Software
**Engineering**

University of Belize
CMPS4131 – Group 3

# Report 2
# System Design


Azriel Baris Cuellar
Levi Coc
Daniel Eugenio Garcia
Victor Alexander Jr. Tillet
Floyd Jason Ack

**Website Link**

https://2019120154.wixsite.com/buscommutecompanion

# Table of Contents


1. INDIVIDUAL CONTRIBUTIONS BREAKDOWN 2
2. ANALYSIS & DOMAIN MODELING 4
    2.1 Concept Definitions 4
    2.2 Association Definitions 7
    2.3 Attribute Definitions 12
    2.4 Traceability Matrix 14
    2.5 Domain Models 15
3. SYSTEM OPERATION CONTRACTS 19
4. DATA MODEL & PERSISTENT STORAGE 21
5. INTERACTION DIAGRAMS 22
6. CLASS DIAGRAM & INTERFACE SPECIFICATION 25
    6.1 Class Diagram 25
    6.2 Data Types and Operation Signature 25
    6.3 Traceability Matrix 26
7. DATA STRUCTURES 27
8. DESIGN OF TESTS 29
    UC #10 – View Booking History 29
    UC #9 – View Payment History 30
    UC #4 – Edit User Profile 31
    UC #6 – Search Trip 32
    UC #12 – Reschedule Ticket 33
    Strategy to Conduct Unit Testing 34
9. PROJECT MANAGEMENT & PLAN OF WORK 35
    9.1 Gantt Chart 36
10. REFERENCES 38

## 1. INDIVIDUAL CONTRIBUTIONS BREAKDOWN

| Responsibility Level | Team Member Names | | | | |
|---|---|---|---|---|---|
| | **Azriel Cuellar** | **Daniel Garcia** | **Victor Tillet** | **Levi Coc** | **Floyd Ack** |
| Sec. 1: Analysis and Domain Modeling | 20% | 20% | 20% | 20% | 20% |
| Sec. 2: Interaction Diagrams | 20% | 20% | 20% | 20% | 20% |
| Sec. 3: Class Diagrams and Interface Specification | 20% | 20% | 20% | 20% | 20% |
| Sec. 4: Algorithms and Data Structures | 20% | 20% | 20% | 20% | 20% |
| Sec. 5: User Interface Design and Implementation | 20% | 20% | 20% | 20% | 20% |
| Sec. 6: Design of Tests | 20% | 20% | 20% | 20% | 20% |
| Sec. 7: Project Management and Plan of Work | 20% | 20% | 20% | 20% | 20% |

## 2. ANALYSIS & DOMAIN MODELING

### 2.1 Concept Definitions

UC-1: Register

| Responsibility Description | Concept Name |
|---|---|
| Coordinates the actions of all concepts associated with the use case and delegate the work to other concepts. | Controller |
| Renders the page that is specified after a request is made. | Page Maker |
| HTML document that displays to the actor what actions can be done and required fields to be filled out in the form. | Interface Page |
| Form specifying the data for the database upload request. | Upload Request |
| Verify if all fields of the form were filled out and if it meets the requirements. | Upload Checker |
| Prepare a database query that uploads the user's login credentials. | Database Connection |

*Table 1: Concepts for Register*

UC-6: Select Trip

| Responsibility Description | Concept Name |
|---|---|
| Coordinates the actions of all concepts associated with the use case and delegate the work to other concepts. | Controller |
| HTML document that displays to the actor what actions can be done and required fields to be filled out in the form. | Interface Page |
| Prepares a database update query to alter the seats table by changing the availability of a specific seat. | Database Connection |
| Renders the page that is specified after a request is made. | Page Maker |
| Retrieves the seat identification of the seat the user requests. | Capture Seat Request |

| | |
|---|---|
| Retrieves the credit card information from the form. | Capture Card Information |
| Retrieves the route identification to determine which route the ticket is being purchased for. | Capture Ticket Request |
| Informs the system whether or not the bank was successful with the transaction. | Notifier |
| Point of interaction between the system and the bank's internal system to conduct transactions (not a part of the system being built). | Bank Application Programming Interface |

*Table 2: Concepts for Select Trip*

UC-9: View Payment History

| **Responsibility Description** | **Concept Name** |
|---|---|
| Coordinates the actions of all concepts associated with the use case and delegate the work to other concepts. | Controller |
| Prepare a database query that locates the user's previous payment and booking. | Database Connection |
| Renders the page that is specified after a request is made. | Page Maker |
| HTML document that displays to the actor what actions can be done and required fields to be filled out in the form. | Interface Page |
| Information specifying the user's unique identifier. | Payment Request |

*Table 3: Concepts for Payment History*

UC-10: View Booking History

| **Responsibility Description** | **Concept Name** |
|---|---|
| Coordinates the actions of all concepts associated with the use case and delegate the work to other concepts. | Controller |
| Prepare a database query that locates the user's previous booking and retrieves it. | Database Connection |

| | |
|---|---|
| Renders the page that is specified after a request is made. | Page Maker |
| HTML document that displays to the actor what actions can be done and required fields to be filled out in the form. | Interface Page |
| Information specifying the user's unique identifier to access the user's record. | Booking Request |

*Table 10: Concepts for Booking History*

UC-12: Reschedule Ticket

| **Responsibility Description** | **Concept Name** |
|---|---|
| Coordinates the actions of all concepts associated with the use case and delegate the work to other concepts. | Controller |
| HTML document that displays to the actor what actions can be done and required fields to be filled out in the form. | Interface Page |
| Renders the page that is specified after a request is made. | Page Maker |
| Retrieves the seat identification of the seat the user requests. | Capture Seat Request |
| Retrieves the credit card information from the form. | Capture Card Information |
| Retrieves the route identification to determine which route the ticket is being purchased for. | Capture Ticket Request |
| Informs the system whether or not the bank was successful with the transaction. | Notifier |
| Point of interaction between the system and the bank's internal system to conduct transactions (not a part of the system being built). | Bank Application Programming Interface |
| Prepares database query able to delete a database record, which best fulfils the user's request and executes it. | Delete Database Record |

| | |
|---|---|
| Prepares database query able to add a database record and executes it. | Add Database Record |

*Table 12: Concepts for Reschedule Ticket*

## 2.2 Association Definitions

UC-1: Register

| Concept Pair | Association Description | Association Name |
|---|---|---|
| Page Maker ↔ Interface Page | Page Maker prepares the Interface Page to be displayed | prepares |
| Controller ↔ Page Maker | Controller passes the specific page request to Page Maker | convey request |
| Controller ↔ Upload Checker | Controller passes the form information to Upload Checker to check if the information supplied meets the requirements. | convey request |
| Controller ↔ Interface Page | Controller updates the Interface Page with the requested web page. | posts |
| Upload Request ↔ Controller | Upload Request provides Controller with the form information | receives |
| Upload Checker ↔ Database Connection | Upload Checker uploads the checked information to the database. | saves data to |

