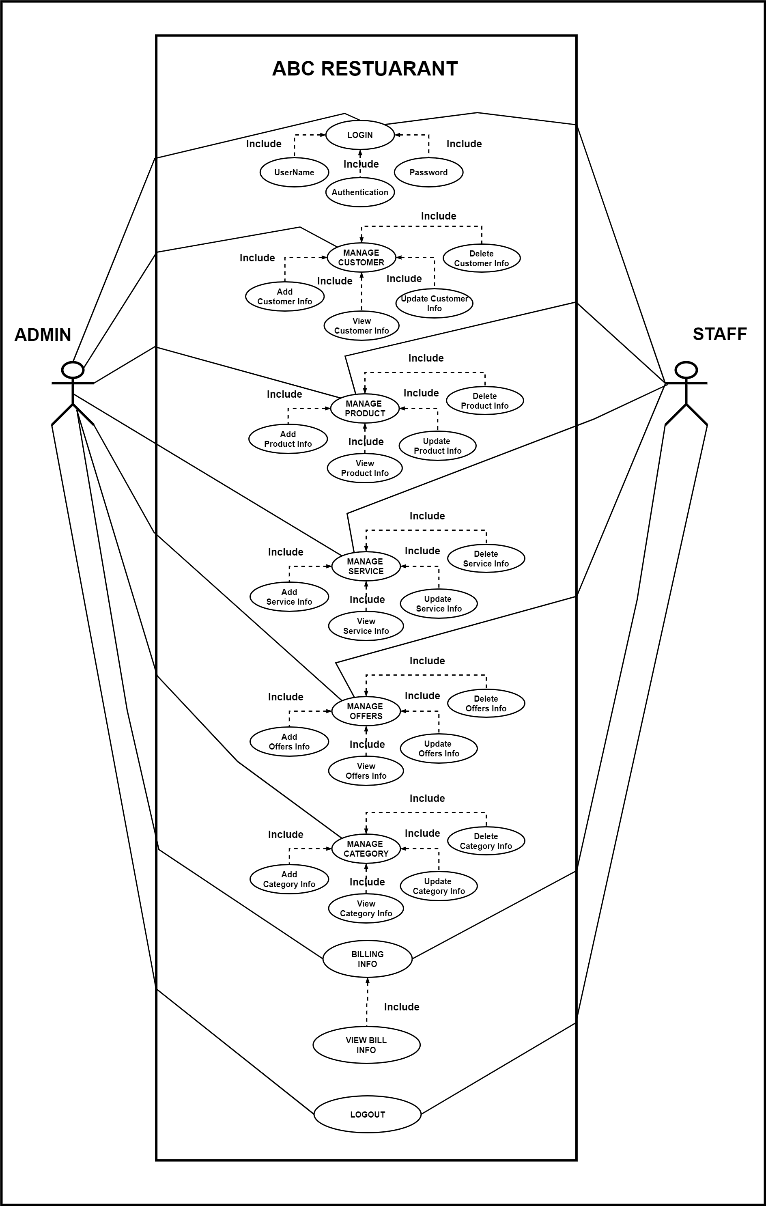
# **Task A**

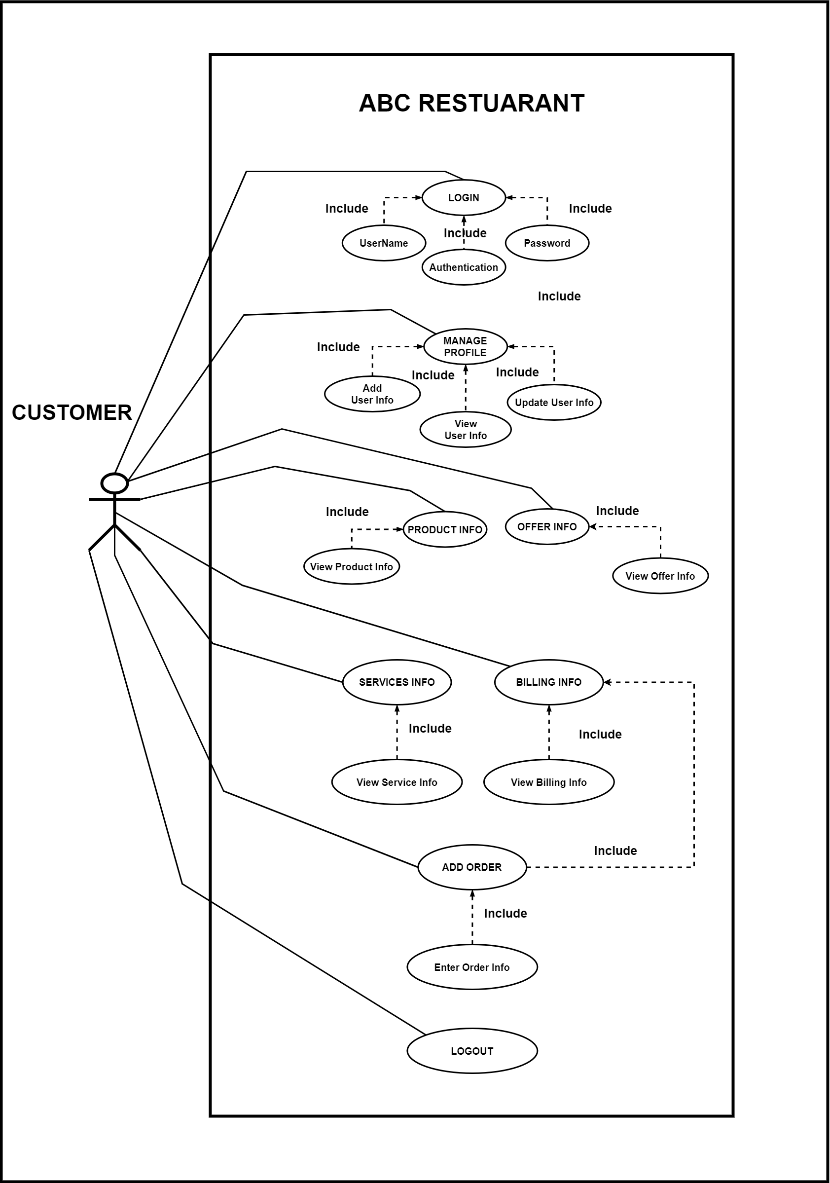
## **UML Diagrams**

### **Admin and Staff Use case Diagram**



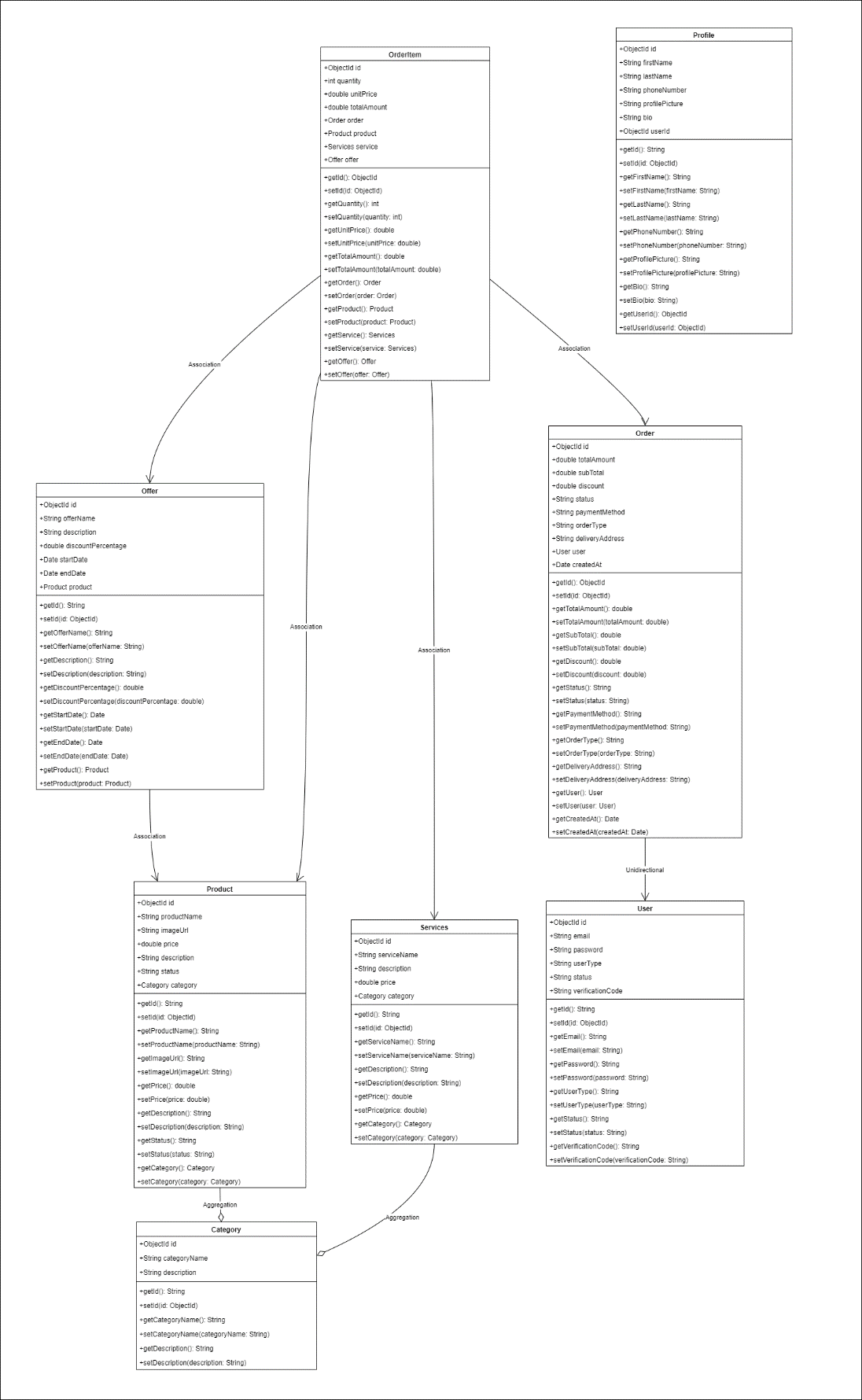
The use case graphic clearly illustrates the roles of Administration and Staff in the ABC Restaurant system. Each user has the option to verify payment details, change a number of system properties (such as customers, products, services, offers, and categories), and log in before logging out. The system is designed to enable both the staff and administrator to do their duties efficiently, within the confines of their respective access privileges.

