COMP3322B Modern Technologies on World Wide Web

Assignment 2 (22%)

[Learning Outcomes 1, 2]

Due by: 23:59, Thursday May 19 2022

Overview

In this assignment, we are going to develop a single-page online shopping web app **iShop** using the MERN stack (MongoDB, Express.JS, ReactJS and Node.js). The main work flow of **iShop** is as follows.

- Upon loading, the sketch of the page is shown in Fig. 1. The user can click a category on the top to view products in the category.

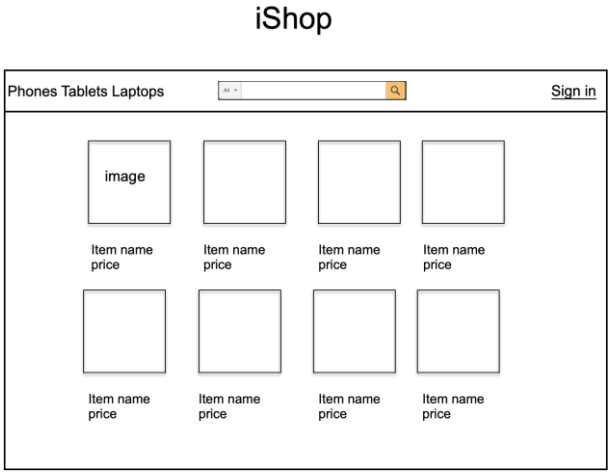

Fig. 1

- The user can search products by entering search string in the search box and clicking on the dropdown list (initially displaying "All") to narrow down the search within a specific category, as follows:

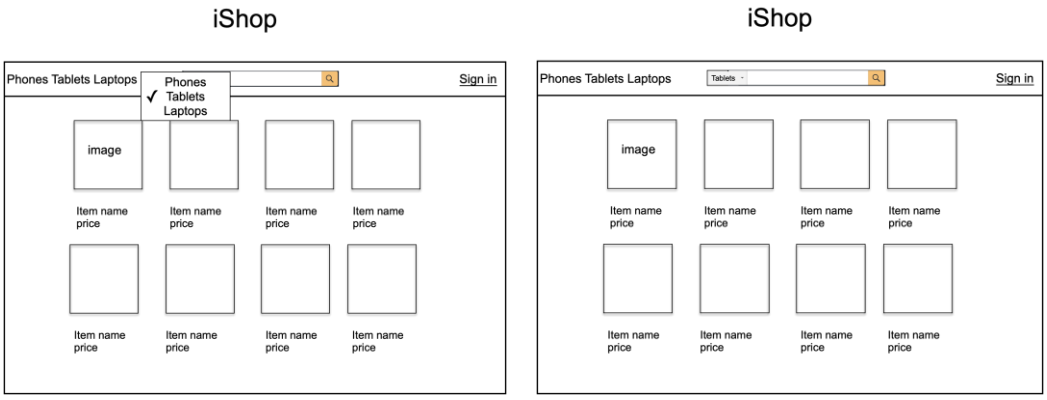

Fig. 2

- When the user clicks on a product (the area containing the item image, name and price), the page displays the detailed information of the product, as follows:

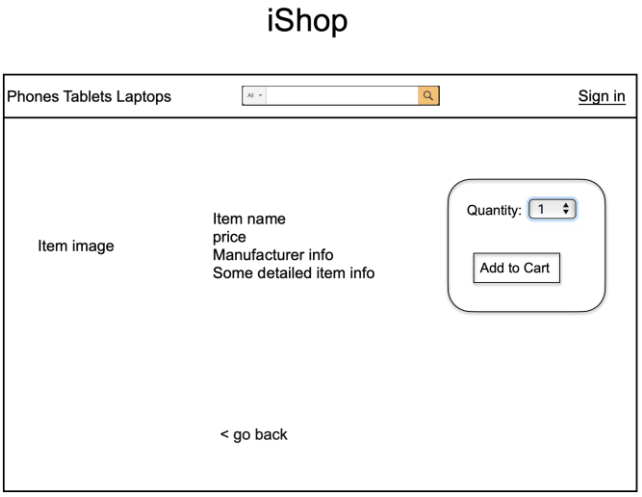

Fig. 3

The user can adjust the quantity and click "Add to Cart" to add the item(s) to its shopping cart.