*Table 6: Associations for Register*

UC-6: Select Trip

| Concept Pair | Association Description | Association Name |
|---|---|---|
| Controller ↔ Interface Page | Controller updates the Interface Page with the requested web page. | posts |

| | | |
|---|---|---|
| Controller ↔ Capture Seat Request | Capture Seat Request provides the Controller with the seat identification number. | receives |
| Controller ↔ Capture Card Information | Capture Card Information provides the Controller with the form information. | receives |
| Controller ↔ Capture Ticket Request | Capture Ticket Request provides the Controller with the route identification number. | receives |
| Controller ↔ Database Connection | Controller provides the Database connection with the seat identification | convey-request |
| Controller ↔ BankAPI | Controller provides the BankAPI with the credit card information of the user. | forward information |
| Controller ↔ Page Maker | Controller passes the specific page request to Page Maker and receives back page prepared for displaying. | convey-request |
| Page Maker ↔ Interface Page | Page Maker prepares the Interface Page | prepares |
| Notifier ↔ Controller | Notifier informs the Controller whether or not the transaction was completed. | notifies |
| BankAPI ↔ Notifier | BankAPI passes Notifier an indication if the transaction was successful or not. | forwards |

*Table 7: Associations for Select Trip*

UC-9: View Payment History

| Concept Pair | Association Description | Association Name |
|---|---|---|
| Page Maker ↔ Interface Page | Page Maker prepares the Interface Page to be displayed. | prepares |
| Controller ↔ Payment Request | Payment Request provides the Controller with the necessary identifier to locate the user's payment history. | receives |
| Controller ↔ Interface Page | Controller updates the Interface Page with the requested web page. | posts |
| Controller ↔ Database Connection | Controller passes Payment Request query to Database Connection | convey-request |
| Controller ↔ Page Maker | Controller passes requests to Page Maker and receives back pages prepared for displaying. | convey-request |
| Database Connection ↔ Page Maker | Database Connection passes the retrieved data to Page Maker to render them for display. | provides data |

*Table 8: Associations for Payment History*

UC-10: View Booking History

| Concept Pair | Association Description | Association Name |
|---|---|---|
| Page Maker ↔ Interface Page | Page Maker prepares the Interface Page to be displayed. | prepares |
| Controller ↔ Booking Request | Payment Request provides the Controller with the necessary identifier to locate the user's booking history. | receives |
| Controller ↔ Interface Page | Controller updates the Interface Page with the requested web page. | posts |

| | | |
|---|---|---|
| Controller ↔ Database Connection | Controller passes Payment Request query to Database Connection | convey-request |
| Controller ↔ Page Maker | Controller passes requests to Page Maker and receives back pages prepared for displaying. | convey-request |
| Database Connection ↔ Page Maker | Database Connection passes the retrieved data to Page Maker to render them for display. | provides data |

*Table 9: Associations for Booking History*

UC-12: Reschedule Ticket

| **Concept Pair** | **Association Description** | **Association Name** |
|---|---|---|
| Controller ↔ Interface Page | Controller updates the Interface Page with the requested web page. | posts |
| Controller ↔ Capture Seat Request | Capture Seat Request provides the Controller with the seat identification number. | receives |
| Controller ↔ Capture Card Information | Capture Card Information provides the Controller with the form identification. | receives |
| Controller ↔ Capture Ticket Request | Capture Ticket Request provides the Controller with the route identification number. | receives |
| Controller ↔ Retrieve Previous Order | Retrieve Previous Order provides the Controller with the ticket order identification number. | receives |
| Controller ↔ Delete Database Record | Controller passes delete request to Delete Database Record | convey-request |

| | | |
|---|---|---|
| Controller ↔ Add Database Record | Controller passes add request to Add Database Record | convey-request |
| BankAPI ↔ Notifier | BankAPI passes Notifier an indication if the transaction was successful or not. | forwards |

*Table 10: Associations for Reschedule Ticket*

## 2.3 Attribute Definitions

UC-1: Register

| Concept | Attributes | Attribute Description |
|---|---|---|
| Upload Request | user information | First name, last name, email, address, phone number, and password. |

*Table 11: Attributes for Register*

UC-6: Select Trip

| Concept | Attributes | Attribute Description |
|---|---|---|
| Capture Card Information | credit card information | Account Number |
| Capture Card Request | seat's identification | It is used to identify the seat the user would like to buy a ticket for. |
| Capture Ticket Request | route's identification | It is used to identify the route the user would like to buy a ticket for. |

*Table 12: Attributes for Select Trip*

UC-9: View Payment History

| Concept | Attributes | Attribute Description |
|---|---|---|
| Payment Request | user’s identification | It is used to determine the user’s previous purchases. |

*Table 13: Attributes for Payment History*

UC-10: View Booking History

| Concept | Attributes | Attribute Description |
|---|---|---|
| Booking Request | user’s identification | It is used to determine the user’s previous bookings. |

*Table 14: Attributes for Booking History*

UC-12: Reschedule Ticket

| Concept | Attributes | Attribute Description |
|---|---|---|
| Capture Card Information | credit card information | Account Number |

| | | |
|---|---|---|
| Capture Seat Request | seat's identification | It is used to identify the seat the user would like to buy a ticket for. |
| Capture Ticket Request | route’s identification | It is used to identify the route the user would like to buy a ticket for. |
| Retrieve Previous Order | ticket order’s identification | It is used to identify the ticket order the user would like to request. |