### **Customer Use case Diagram**



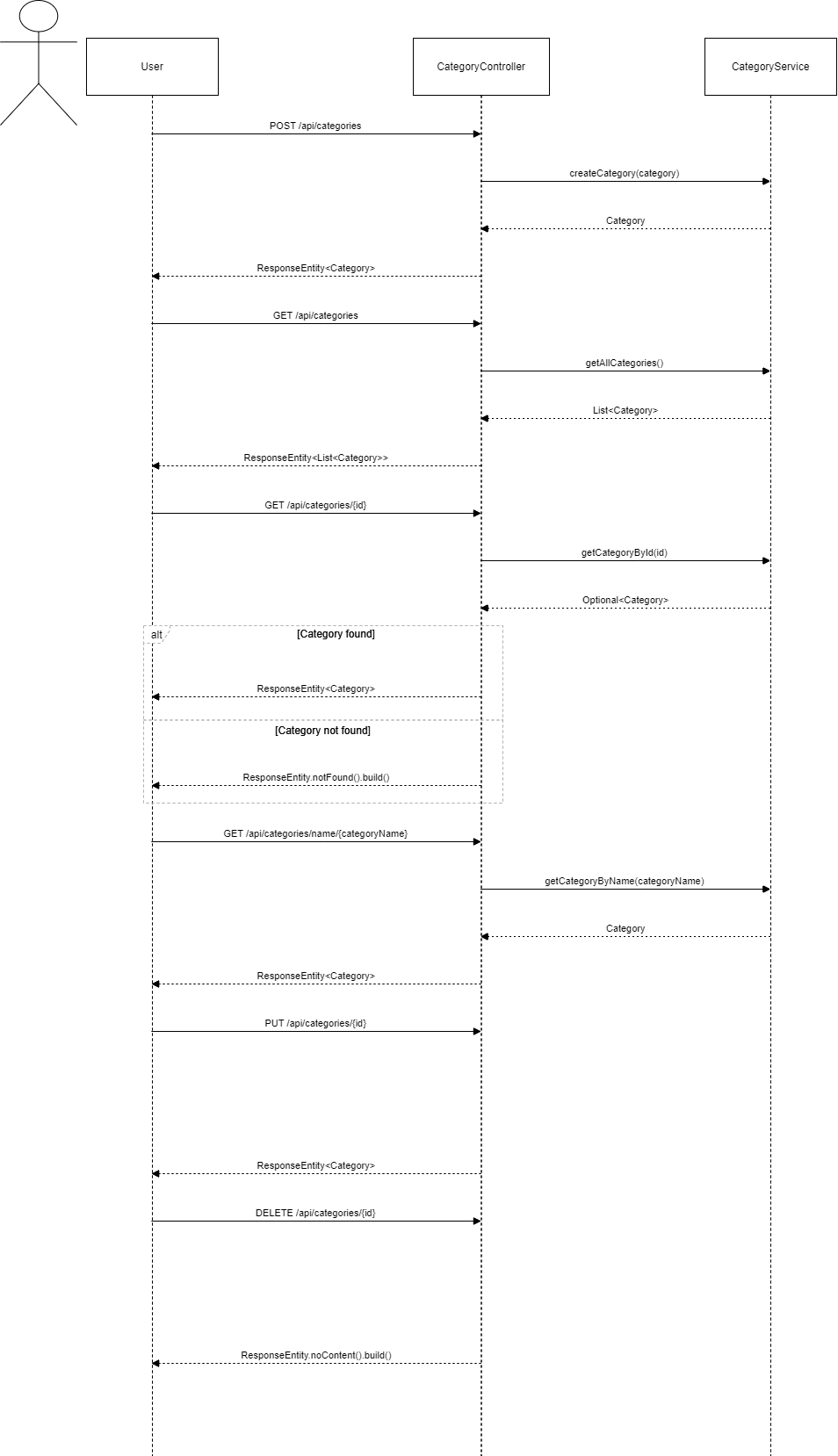
This use case diagram illustrates the many interactions a customer may have with the ABC Restaurant system. The illustration demonstrates how a customer makes purchases, verifies billing information, manages their profile, and views offers, products, and services. Ensuring the security of their experience, the user may access and depart the system with safety.

### **Class Diagram**

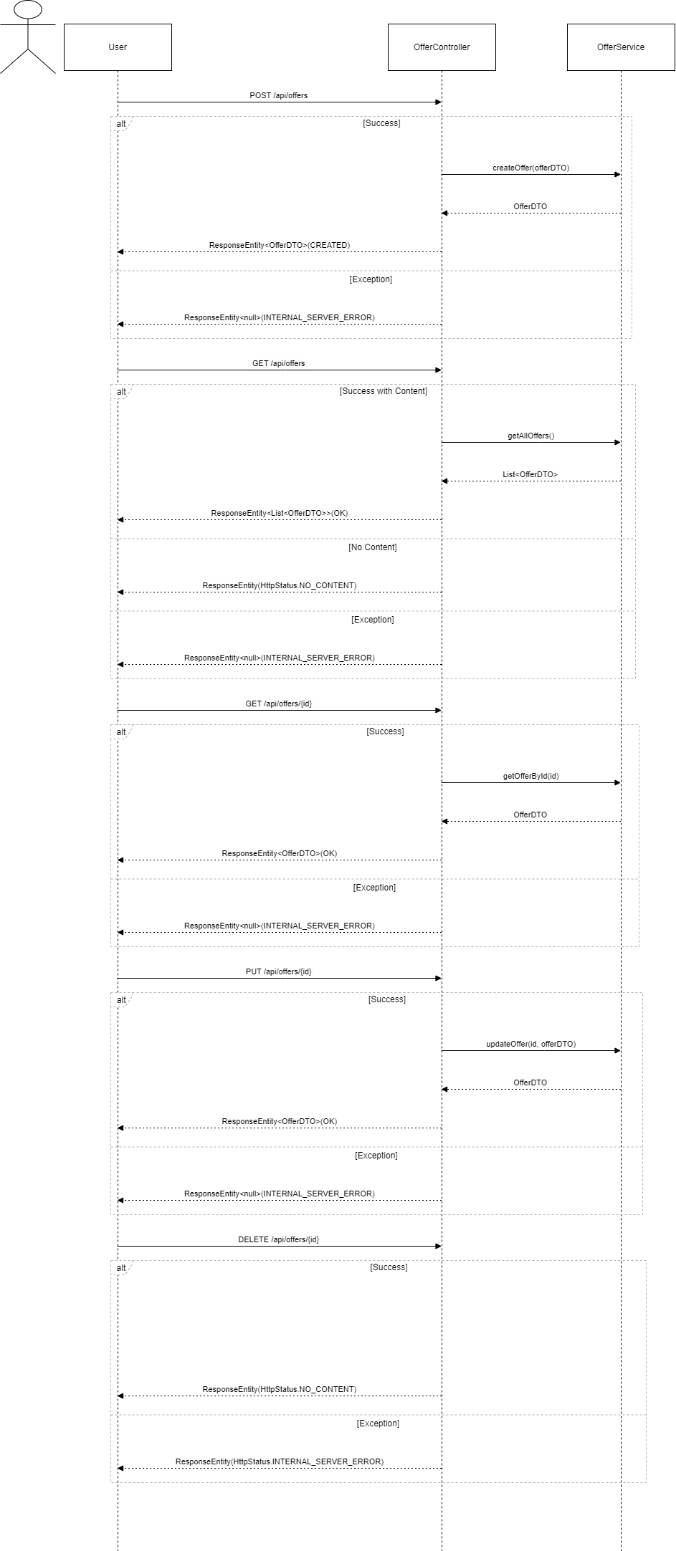


This class diagram, which illustrates the interactions between numerous entities including users, profiles, orders, items, services, deals, and categories, accurately captures the structure of the ABC Restaurant system. It provides a clear image of the attributes, functions, and relationships between each class, which facilitates understanding the system's architecture and data flow.