After clicking "go back", the page returns to the last page that the user is viewing.

- After the user clicks the "Sign in" link in one of the page views as shown above, or clicks the "Add to Cart" button on a page view as in Fig. 3 (without user signed-in before), a sign-in page is displayed:

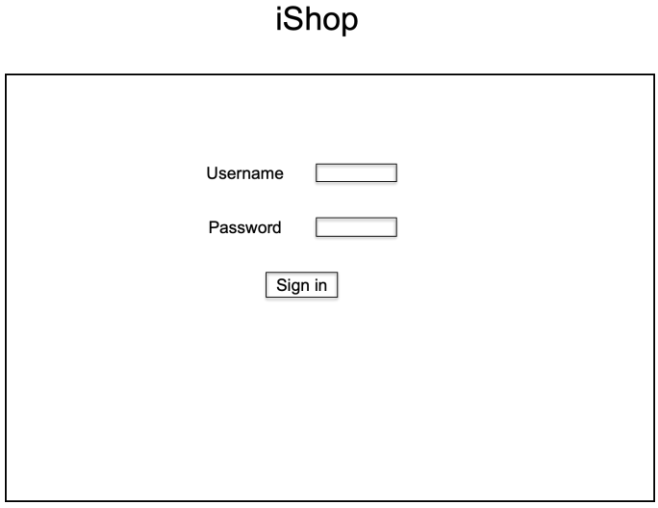

Fig. 4

User can enter its username and password and click the "Sign in" button to sign in. An error message should be prompted on top of the form in Fig. 4's view, if the sign-in fails.

- If the sign-in page was brought up when the user was adding some item(s) to the cart, after successfully signing-in, the page view goes to the following (suppose the user name is Jack), where xx indicates the actual number of items in the user's shopping cart.

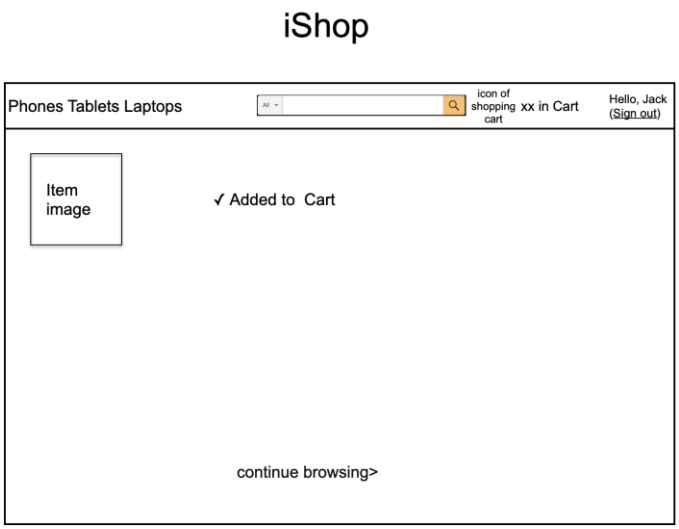

Fig. 5

After clicking “continue browsing”, the page goes back to the initial display of the web app.

- If the sign-in page was brought up by clicking “Sign in” on a page view as in Figures 1-3, after successful signing-in, the page view of Fig. 1 becomes the following, and the page views of Figures 2, 3 change similarly, i.e., the shopping cart icon/information and “Hello, xx (Sign out)” are displayed on the top-right corner of the page, instead of the “Sign in” link.