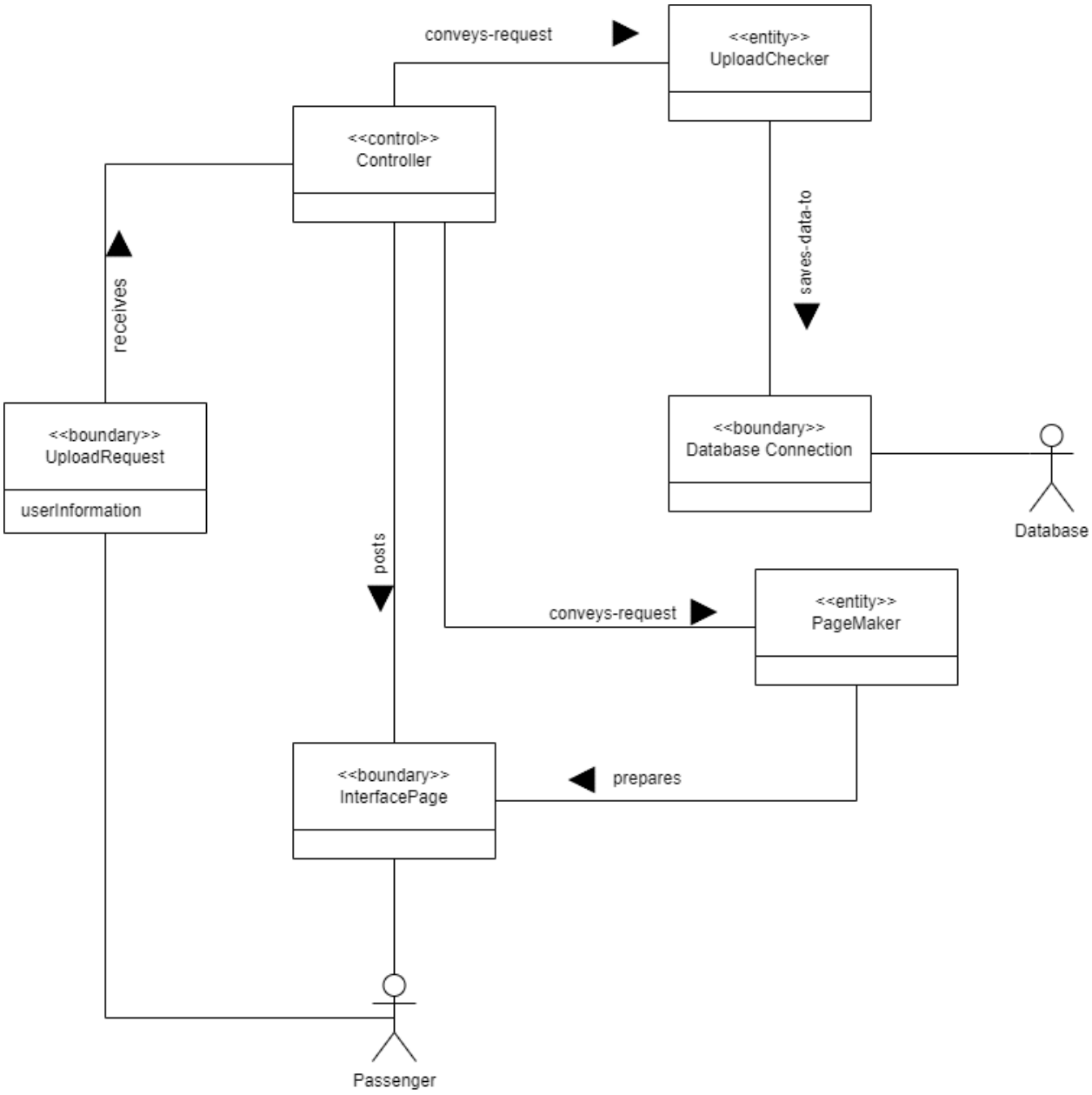
*Table 15: Attributes for Reschedule Ticket*

## 2.4 Traceability Matrix

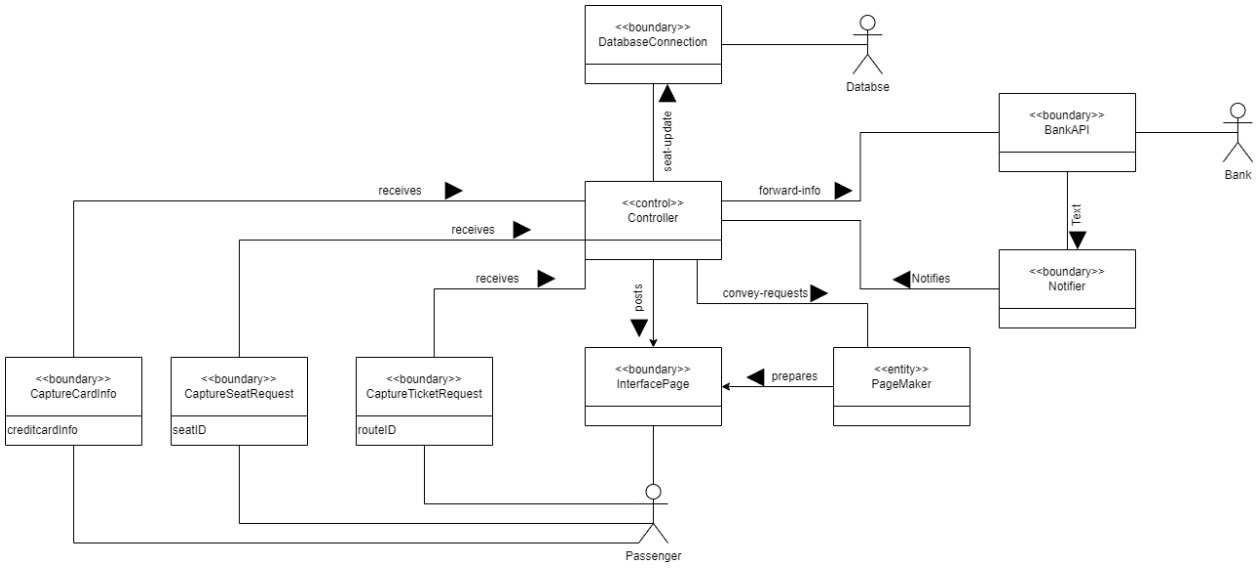
| | | Domain Concepts | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Use Cases | P W | Controller | Page Maker | Interface Page | Upload Request | Upload Checker | Database Connection | Capture Seat Request | Capture Card Information | Notifier | BankAPI | Payment Request | Booking Request | Capture Ticket Request | Add Database Record | Delete Database Record |
| UC-1 | 36 | X | X | X | X | X | X | | | | | | | | | |
| UC-2 | 28 | X | X | X | | X | X | | | | | | | | | |
| UC-3 | 29 | X | X | X | | | X | | | | | | | | | |
| UC-4 | 34 | X | X | X | X | | X | | | | | | | | | |
| UC-5 | 29 | X | X | X | | | X | | | | | | | | | |
| UC-6 | 36 | X | X | X | | | X | X | X | X | X | | | | | |
| UC-7 | 36 | X | X | X | | | X | X | | | | | | | | |
| UC-8 | 53 | X | X | X | | | X | | X | X | X | | | | | |
| UC-9 | 26 | X | X | X | | | X | | | | | X | | | | |
| UC-10 | 42 | X | X | X | | | X | | | | | | X | | | |
| UC-11 | 37 | X | X | X | | | X | | | | X | | | | | |
| UC-12 | 45 | X | X | X | | | X | X | X | X | X | | | X | X | X |
| UC-13 | 31 | X | X | X | | | X | X | | | | | X | | | |
| UC-14 | 21 | X | X | X | | | | | | | | | | | | |
| UC-15 | 15 | X | X | X | | | | | | | | | | | | |
| UC-16 | 26 | X | X | X | | | | | | | | | | | | |
| UC-17 | 28 | X | X | X | | | | | | | | | | | | |