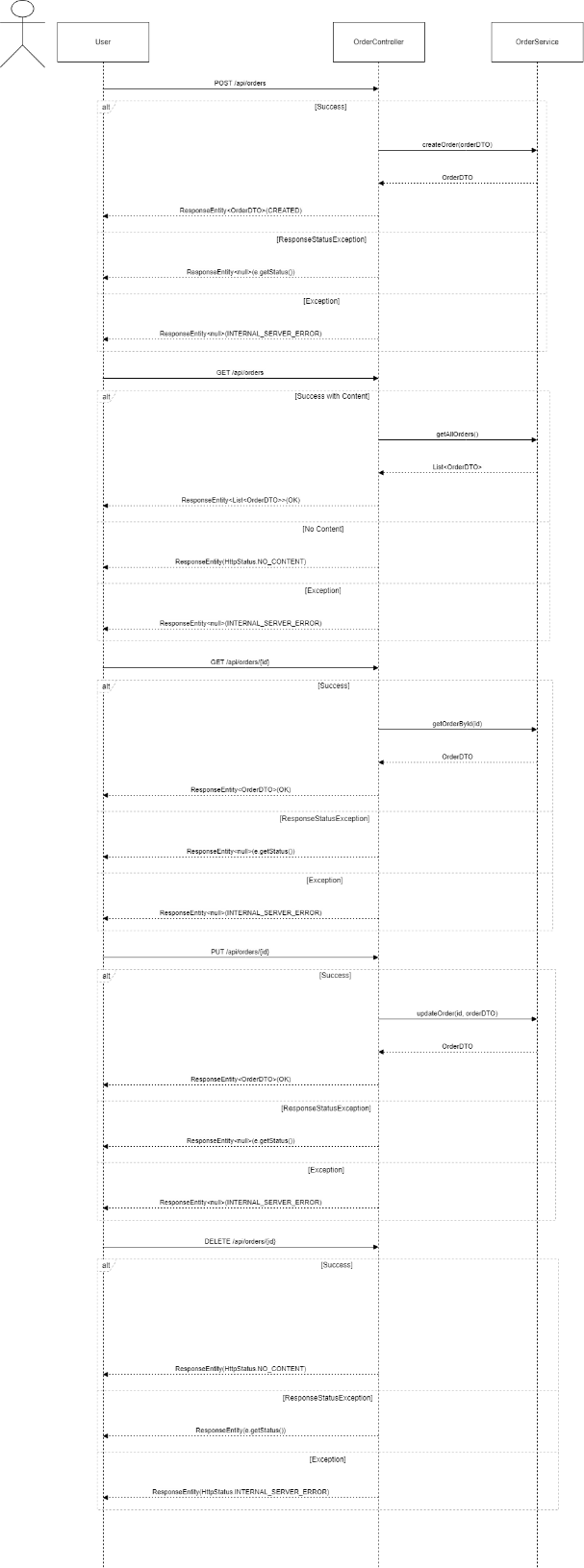
### **Category - Sequence Diagram**



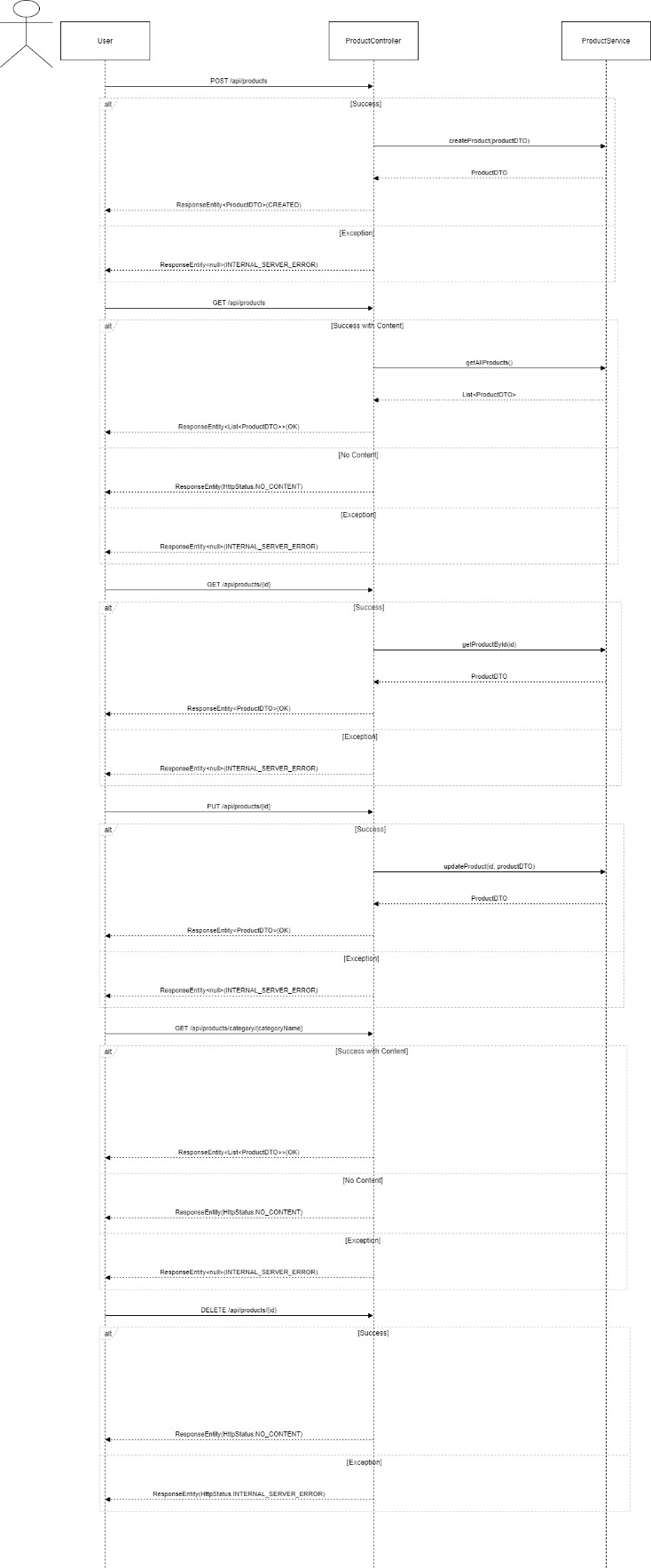
### **Offer - Sequence Diagram**



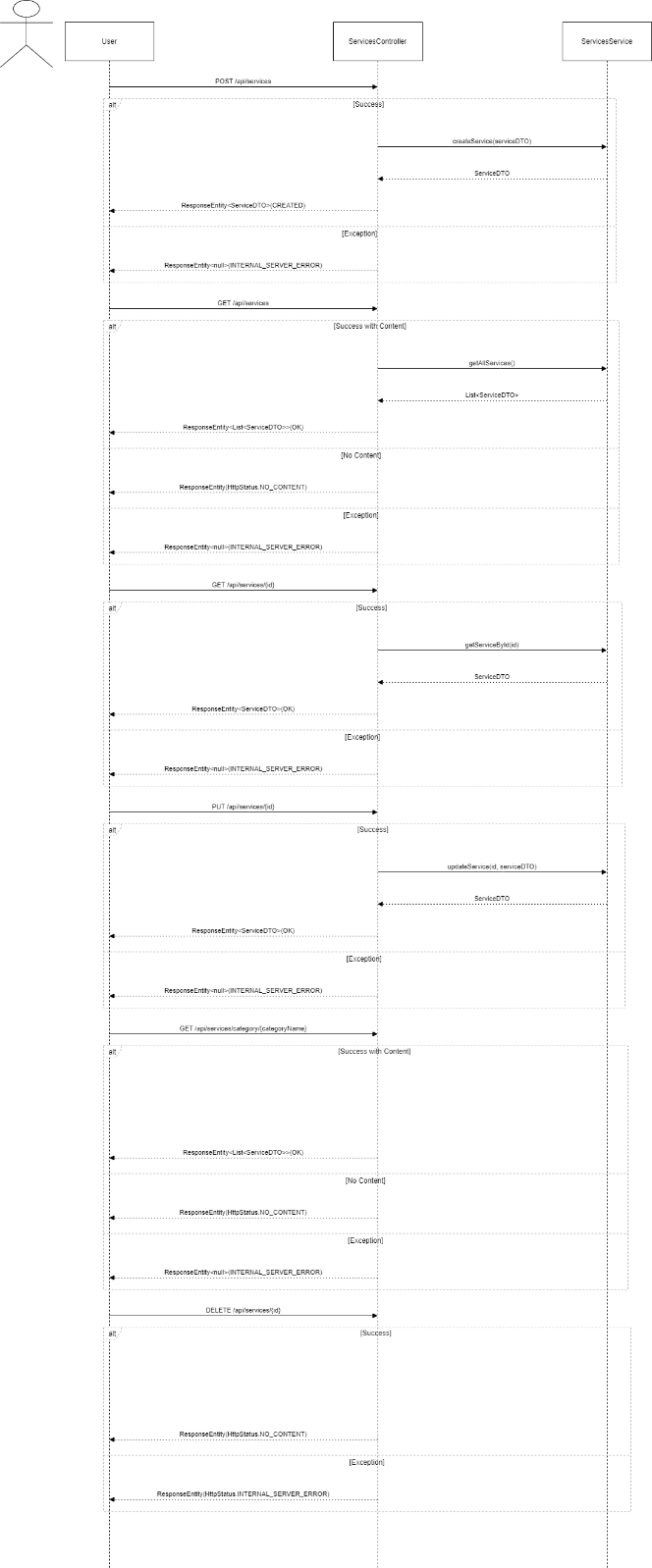
### **Order - Sequence Diagram**



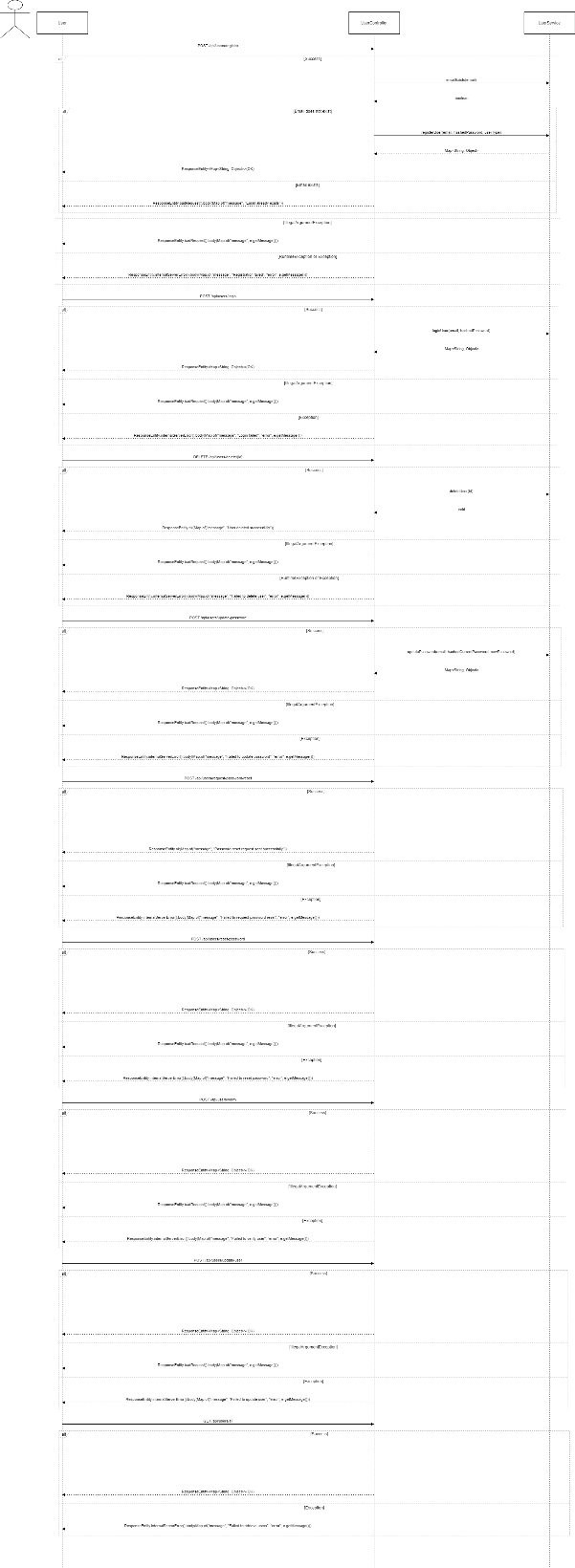
### **Product - Sequence Diagram**



### **Services - Sequence Diagram**



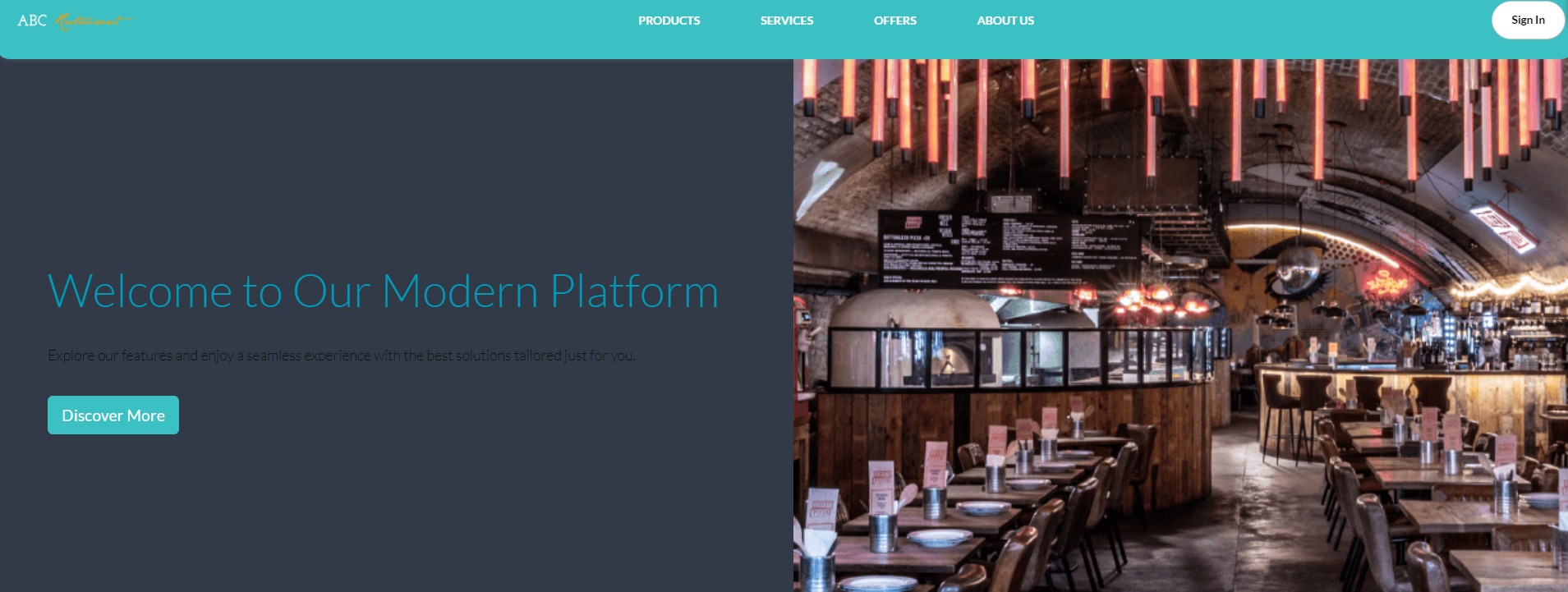
### **Users - Sequence Diagram**

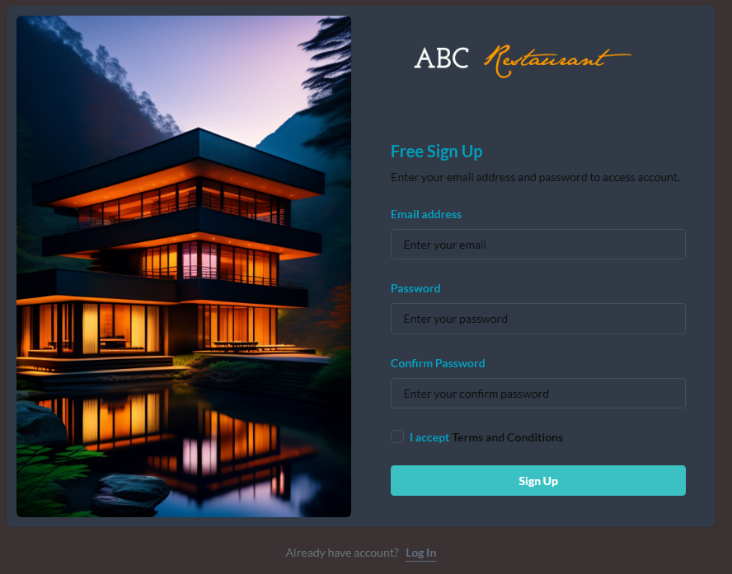


# **Task B**

**User Interfaces**

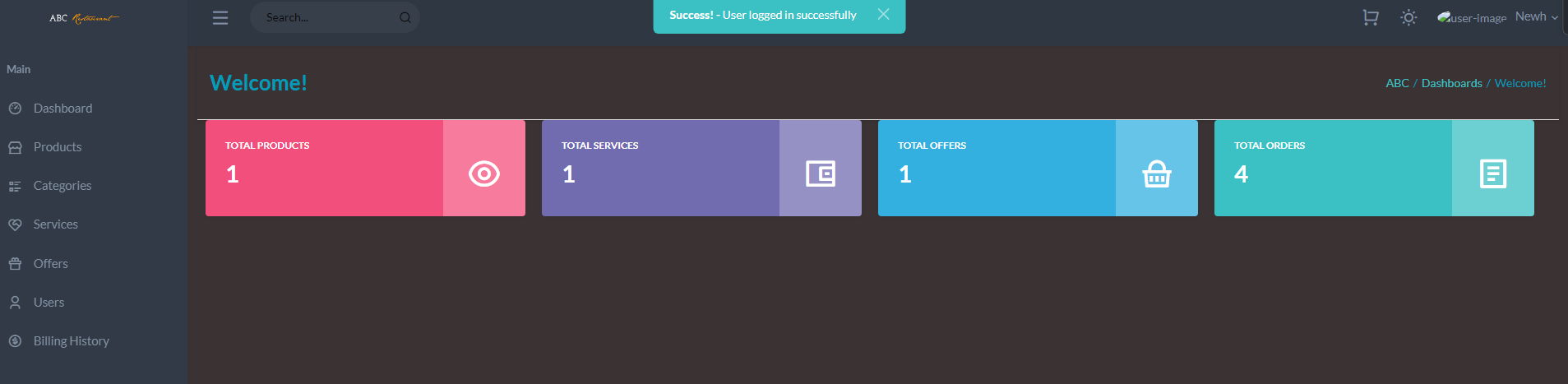
1. Home Page



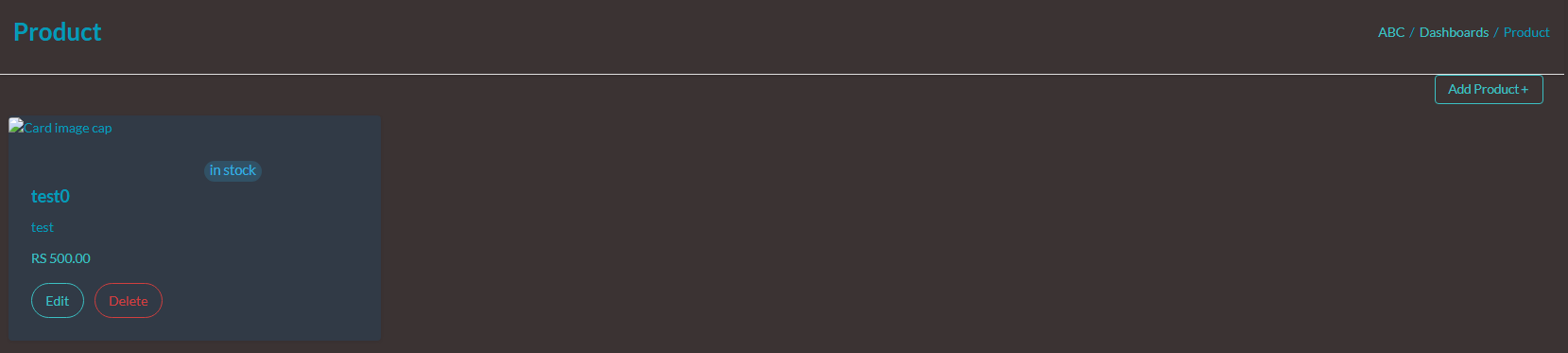
1. Sign In / Sign Up



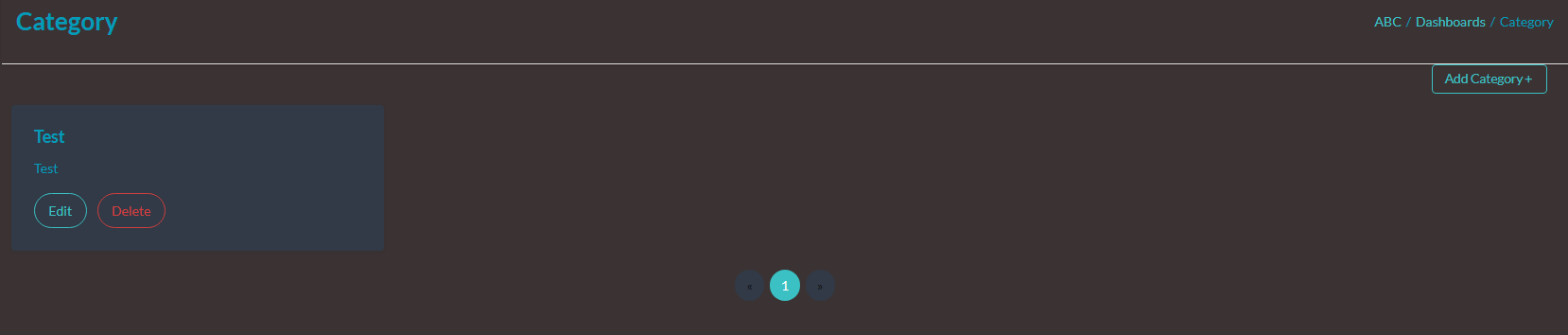
1. Admin Dashboard



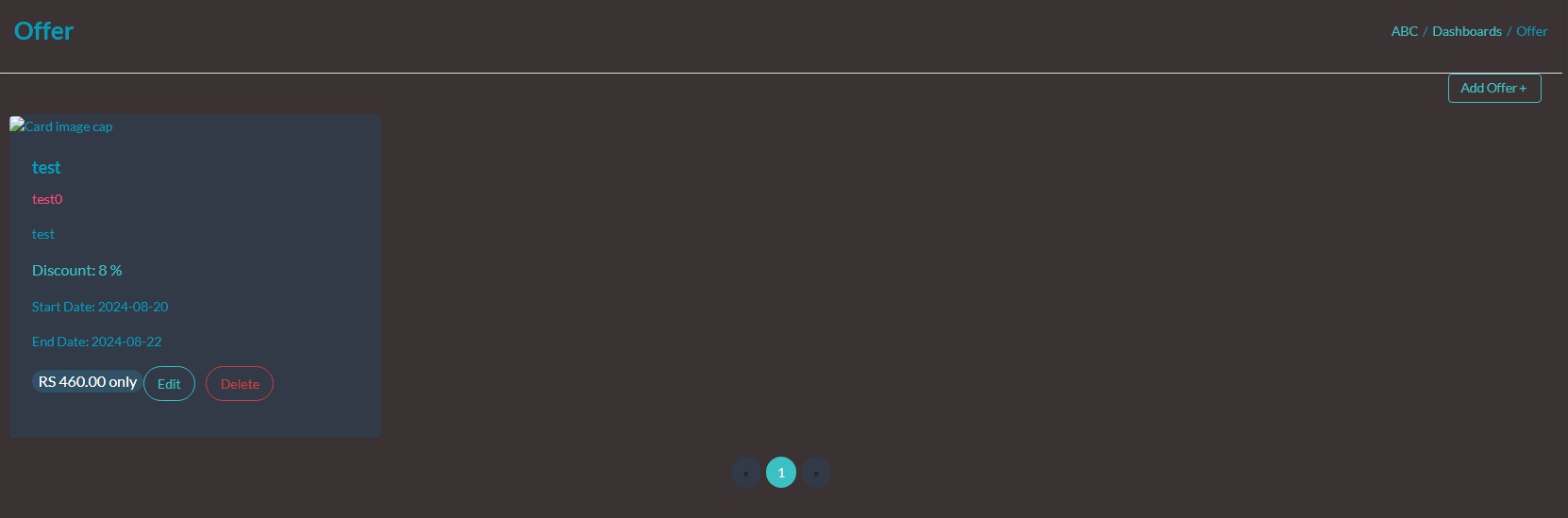
1. Admin Manage Product



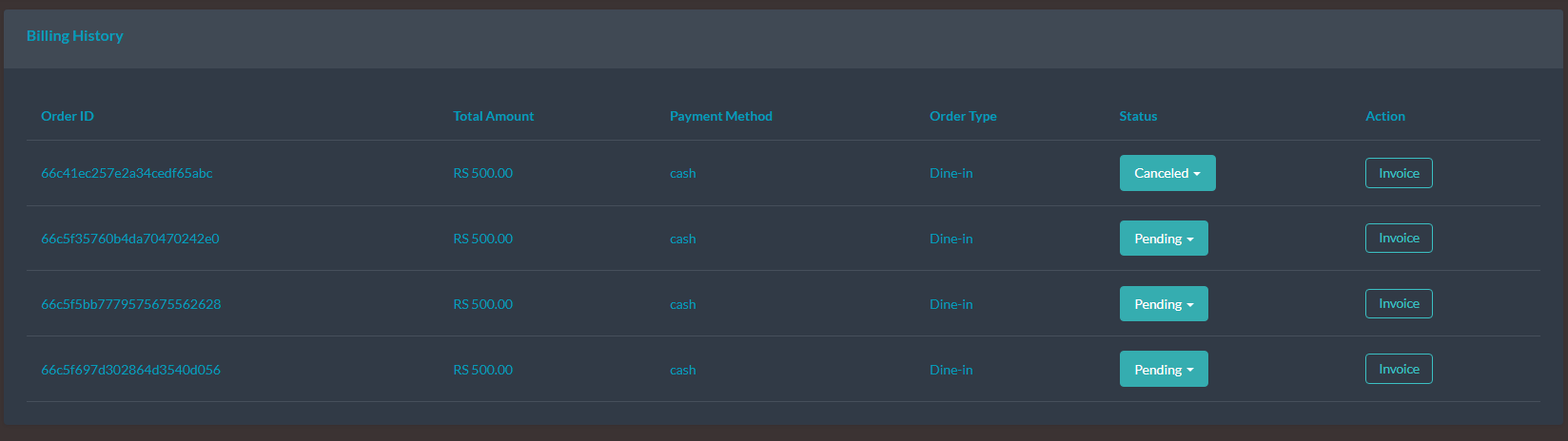
1. Admin Manage Category



1. Admin Manage Offer



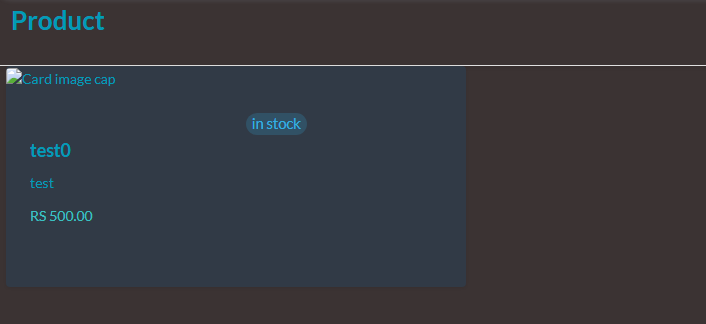
1. Admin Manage Billing History



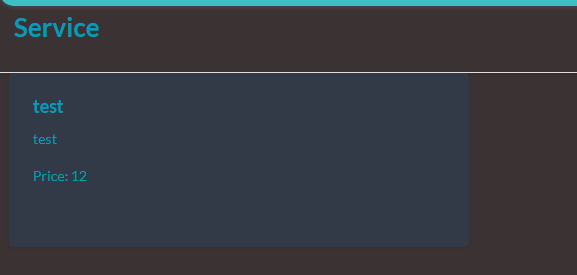
1. Staff Dashboard



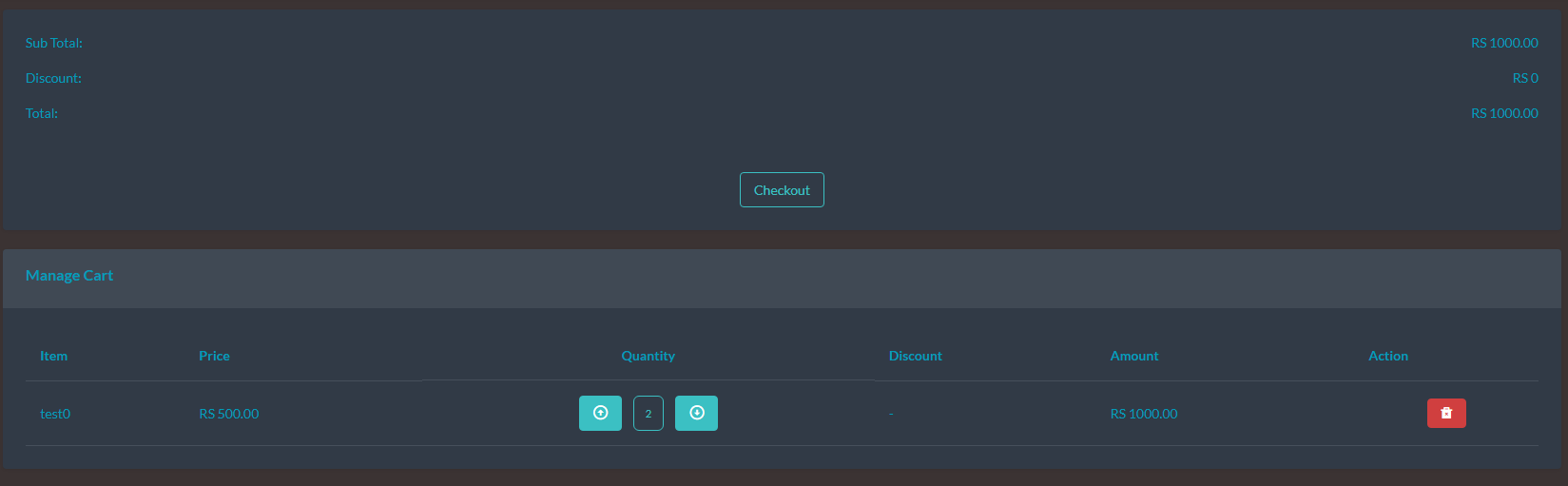
1. View Product



1. View Service



1. User Cart

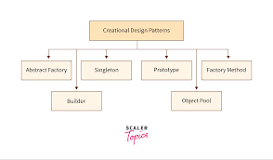


## **Design Patterns**

Well-tested answers to common issues in software development are found in design patterns. These practical solutions, which have been carefully crafted and thoroughly examined by experts in software design, provide programmers effective ways to deal