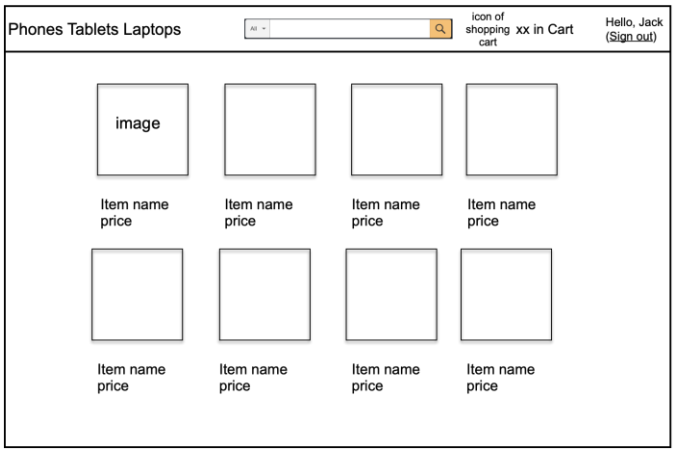

Fig. 6

- When the user clicks the area containing the shopping cart icon and “xx in Cart” on the top-right corner of a page, the shopping cart content is displayed, as shown in Fig. 7. The selected quantity of each product is displayed, and the user can change the quantity. When a quantity is changed, “Cart subtotal (xx item(s)): $xx” should change accordingly. Especially, if there is 1 item, it should be “Cart subtotal (1 item): $xx”; if there are more than one item, it should be “Cart subtotal (xx items): $xx”.

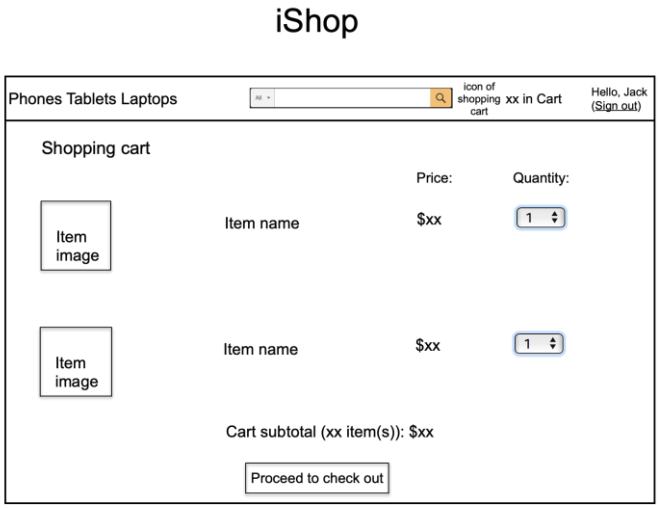

Fig. 7

Whenever the total item quantity is changed, "xx in Cart" at the top of the page should be updated accordingly.

- When the user clicks the "Proceed to check out" button, the page is as shown in Fig. 8. In our simplified shopping app, we omit the payment page, but directly display this order success page (where the total item number and money paid should be showing the actual numbers). Note that the items in the shopping cart should be cleared and "0 in Cart" should be displayed at the top of the page.

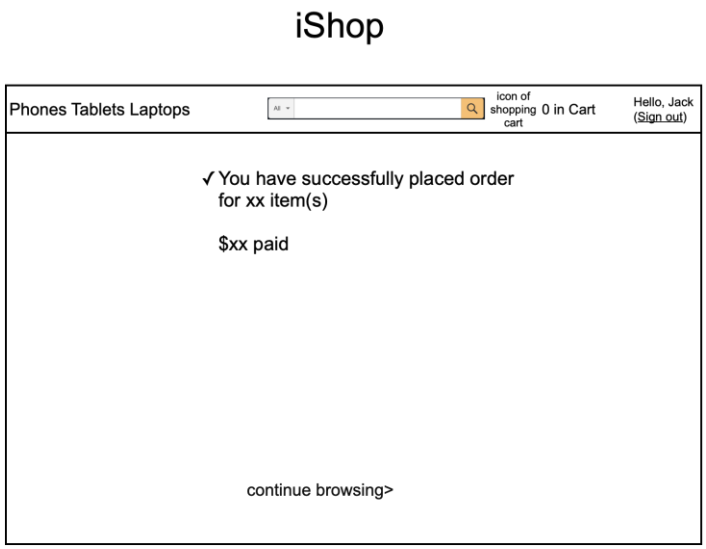

Fig. 8

When the "continue browsing>" link is clicked, the page goes back to a view as in Fig. 6, with the initial page of the web app displayed (the top-right corner shows the after-sign-in contents).