## 2.5 Domain Models

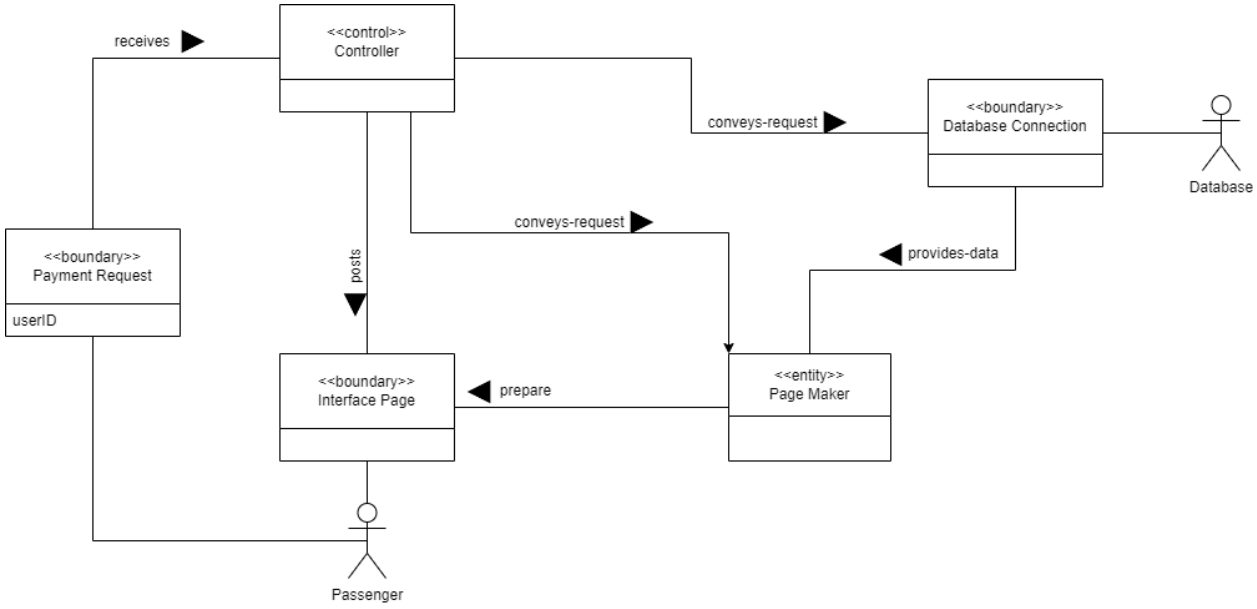
UC-1: Register



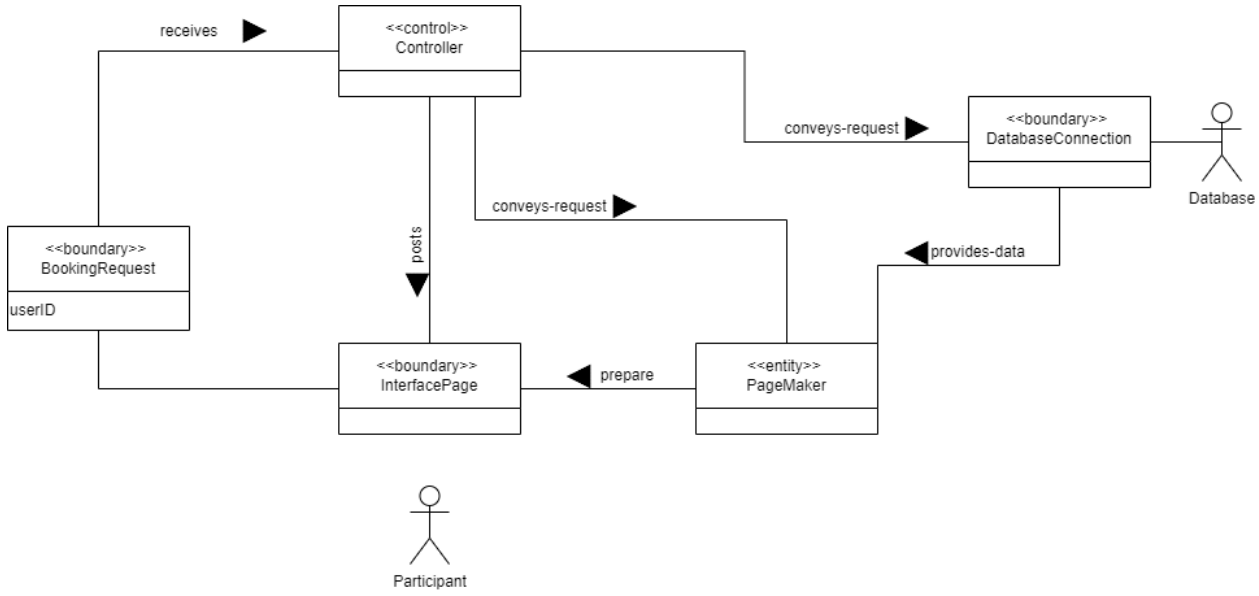
## UC-6: Select Trip



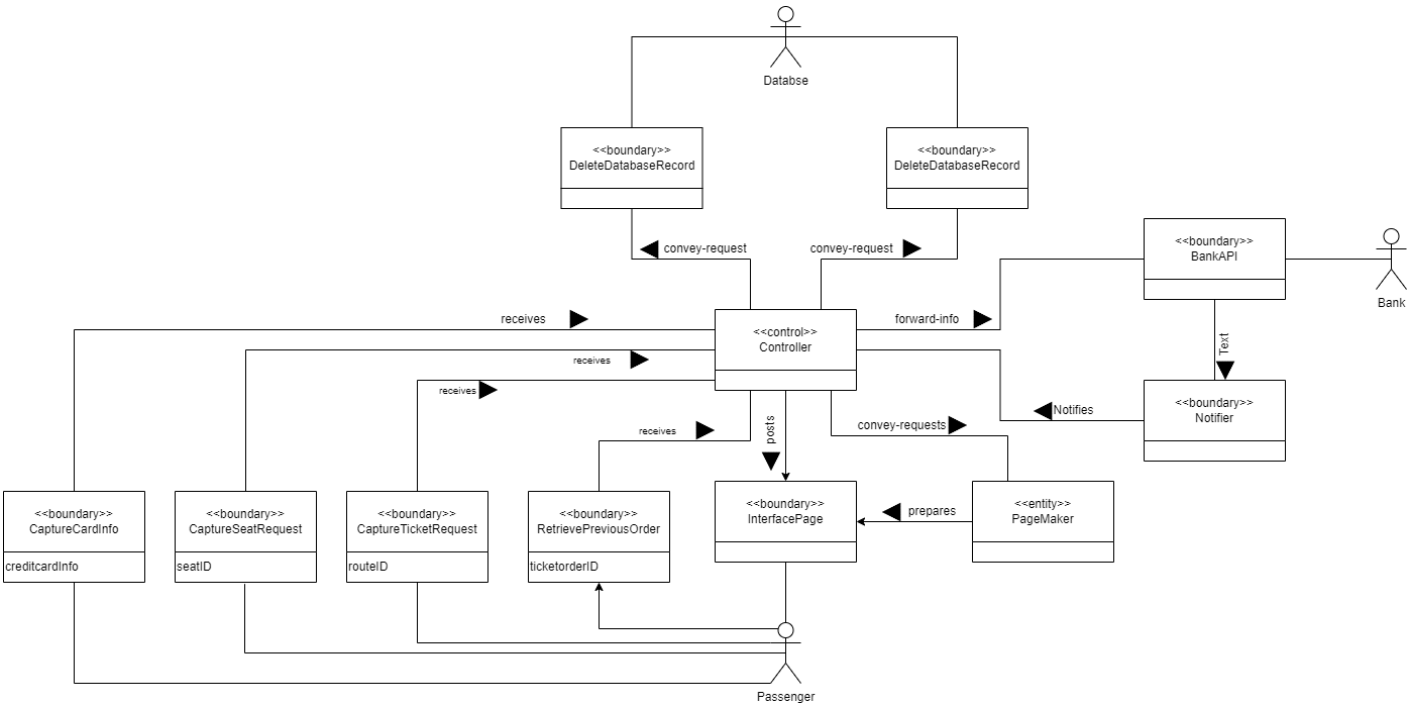

## UC-9: View Payment History



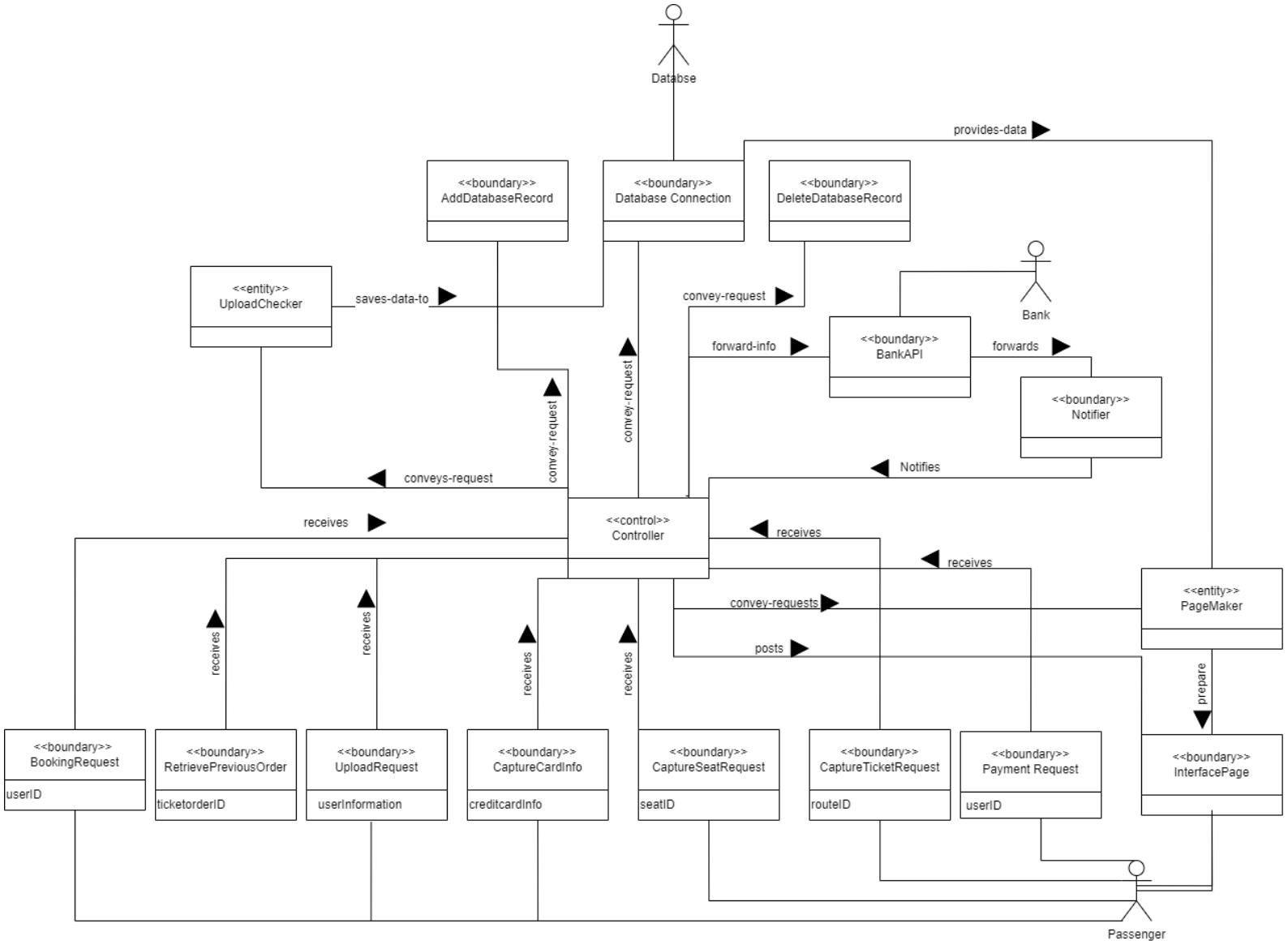
## UC-10: View Booking History




## UC-12: Reschedule Ticket



Domain Model of the Entire System



## 3. SYSTEM OPERATION CONTRACTS

*Table 16: System Operation Contract for Register*

| | |
|---|---|
| **Contract Name** | updateUserInfo(fname, lname, email, phone, num, password) |
| **Cross Reference** | Use Case: Register |
| **Responsibility** | Add user's information into the system |
| **Type** | System |
| **Precondition** | Internet is available and the user hasn't registers with the same email before. |
| **Postcondition** | User is now registered into the system. |

*Table 17: System Operation Contract for Register & Reschedule Ticket*

| | |
|---|---|
| **Contract Name** | checkSeat(seat identification) |
| **Cross Reference** | Use Case: Select Trip & Reschedule Ticket |
| **Responsibility** | Check if the seat is available to be selected. |
| **Type** | System |
| **Precondition** | User has selected a ticket to purchase. |
| **Postcondition** | The user is informed whether or not the seat selection has been purchased before the transaction was completed. |

*Table 18: System Operation Contract for Payment History*

| | |
|---|---|
| **Contract Name** | requestPaymentHistory(user identification) |
| **Cross Reference** | Use Case: Payment History |
| **Responsibility** | Locate all records for the previous purchases the user has completed. |
| **Type** | System |
| **Precondition** | User is registered to the system and has internet connection. |
| **Postcondition** | Request is completed and records are returned. |