with design challenges in a timely and effective manner. Code that is arranged in accordance with these design principles is simpler to reuse, adapt, and maintain. These patterns may lead developers away from common mistakes and risks, which can enhance the quality, stability, and maintainability of software. There are several design pattern substitutes, each with a unique set of benefits and applications.(Javatpoint, 2019)

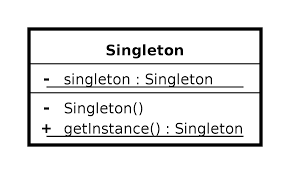
### **Creational design patterns**

Product creation processes are the main focus of the area of creational design patterns. The main goal of these patterns is to guarantee that object creation complies with the specified limitations. Moreover, they provide developers techniques that promote adaptability and reuse while building objects.

. 

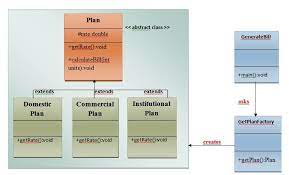
### **The singleton patterns**

When a single object has to coordinate events throughout the whole system, singleton patterns guarantee that a class only produces one instance. By creating a single point of access, this method makes it easier to coordinate system-wide actions.



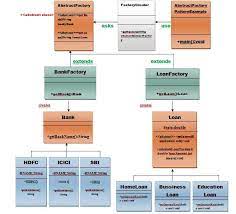
### **The factory patterns**

Subclasses may still choose which class to instantiate, and users can utilise factory patterns to create objects. This technique enables for the delegation of object instantiation chores to several subclasses, which is extremely helpful when a large number of objects need to be produced with various parameters or settings.



### **The abstract factory patterns**

The creation of connected or dependent families of objects is facilitated by abstract patterns, which eliminate the need to explicitly define each class that makes up the family. This method makes it easier to generate interconnected items without needing to know the exact type of each object, which is helpful when many related things need to be made at once.



### **The Factory Design Pattern**

One of the most popular design patterns in the Java programming community is the factory design pattern, sometimes referred to as the factory method. Despite the fact that this pattern "defines an interface to construct an object," as is often said, it is important to realise that subclasses are free to choose the class they want to instantiate. Furthermore, a class deferral mechanism may be used to extend the Factory process. A specific factory class is designated by this design pattern to generate an object of the "Object () [native code]" type on the client's behalf. We rely on this factory to create the objects in this scenario while concealing the difficulties involved in putting them into practice. Using a well defined interface is essential to getting access to produced objects.

(Tutorialspoint, 2022)