- When the user clicks the "Sign out" link in the page views as in Figures 5-8, the page displays a "Sign in" link in the top-right corner of the page replacing the shopping

cart information and “Hello xxx (Sign out)” if the previous page view is Fig. 6, or goes back to the initial page of the web app if the previous page view is Fig. 5, 7 or 8.

We are going to achieve this application by implementing code in a back-end Express app and a front-end React app.

- Express app **productservice**:
  app.js
  ./routes/products.js

- React app **shoppingapp**:
  ./src/App.js
  ./src/index.js
  ./src/App.css

Task 1. Back-end Web Service

We implement the back-end web service using Express.js. The web service is accessed at http://localhost:3001/xx.

Preparations

1. Following steps you have done in a few labs, install the Node.js environment and the Express framework, create an Express project named **productservice**, and install the needed modules.

2. Following steps you have done in a few labs, install MongoDB, run MongoDB server, and create a database “assignment2” in the database server.

(1) Insert a number of records to a productCollection collection in the database in the format as follows, each corresponding to one product.

```
db.productCollection.insert({name: 'iPhone 13', 'category': 'Phones', 'price': 8000, manufacturer: 'Apple Inc.', 'productImage': 'images/iPhone13.jpg', 'description': 'Most advanced dual-camera system ever.'})
```

For implementation simplicity, we do not store images in MongoDB. Instead, we store them in the hard disk under the ./public/images/ folder, and store the path of an image in the productCollection collection only, using which we can identify a product’s image in the images folder. Copy some product images to the images folder under ./public in your project directory.

You should insert information of at least 8 products under each category, for app testing purpose.

(2) Insert a few user records to a userCollection collection in the database in the format as follows. Here productId should be the value of _id generated by MongoDB for the respective product document in the productCollection collection, which you can find out using db.productCollection.find().

```
db.userCollection.insert({'username': 'Jack', 'password': '654321', 'cart':[{'productId': 'xxx', 'quantity': 1}, {'productId': 'xxx', 'quantity': 2}], 'totalnum': 3})
```

Task 1. Implement back-end web service (productservice/app.js, productservice/routes/products.js)

app.js **(7 marks)**

In **app.js**, load the router module implemented in ./routes/products.js. Then add the middleware to specify that all requests for http://localhost:3001/ will be handled by this router.

Add necessary code for loading the MongoDB database you have created, creating an instance of the database, and passing the database instance for usage of all middlewares.

Add the middleware to serve requests for static files in the public folder.

Add the code for enabling CORS, similar to what you did in Lab 8. In addition, specify the following CORS options when calling cors(), to allow accepting requests originated from http://localhost:3000 that carry credentials (i.e., cookies set in this app):

```
var corsOptions = {
    "origin": "http://localhost:3000",
    "credentials":true
}
```

Also load any other modules and add other middlewares which you may need, for implementing this application.

We will let the server run on the port 3001 and launch the Express app using command "node app.js".

./routes/products.js **(30 marks)**

In **products.js**, implement the router module with middlewares to handle the following endpoints:

1. **HTTP GET requests for** http://localhost:3001/loadpage?category=xx&searchstring=yy. The middleware retrieves _id, name, price and productImage of products in the

productCollection collection, whose category matches the category value and whose name contains the searchstring in the URL. If database operations are successful, the retrieved information should be sent in JSON to the client; otherwise, the error message should be sent. You should decide the format of the JSON string, and parse it accordingly in the client-side code to be implemented in Task 2.

2. **HTTP GET requests for** http://localhost:3001/loadproduct/:productid. The middleware should retrieve manufacturer and description of the product from the productCollection collection based on the productid in the URL. Send all retrieved information as a JSON string in the body of the response message if database operations are successful, and the error message if failure. You should decide the format of the JSON string to be included in the response body.