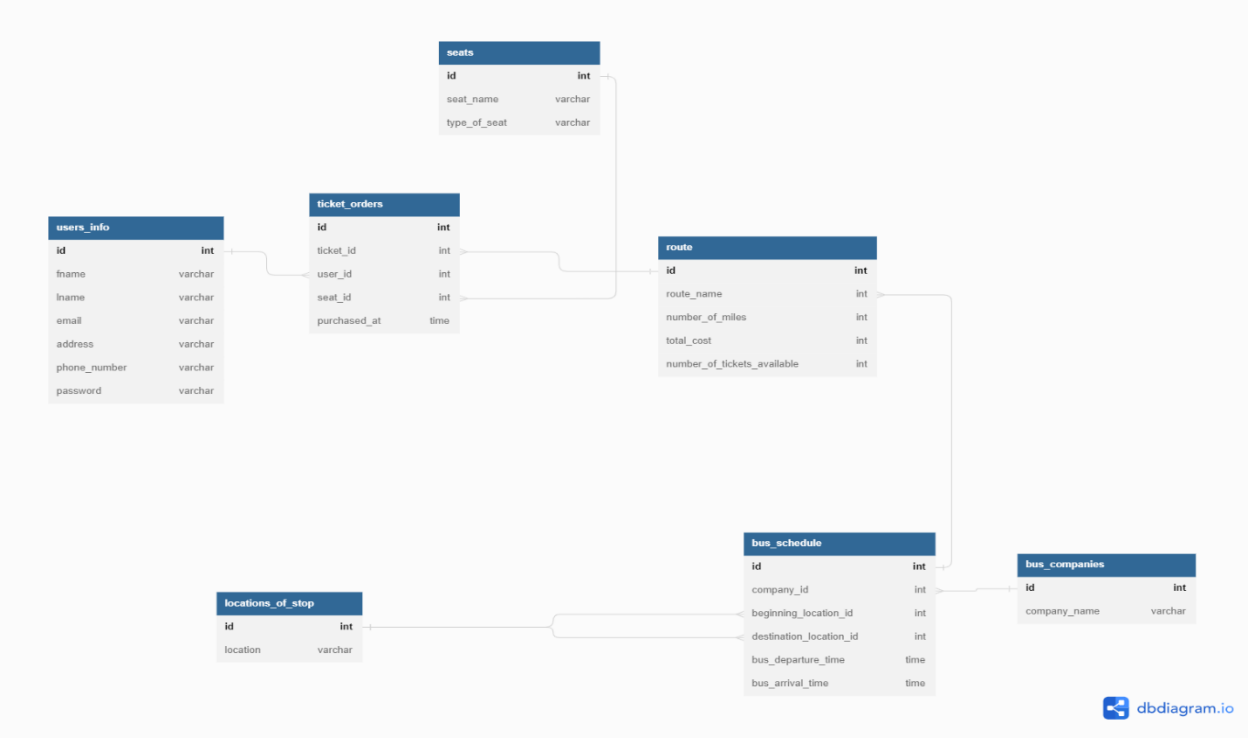
| | |
|---|---|
| **Contract Name** | displayPaymentHistory(user identification) |
| **Cross Reference** | Use Case: Payment History |
| **Responsibility** | Display the Payment History unto the interface as specified. |
| **Type** | System |
| **Precondition** | Payment history is known. |
| **Postcondition** | The user's payment history is displayed. |

*Table 19: System Operation Contract for Reschedule Ticket*

| | |
|---|---|
| **Contract Name** | requestPreviousOrder(user identification, order identification) |
| **Cross Reference** | Use Case: Reschedule Ticket |
| **Responsibility** | Locate a previous purchase completed by the user. |
| **Type** | System |
| **Precondition** | The user has purchased a ticket before. |
| **Postcondition** | Request is completed. |

| | |
|---|---|
| **Contract Name** | removePreviousOrder(user identification, order identification) |
| **Cross Reference** | Use Case: Reschedule Ticket |
| **Responsibility** | Delete the previous record purchased by the user. |
| **Type** | System |
| **Precondition** | Request was completed. |
| **Postcondition** | The user’s previous purchase was removed. |

## 4. DATA MODEL & PERSISTENT STORAGE




The system was designed with the objective of retaining all data throughout its lifespan. In the future, an archive table may be implemented to accommodate records exceeding a year. To optimize data management, the system utilizes a relational database that leverages unique identification numbers for each record, as opposed to string values.

Notably, the “locations_of_stop” table is expected to undergo minimal changes since stops along pickup routes and the number of bus companies are unlikely to fluctuate frequently. Meanwhile, the “bus schedule” table houses essential details such as route start and end points, departure and arrival times, and the company facilitating the route. This information is then linked to the “route” table, which captures data on cost, total distance, and available tickets that reset after each route completion.

The “ticket_orders” table serves as a repository for all user purchases, and its contents remain accessible for historical reference, with no data deletion anticipated.