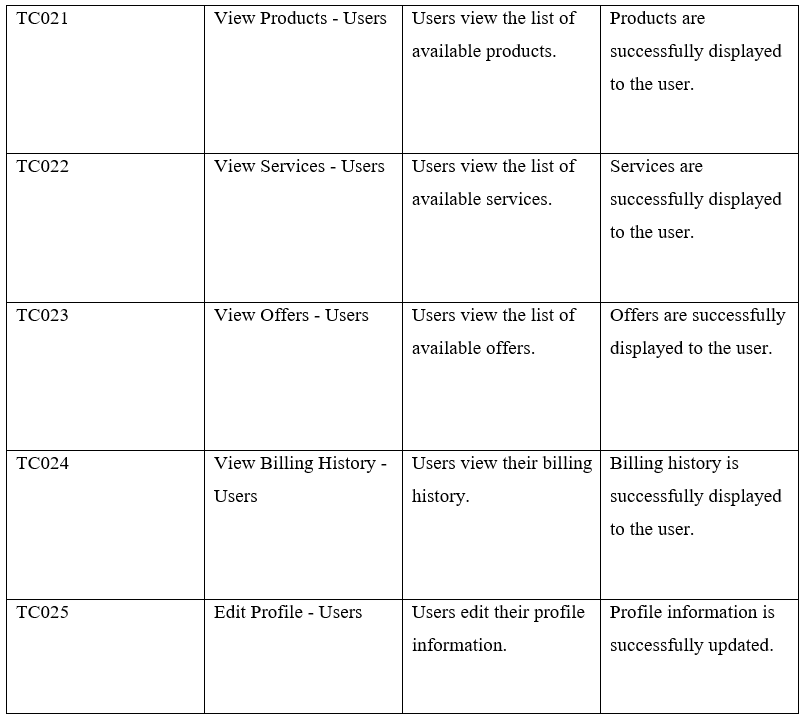
# **Task C**

## **Test Plan and test-driven development**

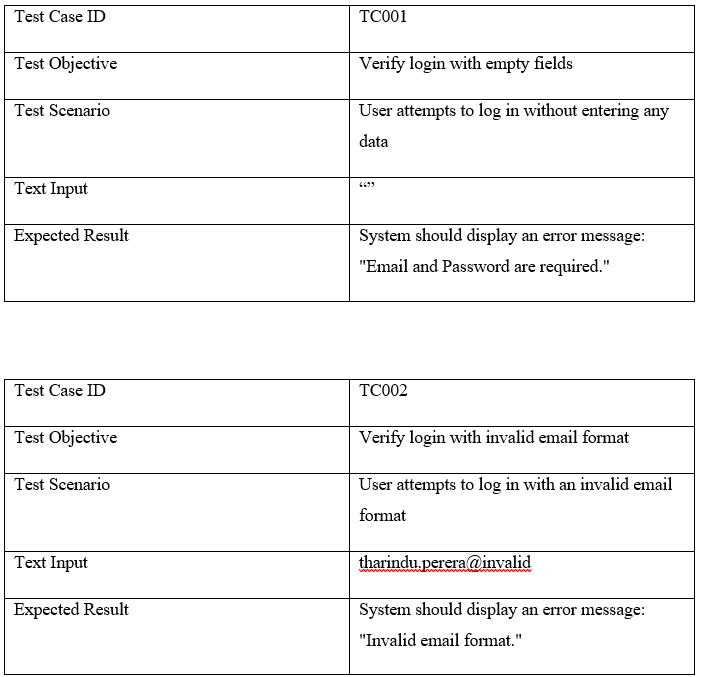


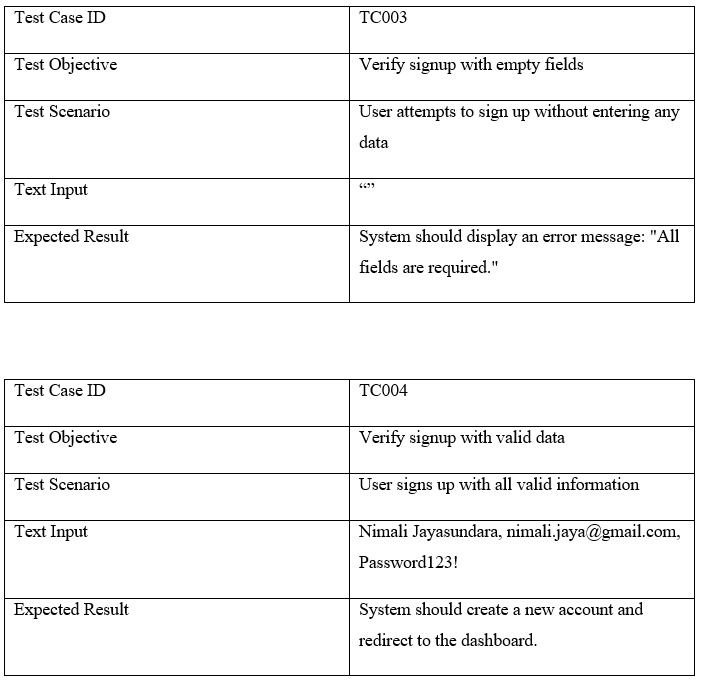


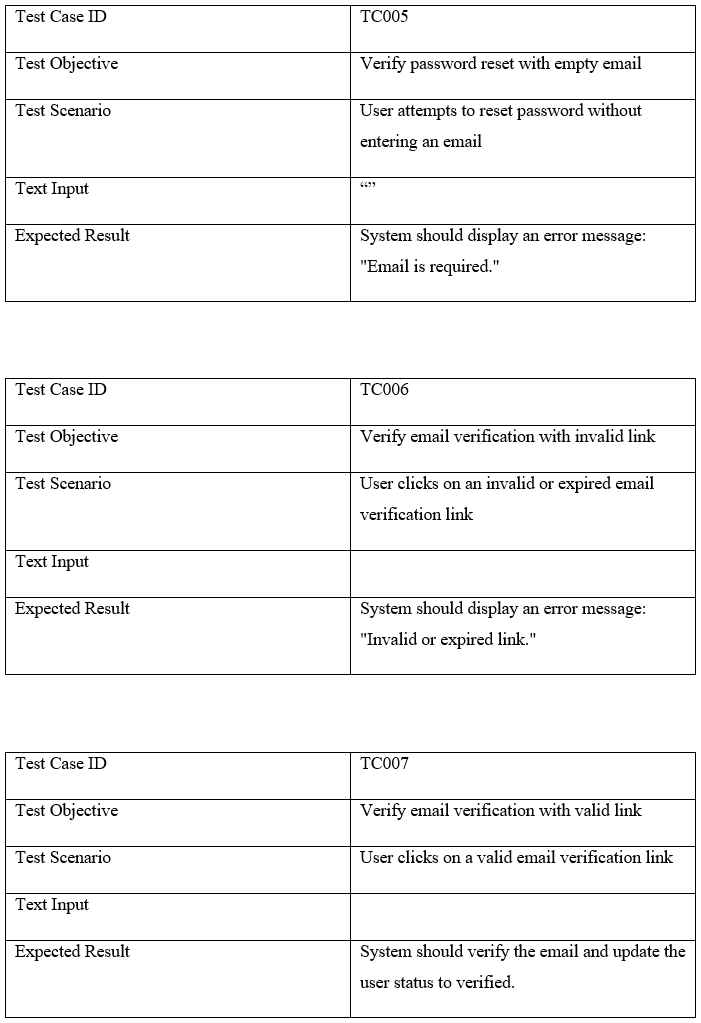


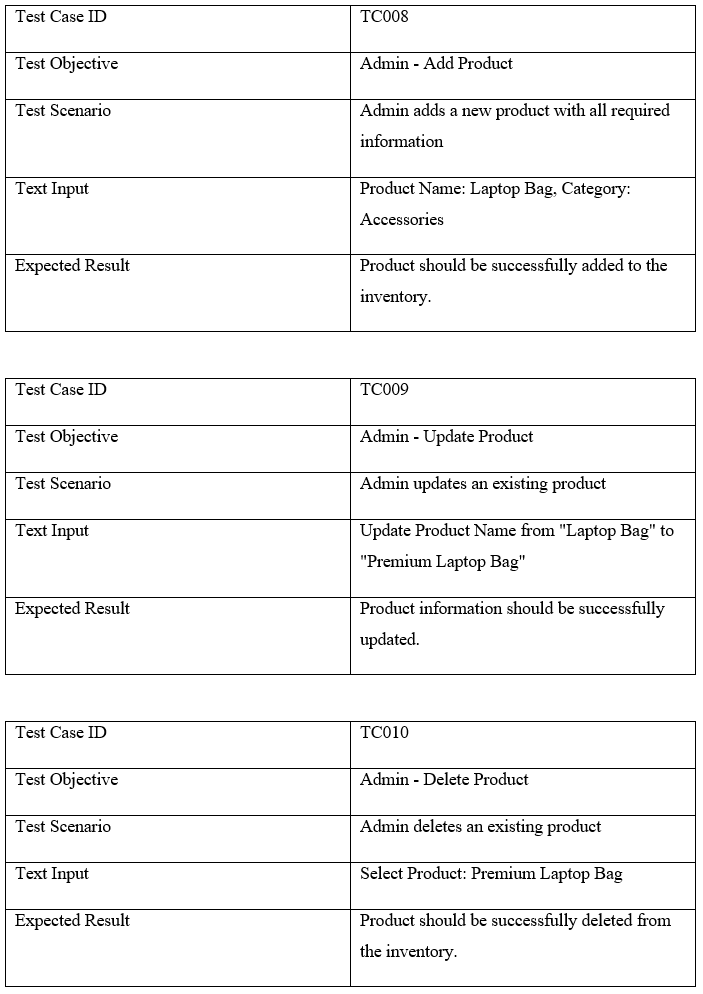


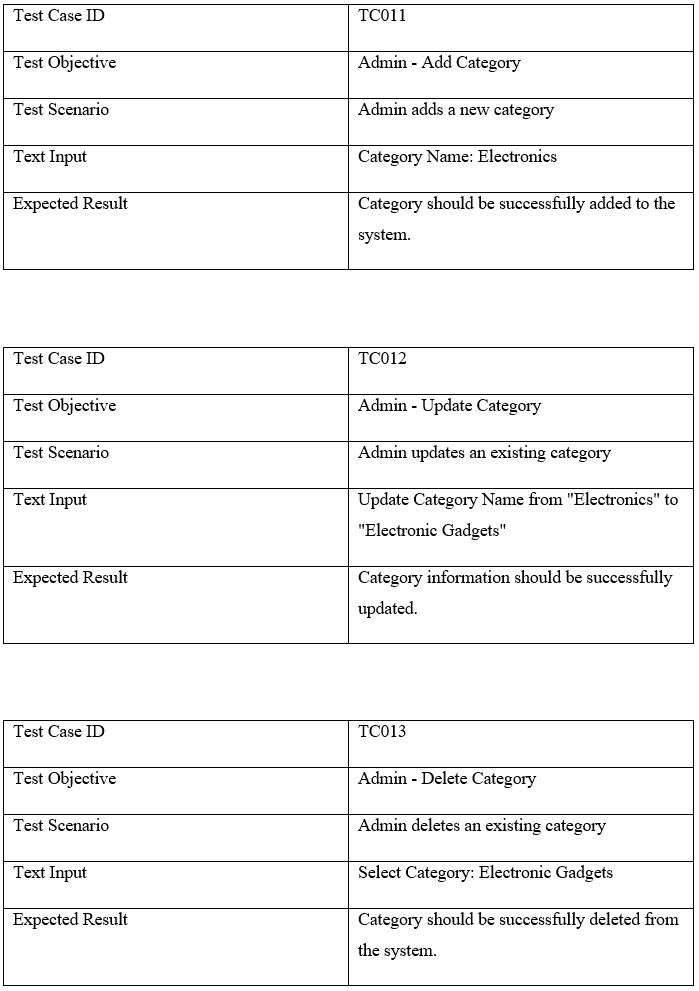
## **Test Cases**

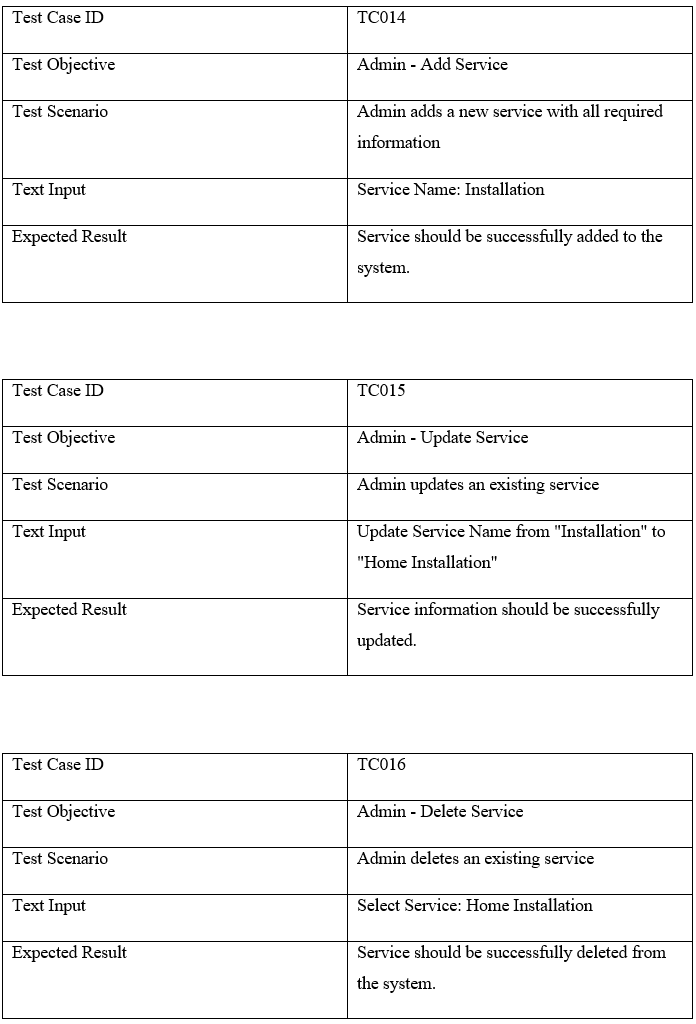


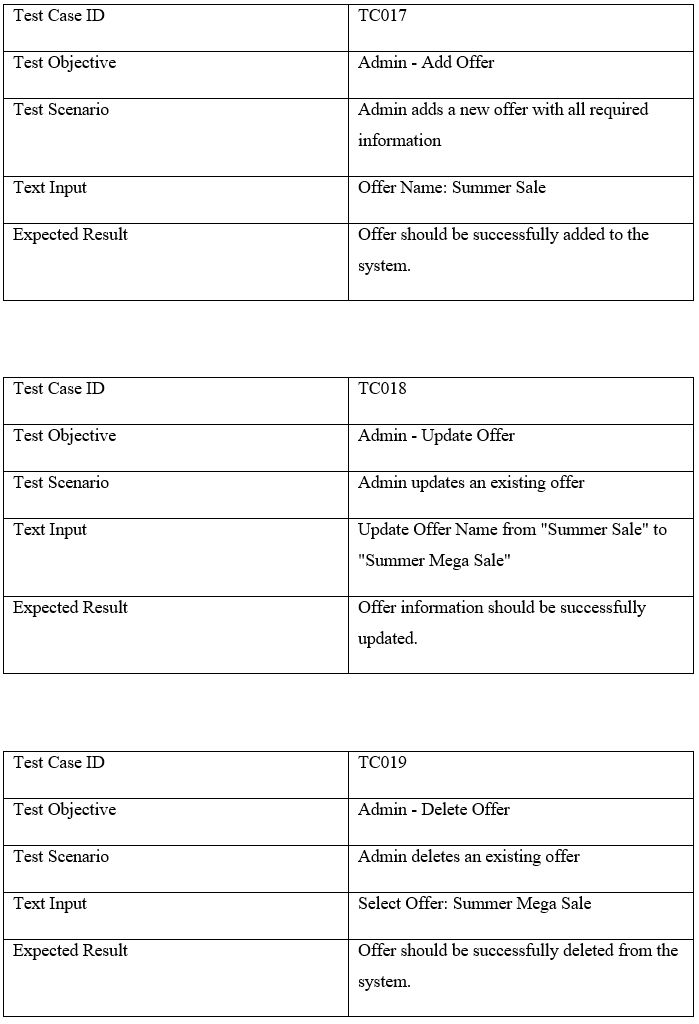


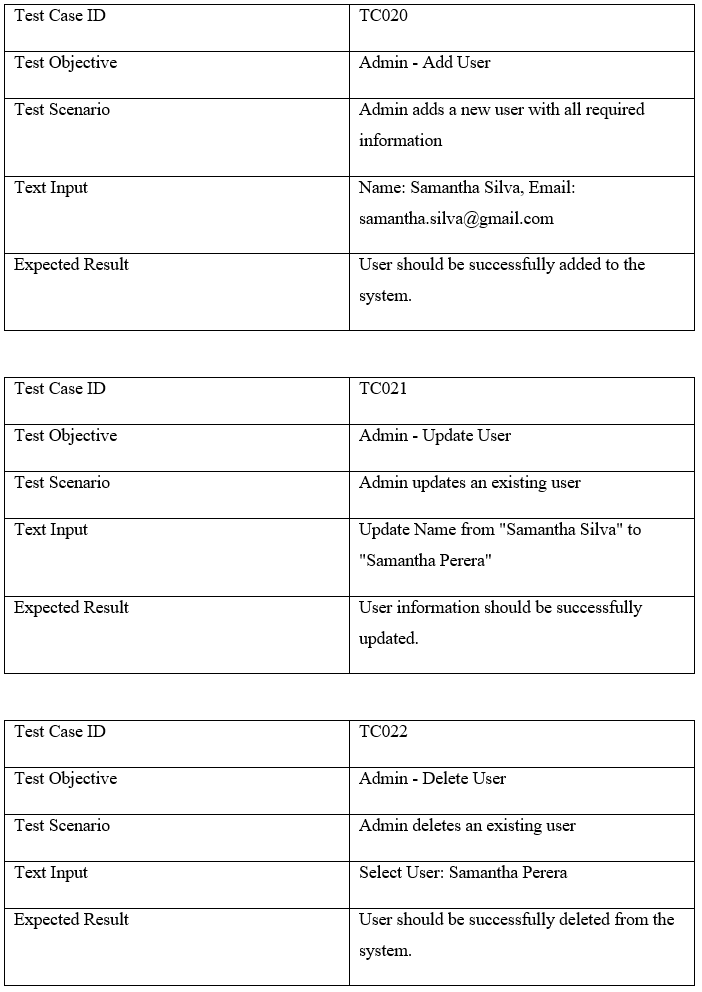


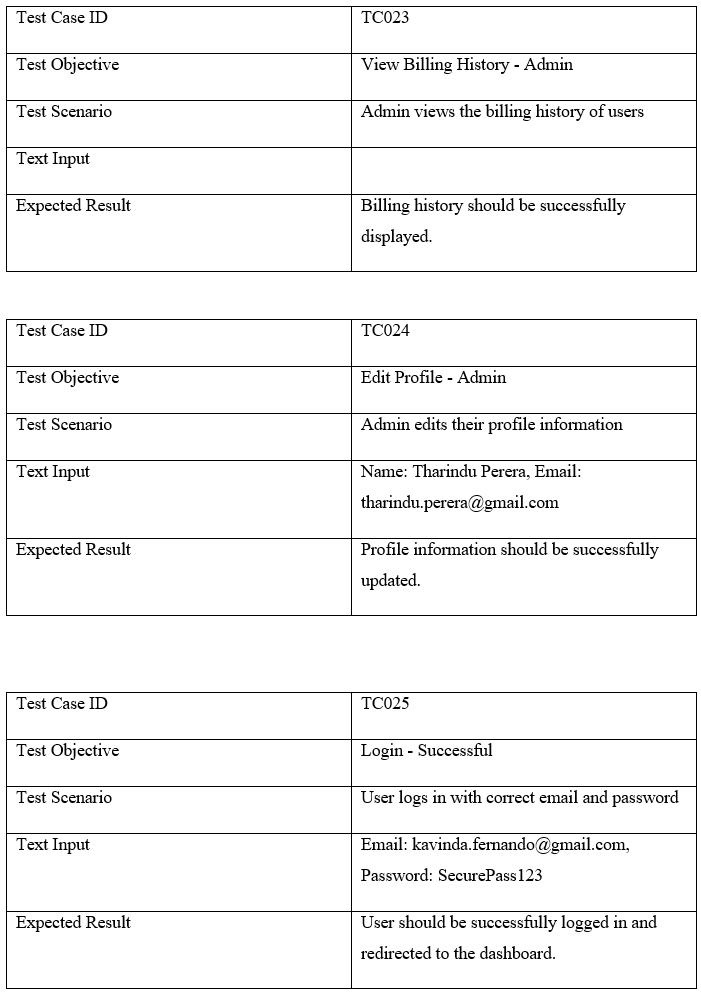












# **Task D**

## **GitHub Repository**

* GitHub Link -

Repo Image

**Code Explanation**

**Category Controller**

****

The primary logic for managing categories is included in the CategoryService, which the controller utilises to carry out various category-related operations. When a new category has to be created, the createCategory method—annotated with @PostMapping—is used. This function receives the Category object from the request, creates the category, processes it, and provides a 200 (OK) response. There are two ways you may get categories: A single category is obtained by its ID using the getCategoryById method, which also makes use of @GetMapping. If the category is discovered, a 200 response is returned; if not, a 404 (Not discovered) reply is returned. A list of all categories is returned by the getAllCategories function, which is marked with @GetMapping. To obtain categories by name, you can also use the getCategoryByName method, which is mapped using @GetMapping and produces a 200 response. An existing category may be modified using the updateCategory function, which is indicated by @PutMapping. It returns the modified category with a 200 status after receiving the ID and the changed data as input. Last but not least, the @DeleteMapping-related deleteCategory function allows a category to be removed by its ID. The deletion was successful, and no content was returned, according to the 204 (No material) result that is returned.