3. **HTTP POST requests for** http://localhost:3001/signin. The middleware should parse the body of the HTTP POST request and extract the username and password carried in request body. Then it checks whether the username and password match any record in the userCollection collection in the database. If no, send "Login failure" in the response message. If yes, set a cookie "userId", store this user's _id in the variable, and then retrieve totalnum of items in the cart of the user from userCollection; send totalnum in the response message if database operations are successful and the error message if failure. You should design a JSON string to send the respective information (in the above different cases) in a unified format back to the client.

4. **HTTP GET requests for** http://localhost:3001/signout. The middleware should unset the "userId" cookie, and send an empty string back to the client.

5. **HTTP GET requests for** http://localhost:3001/getsessioninfo. The middleware checks if the "userId" cookie is set: if so, it retrieves username of the current user (according to the value of the "userId" cookie) and totalnum of items in the cart of the user from the userCollection collection, and send them to the client if database operation is successful and the error message, otherwise; if the "userId" cookie is not set, send empty information to the client. Again, you should decide the format of the JSON string (a consistent format no matter which cases as in the above, and parse it accordingly in the client-side code to be implemented in Task 2.

6. **HTTP PUT requests for** http://localhost:3001/addtocart. According to the productId and quantity contained in the body of the request message, the middleware should update the cart array of the user (identified by the value of "userId" cookie) in the userCollection collection: if the productId is not contained in the cart array, add an object containing the productId and quantity into the array; otherwise, increase the quantity of the product of the productId by the received quantity. In both cases, increase totalnum in the user's record by the received quantity. Return totalnum to the client in a JSON string if success and the error

message if failure.

7. **HTTP GET requests for** http://localhost:3001/loadcart. This middleware retrieves values of productId and quantity in the cart array, as well as totalnum of the user (identified using the value of "userId" cookie) from the userCollection collection. Then retrieve the name, price and productImage of products of those productId's from the productCollection collection. Send all retrieved information as a JSON string in the body of the response message if database operations are successful, and the error message if failure. You should decide the format of the JSON string to be included in the response body.

8. **HTTP PUT requests for** http://localhost:3001/updatecart. The middleware should update the cart array of the user (identified by the value of "userId" cookie) in the userCollection collection, by updating the quantity of the product of the productId carried in the request body by the quantity in the request body, and updating totalnum in the user's document accordingly. Return totalnum to the client if success and the error message if failure.

9. **HTTP DELETE requests for** http://localhost:3001/deletefromcart/:productid. The middleware should delete the object {'productId': 'xx', 'quantity': yy}, where xx is the productid value in the URL, from the cart of the user's document in the userCollection collection (using _id of the user from the userId cookie), and update totalnum in the user's collection accordingly. Return totalnum to the client if success and the error message if failure.

10. **HTTP GET requests for** http://localhost:3001/checkout. Empty the cart array of the user's document in the userCollection collection (using _id of the user from the userId cookie), and set totalnum to 0 in the user's document. Send an empty string to the client if success and the error message if failure.

Task 2 Front-end React App

Implement the front-end React app. The app is accessed at http://localhost:3000/.

Preparations

Following steps in Lab 8, create a React app named **shoppingapp** and install the jQuery module in the React app.

Implement the React app (shoppingapp/src/index.js, shoppingapp/src/App.js, shoppingapp/src/App.css)

index.js **(3 marks)**

Modify the generated **Index.js** in the ./src folder of your react app directory, such that it renders the component you create in **App.js** in the "root" division in the default **index.html**.

App.js **(50 marks)**

App.js should import the jQuery module and link to the style sheet **App.css**.

Design and implement the component structure in **App.js**, such that the front-end page views and functionalities as illustrated in Figures 1-8 can be achieved. You can decide the components to create and HTML elements to use, as long as the page views follow the sketches in Figures 1-8.

