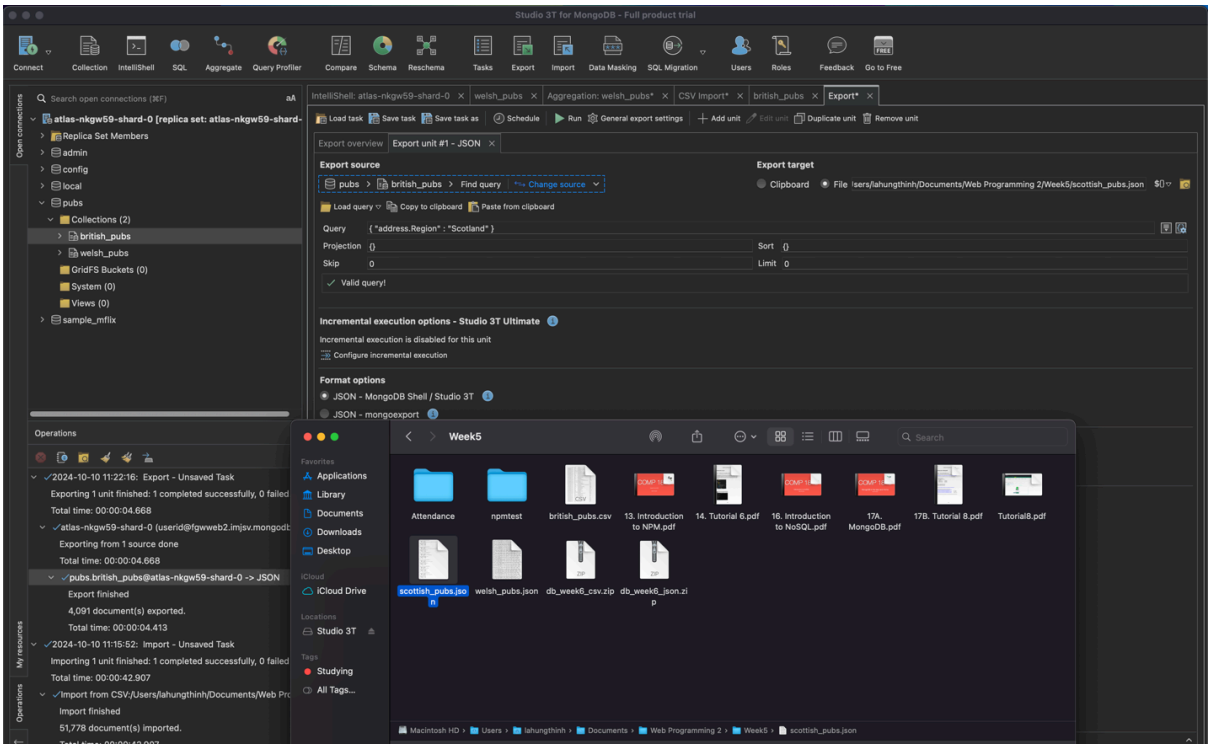
Lesson 4 exercise 1:



Lesson 4 exercise 3:



Lesson 5 exercise 3:



Lesson 6 exercise 2:

NoSQL databases, such as MongoDB, provide a flexible approach to data storage, differing significantly from traditional relational databases. Unlike the rigid table structure of relational databases, NoSQL databases are schema-less, which allows for the storage of diverse data types and structures without predefined schemas. MongoDB is a document-oriented NoSQL database that organizes data into three hierarchical levels: databases, collections, and documents. Collections hold multiple documents, which consist of key-value pairs similar to JSON objects. This document model offers versatility in storing complex, nested data and accommodates changes in data structures over time without the need for database migrations.

A key feature of MongoDB is its support for embedded documents, which allow documents to include nested arrays or objects. This enables complex relationships to be represented within a single document, reducing the need for joins, which are common in relational databases. As a result, MongoDB can manage relationships with fewer collections, enhancing query performance and simplifying data modeling.

MongoDB also includes aggregation capabilities, which are used for data processing and transformation. Its aggregation framework allows for operations like filtering, grouping, and sorting using an aggregation pipeline, which chains multiple stages to manipulate data efficiently. This approach is analogous to SQL's `GROUP BY` but

with greater flexibility and power, enabling complex data transformations and analysis. Overall, MongoDB's design makes it suitable for applications that require fast development cycles, dynamic data structures, and scalability.