**Image Controller**

****

The uploadImage method, which is designated with @PostMapping("/upload"), is used to handle image uploads. This method accepts a MultipartFile from the client and makes use of the imageService.UploadImage(file) should be used to finish the upload. Upon successful completion, the filename and a 200 (OK) status are sent back. In the event of an error, the method logs the 500 (Internal Server Error) status, logs the exception, and returns an error message. On the other hand, the retrieveImage method may be used to retrieve an image by its filename; this can be expressed as @GetMapping("/retrieve/{fileName}"). If the image is successfully located, a 200 (OK) status and the appropriate JPEG format image data are provided. If there is a problem or the image cannot be found, the method returns 404 (Not located).

**Offer Controller**



The createOffer method, mapped to @PostMapping, is responsible for creating a new offer using the provided OfferDTO. If creation is successful, it returns the produced offer along with a 201 (generated) status. In the event that problems occur throughout the process, the method returns a 500 status (Internal Server Error). The getAllOffers method acquires all of the available offers by using @GetMapping. If the retrieval is successful, the offers are returned with a 200 (OK) status; otherwise, a 204 (No Content) result is returned. The status is 500 if there are any problems throughout this procedure. To get an offer by its ID, use the getOfferById method, which is also mapped to @GetMapping. If the retrieval is successful, the method returns the offer with a 200 (OK) status; if there are any problems, it returns a 500 status. To update an existing offer, use the updateOffer method (annotated with @PutMapping) with the provided ID and OfferDTO. The updated offer is provided with a 200 (OK) status when an update is successful, and a 500 status is returned in the event of a problem. Finally, the @DeleteMapping-connected deleteOffer method is used to remove an offer based on its ID. If the deletion is successful, a 204 (No Content) status is issued; if there are any problems throughout the process, a 500 status is given.

