ISYS2160 LocalShop Implementation Steps

Feature 1 - Product Catalogue

Built a database model for products using the Mongoose library, with the following 8 attributes: name, description, brand, category, price, stock, image, and user.

Applied necessary integrity constraints to each attribute. For instance, we declared

- Name, description, brand, and category to be non-empty strings with reasonable max length.
- Price to be positive floating point number
- Stock to be positive integer
- User to be valid user-id (foreign key referencing id of a row in user model)

Enforced other attribute-specific constraints such as name having to be unique, and added timestamp for when the product is created or edited. Made all 8 fields required.

Built backend RESTful APIs to handle CRUD operations on Product using node and express. While doing so, we

Handled edge cases by validating

- whoever is requesting create, edit, and delete operation has an admin role.
- product Id received for edit and delete operation is valid (exists in DB)
- all required fields are included in the request body and are in valid type and size which conforms to the schema defined for Product model.

Connected to the database and executed appropriate queries / statements.

- ViewAll operation: fetch all products from DB
- View operation: fetch specific product from DB by its id.
- Create operation: creates new product in DB with received user input (name, description, brand, category, price, stock) and auto-generated Id.
- Edit operation: edit existing product in DB by its id.
- Delete operation: delete existing product in DB by its id.

Tested the backend APIs are working as intended using Postman

Developed web-based user interface to communicate with REST APIs. To post to, or fetch from

Create & Edit product API:

- Created HTML form containing appropriate input fields to collect user input in the desired type / form and designed it to match color scheme and gave enough room for input fields
- Made the form accessible only via admin portal
- Added slot for error message if submission fails due to error raised from server, that notifies admin which input field does not conform to the schema requirements

ViewAll product API:

- Created a bar-like React component that displays product details compactly and doesn't render images. Designed the component to have full-width and max height of 3em. We will be referring to this component as "product-bar" in this report.
- Created a box-like React component that displays product details and renders images.

Designed the component to have responsive dimensions according to screen size and render images with aspect ratio of 4 : 3. We will be referring to this component as “product-box” in this report.

View product API:

- Linked “product-box” to a separate tab that displays the full details of the clicked product.
- Displayed clicked product, re-using “product-box” but customised to have full-screen width.

Update & Delete product API:

- Customised “product-box” to display two buttons - update & delete besides product details when the component is used from the admin portal.
- Linked update button to the form where admin can edit and submit new attribute values for the product.
- Made delete button to make request for delete API with product Id in request body

Created a table-like React component to list all categories. The categories were initially set up directly interfacing with DBMS (Mongo Compass). We’ll refer to this component as “category-filter”.

Placed “category-filter” at the very left of the product-browsing page. Made each category checkable. Filtered products to be selectively displayed based on which categories are checked within the component.

Feature 2 - Shopping cart

Built a database model for the shopping cart using the Mongoose library, with the following 2 attributes: products & user.

Applied necessary integrity constraints to both attributes. We declared

- products to store a list of objects that contains Id and quantity of product added to the cart.
- user to store the user Id of whoever owns the cart

Discussed if using an array for the first attribute, products, is a good Idea. Decided we’ll keep the implementation because MongoDB excels at storing semi-structured data and the practice doesn’t introduce any duplicate data since only one shopping cart is created and assigned per user. It could even increase efficiency of the query searching for details of the products added to the cart.

Built backend RESTful APIs to handle read, insert-product, update-product, and clear operations on Cart using node and express. While doing so, we

Handled edge cases by

- allowing only one cart per user to simplify implementation, and hence, create and assign the cart upon user registration.
- rather calling update operation if insert operation is requested for a product but already exists in the shopping cart.

Connected to the database and executed appropriate query / statements

- Insert-product operation: create an anonymous object with productId and quantity specified in the request body; then, append the object to products attribute of the shopping cart owned by the user making the request.
- Update-product operation: search through products attribute of the shopping cart owned by the user making the request; then, update the quantity of the product that has id matching the

productId specified in the request body.
- Clear operation: set the products of the shopping cart owned by the user making the request to an empty array.
- Read operation: return products of the shopping cart owned by the user making the request

Tested the backend APIs are working as intended using Postman

Developed web-based user interface to communicate with REST APIs. To post to or fetch from Insert-product & Update-product cart API:

- Customised “product-box” to contain a HTML form with a single numeric input field and submit button. Numeric input field is for setting the quantity of the product the user wants to add to cart or update before proceeding to checkout.
- Changed the button’s value to “Add to Cart” for requesting insert-product cart API.
- Changed the button’s value to “Update” for requesting update-product cart API.

Clear cart API:

- Created button “Clear cart” that makes request for clear cart API upon click.

Read cart API:

- Mapped the result (list of objects {productId, quantity}) to be displayed using “product-box”

Placed the components in the appropriate space in the managing-cart page. Made all products added to the cart be displayed using the “product-box” component at the top of the page. Placed the “clear shopping cart” button at the bottom of the page, alongside the total price of the cart that’s calculated with vanilla javascript and simple maths.