Note that each AJAX HTTP request from the client side should set withCredentials property (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/withCredentials) to be true, e.g, when using $.ajax, add xhrFields: {withCredentials: true} into the request configuration object. This allows cross-site access-control requests to be made using credentials such as cookies, and also allows cookies in the response message to be set properly.

Hints:

1. You can use conditional rendering to decide what component to be displayed on the page view. For example, in the root component that you are creating in **App.js**, you may use a state variable to indicate if a user has signed in or not, and then render the components presenting the top part of the respective page view accordingly. You can use state variables to decide whether it is the product list, individual product info, shopping cart info, etc., that should be displayed in the main part of the page view accordingly.

2. **Initial page load.** When the browser loads http://localhost:3000/, an AJAX HTTP GET request is sent to the web service server to retrieve http://localhost:3001/loadpage?category=&searchstring=, and another AJAX HTTP GET request is sent for http://localhost:3001/getsessioninfo. Using data received from the server side, a page as shown in Fig. 1 or Fig. 6 should be displayed (depending on whether the user has signed in). You can assume the three categories of products (Phones, Tablets, Laptops) displayed at the top of the page are fixed in this app. You should order the retrieved products in alphabetic order of their names on the page. _id of each product should be stored (e.g., in an attribute of the element displaying the product information) for future usage. You can assume all retrieved products are displayed on the same page and display four products in one row.

3. **Product display according to category.** When a product category among the three categories at the top of the page is clicked, an AJAX HTTP GET request is sent for http://localhost:3001/loadpage?category=xx&searchstring=, where xx is the category value. Using data received from the server side, a page containing the products in this category will be shown, following the sketch in Fig. 1 or Fig. 6 (depending on whether the user has signed in). Order the products in alphabetic order of their names. _id of each product should be

stored (e.g., in an attribute of the element displaying the product information) for future usage.

4. **Product display according to search string and category.** When the search button is clicked, an AJAX HTTP GET request is sent to the web service server for http://localhost:3001/loadpage?category=xx&searchstring=yy, where yy is the search string user has entered in the search box and xx is the category value that user has selected on the dropdown list (if no selection, the category is default to "All" and the category value should be an empty string in the URL). Using data received from the server side, the products in this category, whose names contain the search string, will be shown, following the sketch in Fig. 1 or Fig. 6 (depending on whether the user has signed in). Order the products in alphabetic order of their names. _id of each product should be stored (e.g., in an attribute of the element displaying the product information) for future usage.

5. **Display detailed product information.** On a page view as in Fig. 1 or Fig. 6, when the block area containing a product's image, name and price is clicked, send an AJAX HTTP GET request for http://localhost:3001/loadproduct/:productid, where productid should be the productId of this product. Using received data from the server side, display the product's detailed information following the sketch in Fig. 3 (show before- or after-sign-in information on the top right part of the page, depending on whether the user has signed in or not).

6. **Go back.** When the "<go back" link on the page view as in Fig. 3 is clicked, the page view goes back to the page that the user was visiting before clicking a product to view its details.

7. **Sign in**. When the "Sign in" link on a page view as in Figures 1-3 is clicked, show the sign-in page as given in Fig. 4. When the user clicks the "Sign in" button on the page, check if both the username and password input textboxes are non-empty: if so, send an AJAX HTTP POST request for http://localhost:3001/signin, carrying the input username and password in the request body; otherwise, show an alert popup box stating "You must enter username and password". If a "Login failure" or other error message is received, the page view remains as in Fig. 4, except for displaying "Login failure" or the other error message above the input boxes in addition; otherwise, display the page that the user was visiting before clicking "Sign in", while showing the shopping cart information and "Hello xx (Sign out)" on the top right part of the page, based on the input username and totalnum received.