**Order Controller**

****

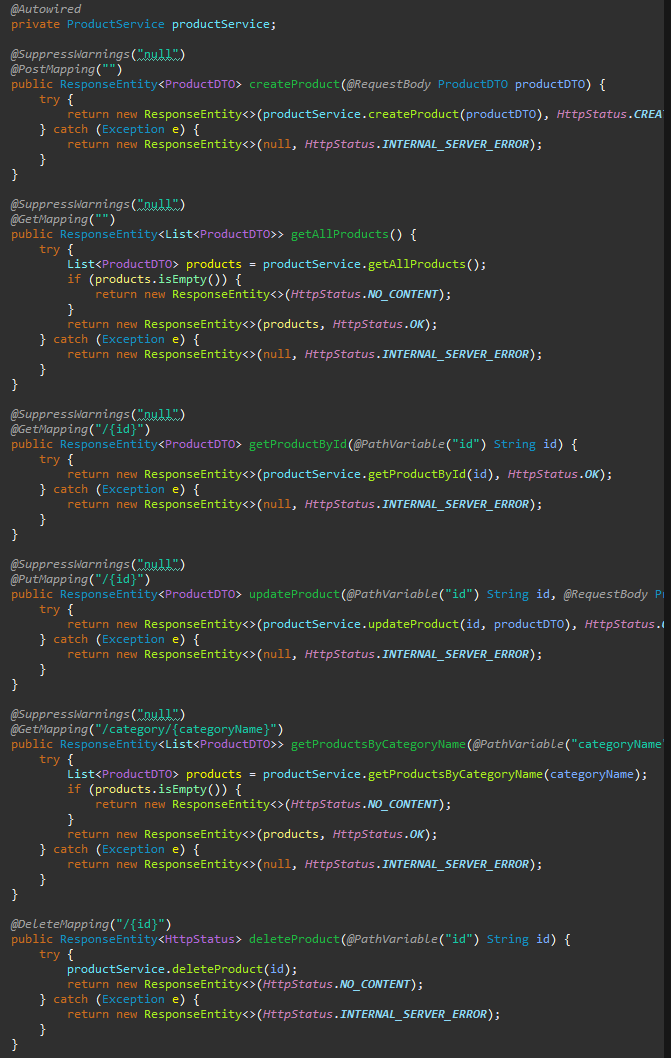
The createOrder function, marked with @PostMapping, is meant to create a new order using the provided OrderDTO. If creation is successful, it returns the created order with a 201 (Created) status. When a specific problem occurs, such a ResponseStatusException, the method returns the related error status. If there are any more unanticipated errors, a 500 (Internal Server Error) status is shown. The getAllOrders method, which is mapped to @GetMapping, is used to obtain all orders. If no orders are discovered, a 204 (No Content) status is returned; if orders are located, a 200 (OK) status is issued. The status is 500 if there are any problems throughout this procedure. Using the getOrderById method, which is also linked to @GetMapping, an order may be obtained by ID. If the method is successful, it returns the order with a 200 (OK) status. A 500 status is caused by other issues; if a ResponseStatusException is raised, the specific error status is returned. The @PutMapping updateOrder method updates an existing order using the provided ID and OrderDTO. If the modification is successful, the modified order is returned with a 200 (OK) response. Unexpected errors produce a 500 status; a ResponseStatusException raises an exception and returns the given error status. Finally, the deleteOrder method, mapped to @DeleteMapping, may be used to remove an order by its ID. If the deletion is successful, a 204 (No Content) answer is sent back. In the case of a ResponseStatusException, the function returns the specific error status; other problems get a 500 status.

**Order Item Controller**

****

Using the supplied OrderItemDTO, the createOrderItem method, mapped to @PostMapping, is in charge of producing a new order item. The method returns the generated item with a 201 (generated) status once creation is successful. A 500 (Internal Server Error) status is issued in the case of an error. All order items are retrieved using the @GetMapping-related getAllOrderItems function. A 204 (No Content) status is returned if no items are detected, while a 200 (OK) status is given if things are found. If there are any issues throughout this operation, the status is 500. To obtain an order item by its ID, use the getOrderItemById function, which is also mapped to @GetMapping. The method returns the item with a 200 (OK) status if the retrieval is successful; failures are handled by returning a 500 status. An order item may be deleted by its ID using the @DeleteMapping-linked deleteOrderItem function. Any failures result in a 500 status, but successful deletions return 204 (No Content). Furthermore, all order items connected to a certain order ID are deleted using the deleteOrderItemsByOrderId function, which is mapped to @DeleteMapping as well. The method returns a 204 (No Content) status if it is successful; faults result in a 500 status. Last but not least, the @GetMapping function getOrderItemsByOrderId obtains every order item connected to a given order ID. A 204 (No Content) status is returned if no items are detected, while a 200 (OK) status is given if things are found. If there are any issues throughout this operation, the status is 500.

**Product Controller**

****

The createProduct function, mapped to @PostMapping, is responsible for creating a new product using the provided ProductDTO. If creation is successful, it returns the newly created product with a 201 (formed) status. In the event of an error, the function returns a 500 (Internal Server Error) status. Using the getAllProducts method connected to @GetMapping, all accessible commodities are fetched. In the event that no goods are identified, it returns a 200 (OK) value; otherwise, it returns a 204 (No Content) status. A 500 status is issued in the event that there are any issues with this retrieval process. Use the getProductById method, which is mapped to @GetMapping as well, to get a product by its ID. When retrievals are successful, the product is returned with a 200 (OK) status; otherwise, errors result in a 500 status. The updateProduct method connected to @PutMapping makes changes to an existing product using the provided ID and ProductDTO. If the update is successful, the method returns the updated product with a 200 (OK) status; if there are any problems, it returns a 500 status. The getProductsByCategoryName method, mapped to @GetMapping, retrieves products by name in their category. If no items are found, a 204 (No Content) status is returned; if things are found, a 200 (OK) status is issued. The status is 500 if there are any issues throughout this process. Finally, the @DeleteMapping-related deleteProduct method may be used to remove a product by its ID. 500 is returned if there are any problems; 204 (No Content) is returned if deletion is completed.

**User Controller**

****

The registerUser function may be used to register a new user using their email address and password. First, the email address is compared to a database to determine whether it already exists. whether it doesn't, the password is hashed and the user is added as a "customer." Following a successful registration, it provides the user's information along with a 200 (OK) status. Any issues that come up throughout this process are denoted by the appropriate status codes. The loginUser function controls user login by confirming the password and email provided. If the credentials are genuine, the procedure delivers user data with a 200 (OK) status; if not, an internal server error 500 status is returned.

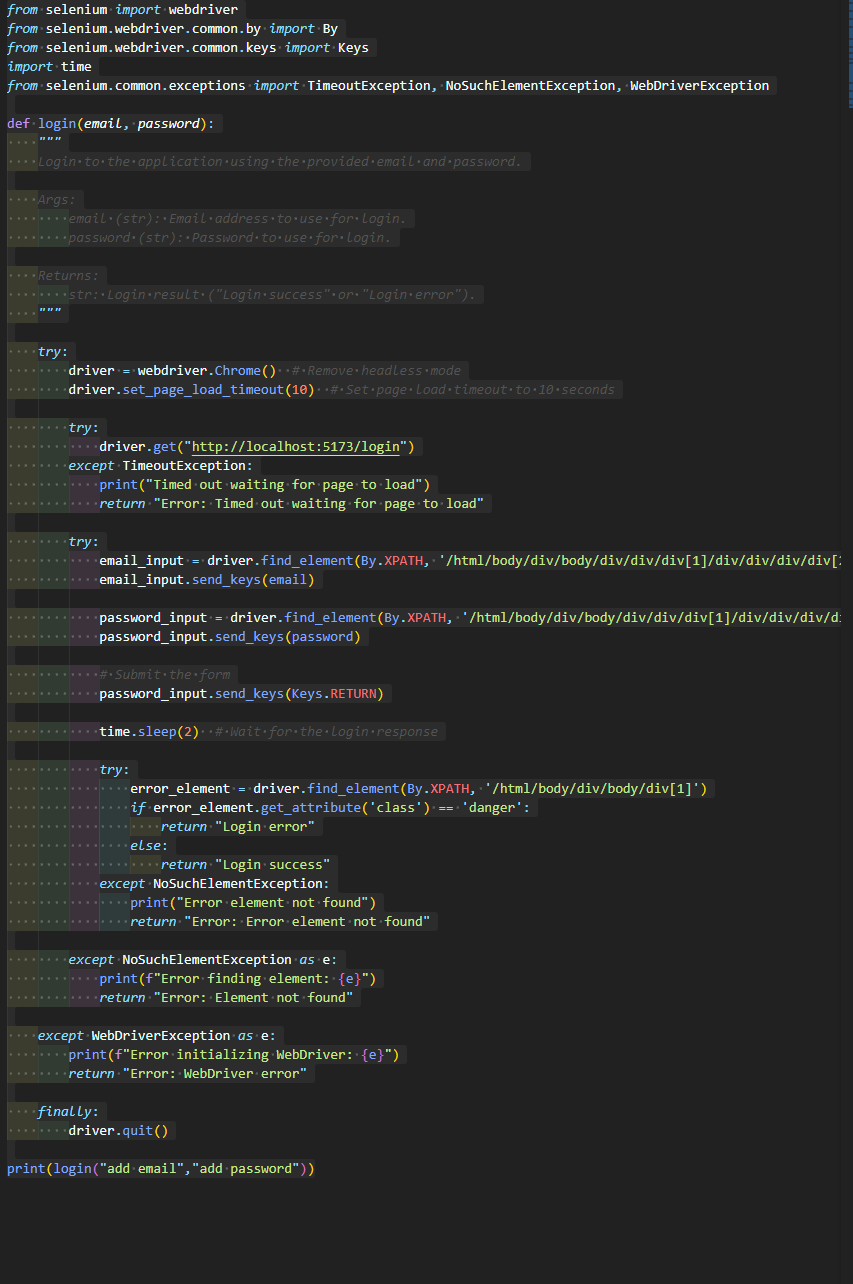
Use the deleteUser function to remove a user by their ID. Any issues yield a 500 status, whereas removals that are successful get a 200 (OK) status with a success message. Users may utilise the updatePassword method to update their passwords by providing their email address, old password, and new password. If the update is successful, the method returns a 200 (OK) status; otherwise, it returns a 500 status.

The process of changing the password is initiated using the requestPasswordReset method. A success message and a 200 (OK) status are returned if it is successful; a 500 status is given if there are any problems. By utilising the resetPassword feature, users may reset their password along with a verification number. If the reset is successful, it returns a 200 (OK) status; if an error occurs, it returns a 500 status.

The verifyUser method is in charge of using a verification code to confirm a user. When anything goes wrong, the status is 500; when it works, the status is 200 (OK). Use the updateUser method to update user data, such as type and status. An update that is successful returns a 200 (OK) status; failures result in a 500 status.

The hashPassword function is used to hash passwords using the SHA-256 algorithm. If the hashing process fails, a RuntimeException is produced. Lastly, the getAllUsers method is used to obtain every user. After a successful retrieval, users get a 200 (OK) status; any errors result in a 500 status.

**Selenium Testing**

****

The script automates the login process for a web application using Selenium WebDriver. Chrome is the recommended browser for initialising the Selenium WebDriver. The necessary modules are then installed in order to interface with web components, handle problems, and perform actions such as entering and submitting forms.

The login function of the script takes two inputs: an email address and a password. The initial stages in the process include opening the user-visible Chrome browser and visiting the login page URL, which is http://localhost:5173/login. The script then waits for the page to fully load, sometimes for up to 10 seconds. It returns a timeout error message if it doesn't.

After the page loads, the script uses its XPath selectors to find and interact with the email and password fields on the login form. It sends the RETURN key to submit the form after the specified credentials have been entered. The script checks the page for an error element after a submit to make sure the login was successful. An element with the "danger" class indicates a failed login attempt; in this scenario, the script displays a "Login error" message. If no such element is detected, the script returns the message "Login success" and considers the login successful.

Throughout the script, try-except blocks are utilised to handle common errors like as missing objects, issues with page loading, and WebDriver failures. Finally, to prevent resource leaks, the script ensures that the WebDriver object is closed by calling driver at the end of the run.stop(), regardless of the success or failure of the login attempt.