Feature 3 - Secure Payment Gateway	
1	Researched into what 3rd party library could be the best option for integrating into our system, to ensure users' payments are securely encrypted and handled. Decided to use Stripe. Created a Stripe account and got a unique API key we could use for the integration at the development and testing stage.
2	Integrated Stripe API to perform secure checkout by redirecting users to the external url provided by Stripe API. The Stripe API mandates the use of HTTPS for every transaction so that users’ credentials and credit card information are encrypted and protected from external threats. Built function to handle successful payment and order placement. The function first creates a new order in DB by transferring products list from cart and empties the cart. There’s no need to set the status of the order created (placed) since it defaults to “Processing”. (more on feature 4)
3	Tested that the integration is working with Postman. Because we wouldn’t want to burn real money to just test if our application’s payment gateway is working, we referred to the official Stripe documentation to find credit card details provided by Stripe for testing purposes. We’ve found that it’s possible to simulate successful payments by utilising the card number 4242 4242 4242 4242, along with any valid expiry date (not expired), cardholder name, and CVC.

4	Integrated Stripe page to our UI by linking the “Move to Checkout” button in manage-cart page to Stripe’s secure payment gateway. The gateway’s link is valid only once when the link is open for the first time. Set up a redirection route for the user to be redirected back to our site when payment is complete and successful. Gave the user feedback if their payment was successful or not, on the redirected page alongside a link to products browse page (index page).

Feature 4 - Order Management System	
1	**Built a database model for the order** using the Mongoose library, with the following 6 attributes: products, total price, total quantity, status, user, and order date. Applied necessary integrity constraints to each attribute. For instance, we declared - Products to store a list of objects that contains id and quantity of product, representing products ordered for a single transaction - Total price and total quantity to be floating point numbers - Status to be one of “Processing” (default), “Shipping, and “Delivered” - User to be valid user-id (foreign key referencing id of a row in user model) - Order date to be current timestamp indicating when the order was placed Reasoning behind total price and total quantity was to reduce workload of client-side or server-side machines by reducing extra arithmetics to compute them everytime request is made to view the order.
2	**Built backend RESTful APIs** to handle create, update, and read operation on Orders using node and express. While doing so, we Handled edge cases by validating: - update operation is only accessible to user with admin role - the new status for order update operation equals one of the following three options: Processing, Shipping, Delivered. - order Id received for read and update operation is valid (exists in DB) - all the required fields for create operation: that is, user Id & product list with quantity specified for each product are included in the request body Connected to the database and executed appropriate queries / statements. - Create operation: Create new Order in DB with received input specified in req body - Update operation: Update the status attribute of the order - Read operation: Respond to client with details of the order identified by the order Id included in the request parameter. - ReadAll operation: Respond to normal users with all orders placed by the user. Respond to admin with all orders in the server.
3	**Tested the backend APIs** are working as intended using Postman
4	**Developed web-based user interface** to communicate with REST APIs. To post or fetch from: Create order operation: Created feedback page for successful or failed payment attempts. User input isn’t needed to create the order itself, because making the payment implicitly creates the

	order with appropriate user information. Update order operation: Created a table-like React component similar to “product-bar” that displays order details including order id, date, status, total price and quantity. The component also has a dropdown menu with 3 status options and a submit button to update the status of the order. The component was made to be only accessible via the admin panel. View order operation: Reused the structure of the above component to display order details to users (whoever placed the order). It was made to display more elements as users would appreciate the image content, what total cost for each product is, etc.

Feature 5 - User Authentication & Authorisation	
1	**Built a backend model for the user** using the Mongoose library, with the following 5 attributes: userId, name, email, password, and role. Applied necessary integrity constraints to each attribute. For instance, we declared - Name, and email to be non-empty strings with reasonable max length - Email to contain @ symbol, and be unique - Password to be in encrypted - Role to default to “user”, and haven’t provided any means to change the role of user via API calls to prevent any security issue, since an admin account has so much power. To register as an admin, one would have to first create an account, and interface with the database directly to change the field.
2	**Built backend RESTful APIs** to handle CRUD operations on Product using node and express. While doing so, we Handled edge cases by validating - all required fields are included in the request body prior to calling registration or login: that is, email and password for both operations and name for registration. Connected to the database and executed appropriate queries / statements - Registration: store the name, email and encrypted password to the DB - login: Look for the matching email and password (encrypt the password using the same algorithm and salt, then compare it with whatever’s stored in the database). If the user matching credentials is found, issue a JWT token, configure (expiry date, etc), then return it as a http-only cookie (read only) to the user. - Logout: Remove the issued token for the specific user that’s been stored in the server. If a token is expired or removed from the server’s cookie storage, the user attempting to access the system will be rejected to perform whatever operation that requires userId since the middleware function operated prior to calling most APIs will raise an error. (see below for more details)
3	**Built middleware functions** that can perform authorisation. intended to be called before calling backend APIs that require the user to be signed in. For instance, when a request is made to create a new product, the router first calls this authorization middleware function. The authorisation first checks if the jwt token sent from client(web) matches one of the tokens issued in the past 7 days (the valid token duration set up for our system). If the token is found

	invalid, a new authorisation error is raised with the status code of 401.
4	**Developed web-based user interface** to communicate with REST APIs. To post to, or fetch from Register and Login API: - Created HTML form containing appropriate input fields to collect user input in the desired type / form and designed it to match color scheme and gave enough room for input fields - Made the form accessible only when users are not logged in - Added slot for error message if submission fails due to error raised from server, that notifies user which input field does not conform to the schema requirements Logout API: - Created button that calls logout API that only appears at the top right corner of the screen when the user is logged in.
5	Used local storage to use cookie storage for storing the the JWT token issued by the server