8. **Add to cart.** When the "Add to Cart" button on a page view as in Fig. 3 (before user has signed in) is clicked, a sign-in page as in Fig. 4 is displayed. When the user clicks the "Sign in" button on the page, check if both the username and password input textboxes are non-empty: if so, send an AJAX HTTP POST request for http://localhost:3001/signin, carrying the input username and password in the request body; otherwise, show an alert popup box stating "You must enter username and password". If a "Login failure" or other error message is received, the page view remains as in Fig. 4, except for displaying "Login failure" or the other error message above the input boxes in addition. Otherwise, send an AJAX HTTP PUT request for http://localhost:3001/addtocart, containing key-value pairs of productId of the

product and the quantity selected in the previous page view as in Fig. 3 (where the user clicks the "Add to Cart" button); then display a page view as in Fig. 5, using the input username and totalnum received from the server side.

When the "Add to Cart" button on the page as in Fig. 3 (after the user has signed in) is clicked, send an AJAX HTTP PUT request for http://localhost:3001/addtocart, containing key-value pairs of productId of the product and the quantity selected. Display a page view as in Fig. 5, using the input username and totalnum received from the server side.

9. **Load the shopping cart**. When the user clicks the block area containing the icon of the shopping cart and "xx in Cart" on the top-right part of the page view as in Figures 5-8, send an AJAX HTTP GET request for http://localhost:3001/loadcart. Display a page as in Fig. 7 using data received from the server side. Store productId of each product in the cart (e.g., as an attribute of the element displaying the product information) for future usage. The numbers in the select elements should range from 0 to a large number at your choice.

10. **Update product quantity in the shopping cart.** When the user has changed the number in a select element on the shopping cart display page (Fig. 7), if the number is not 0, send an AJAX HTTP PUT request for http://localhost:3001/updatecart, containing key-value pairs of productId and the quantity selected of the corresponding product. Use the totalnum received from the server side, and the total price computed with product prices and quantities in the current shopping cart, to update the page view.

11. **Delete a product from the shopping cart**. When the user has changed the number in a select element to 0 on the shopping cart display page (Fig. 7), send an AJAX HTTP DELETE request for http://localhost:3001/deletefromcart/:productid. Use the totalnum received from the server side, and the total price computed with prices and quantities of products remaining in the shopping cart, to update the page view (the deleted product should be removed from the page view).

12. **Check out**. When the "Proceed to check out" button on the shopping cart display page (Fig. 7) is clicked, if the shopping cart is empty, do nothing; otherwise, send an AJAX HTTP GET request for http://localhost:3001/checkout. If an empty string is received from the server side, display a page as shown in Fig. 8.

13. **Continue browsing**. When the "continue browsing>" link on a page view as in Fig. 5 or Fig. 8 is clicked, send an AJAX HTTP GET request for http://localhost:3001/loadpage?category=&searchstring=. Using data received from the server side, display a page as shown in Fig. 6, i.e., display the initial page view of the app.

14. **Sign out.** When the "Sign out" link on a page view as in Figures 5-8 is clicked, send an AJAX HTTP GET request for http://localhost:3001/signout. When an empty string is received from the server side, the page displays "Sign in" at the top-right corner of the page, if the previous page view is Fig. 6, or goes back to the initial page view of the app if the previous

page view is Fig. 5, Fig. 7 or Fig. 8. Note that more AJAX requests may need to be sent to retrieve needed information for page display.

App.css (**5 marks**)
Style your page views nicely using CSS rules in **App.css.**

Other marking criteria:

(**5 marks**) Good programming style (avoid redundant code, easy to understand and maintain). You are encouraged to provide a **README** file to let us know more about your programs.

Submission:

You should zip the following files (in the indicated directory structure) into a **yourstudentID-a2.zip** file

productservice/app.js
productservice/routes/products.js
shoppingapp/src/index.js
shoppingapp/src/App.js
shoppingapp/src/App.css

and submit the zip file on Moodle:
(1) Login Moodle.
(2) Find “Assignments” in the left column and click “Assignment 2”.
(3) Click “Add submission”, browse your .zip file and save it. Done.
(4) You will receive an automatic confirmation email, if the submission was successful.
(5) You can “Edit submission” to your already submitted file, but ONLY before the deadline.