## 5. INTERACTION DIAGRAMS

*Figure 1: Sequence Diagram for User Registration (UC-1)*

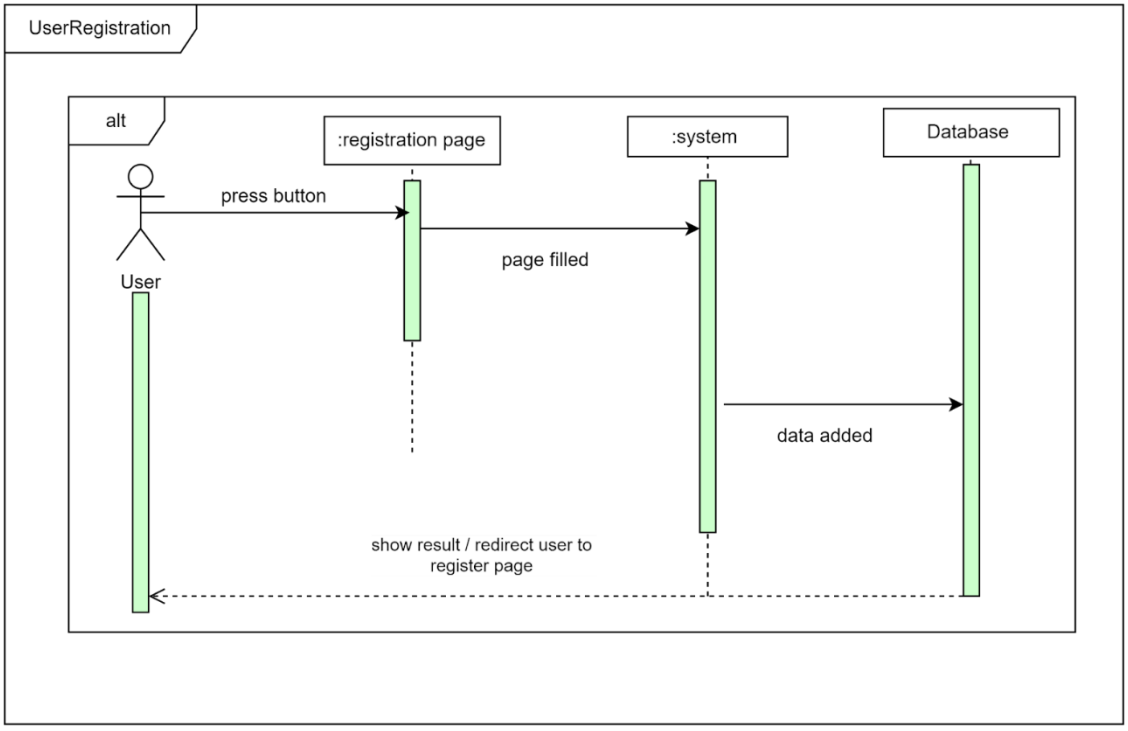



*Figure 2: Sequence Diagram for Selected Trip (UC-6)*

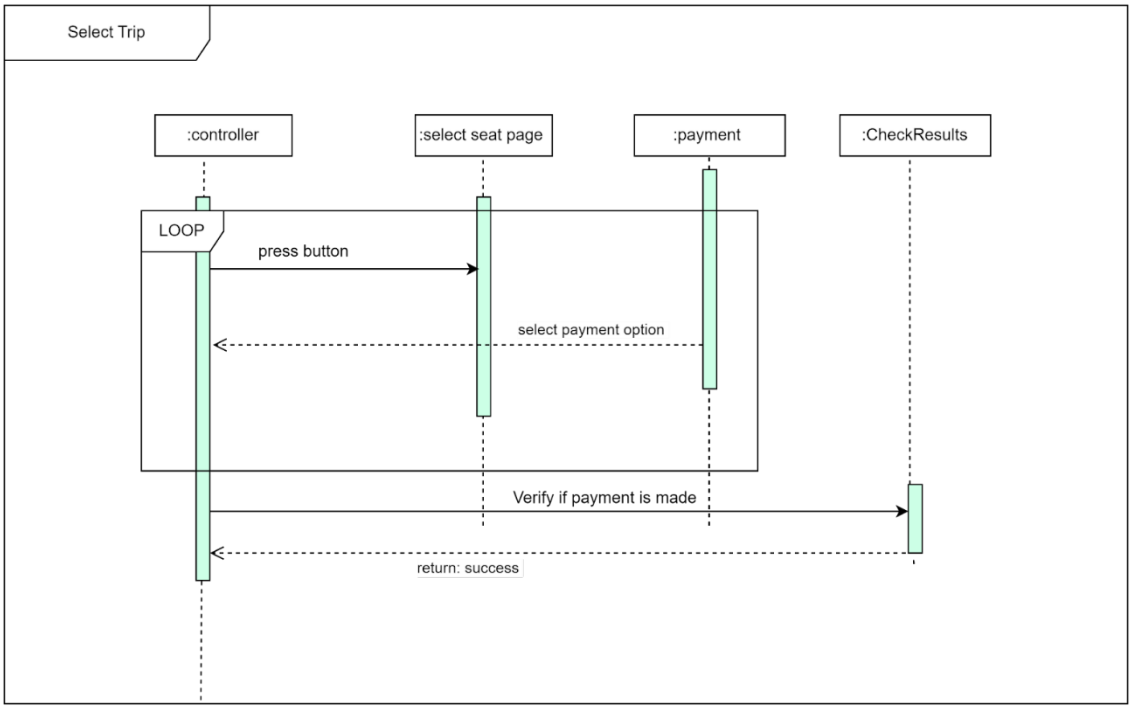


*Figure 3: Sequence Diagram for Payment History (UC-9)*

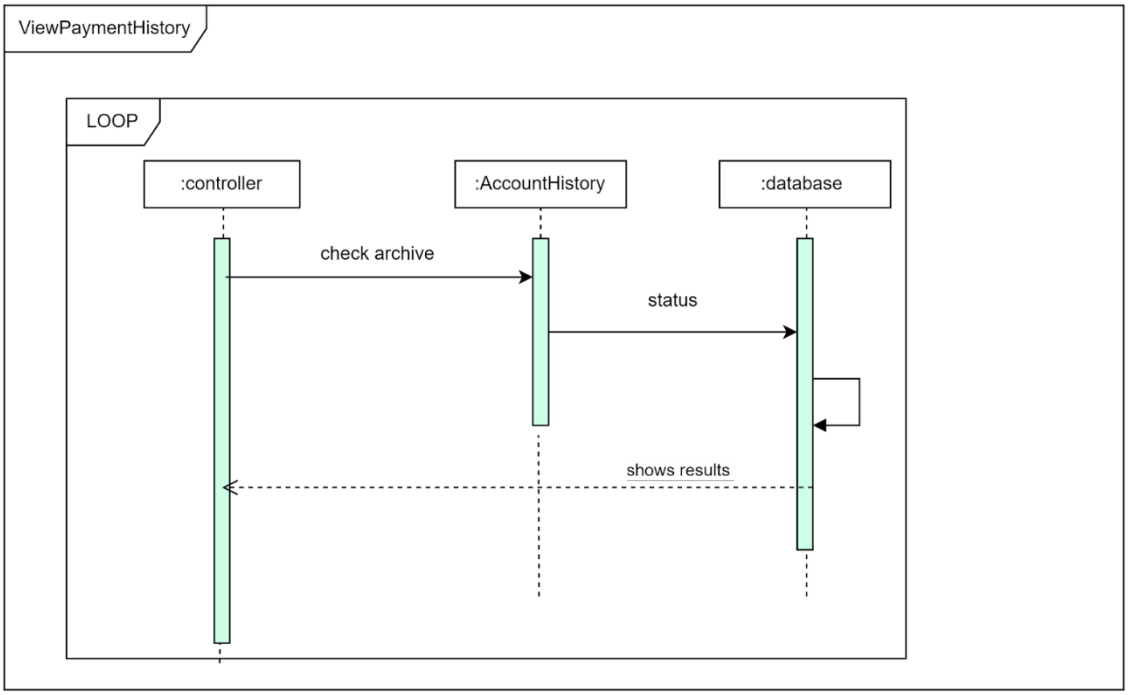



*Figure 4: Sequence Diagram for Booking History (UC-10)*

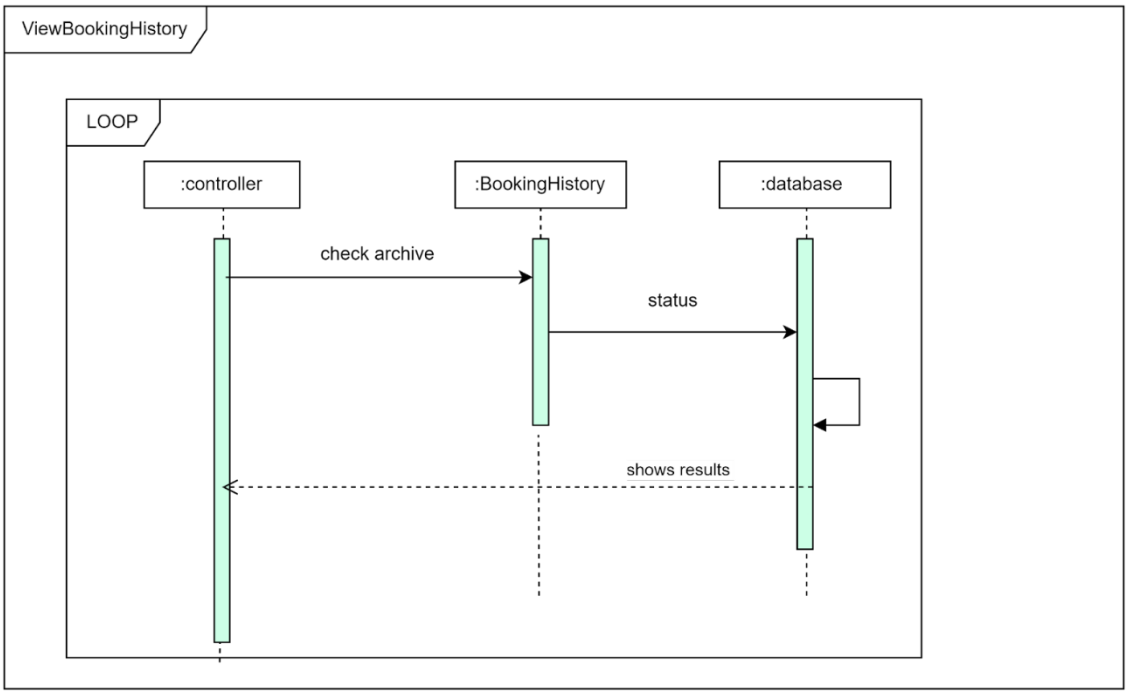
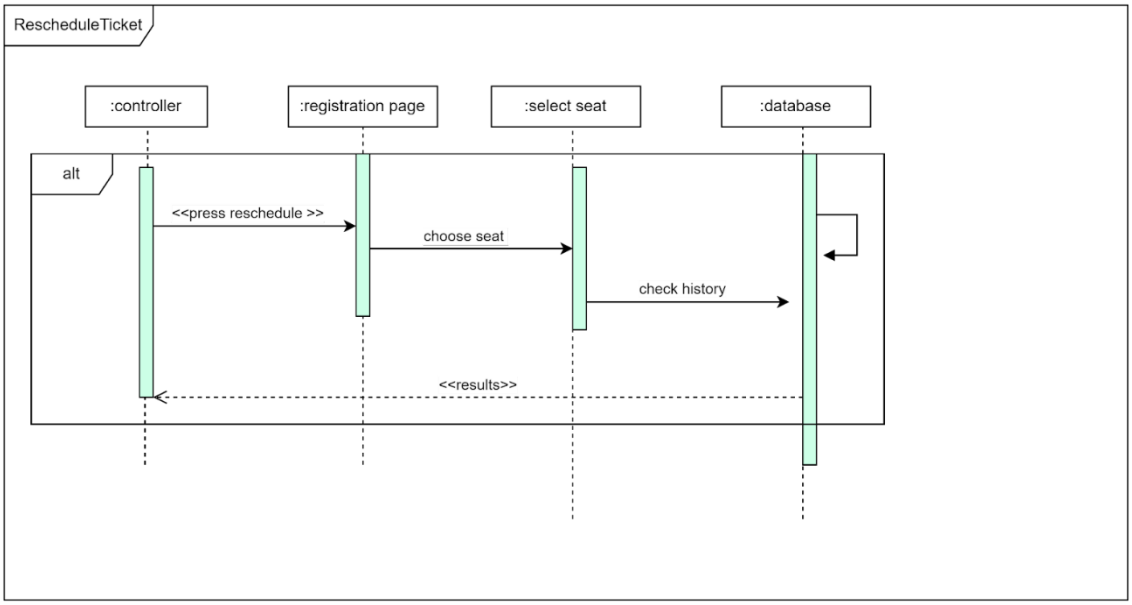
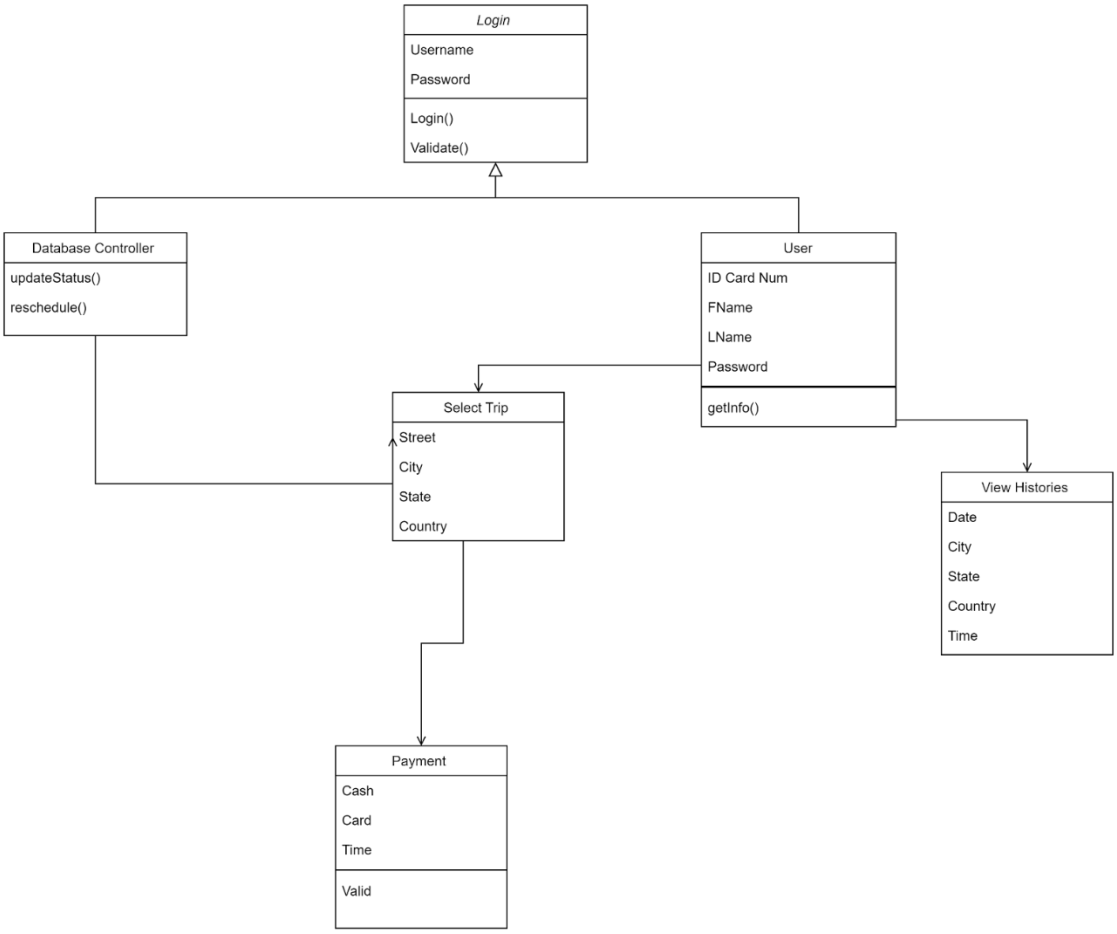


*Figure 12: Sequence Diagram for Reschedule Ticket (UC-12)*



# 6. CLASS DIAGRAM & INTERFACE SPECIFICATION

## 6.1 Class Diagram




## 6.2 Data Types and Operation Signature

| *Login* |
|---|
| Username-string |
| Password - string |
| Login() - void |
| Validate() - void |

| User |
|---|
| ID Card Num- int |
| FName- string |
| LName -string |
| Password - string |
| getInfo() - void |

| Database Controller |
|---|
| updateStatus() -void |
| reschedule() - void |

| Select Trip |
|---|
| Street - string |
| City - string |
| State - string |
| Country - string |

| Payment |
|---|
| Cash - int |
| Card - void |
| Time - int |
| Valid - void |

| View Histories |
|---|
| Date -int |
| City- string |
| State - string |
| Country - string |
| Time -int |

## 6.3 Traceability Matrix

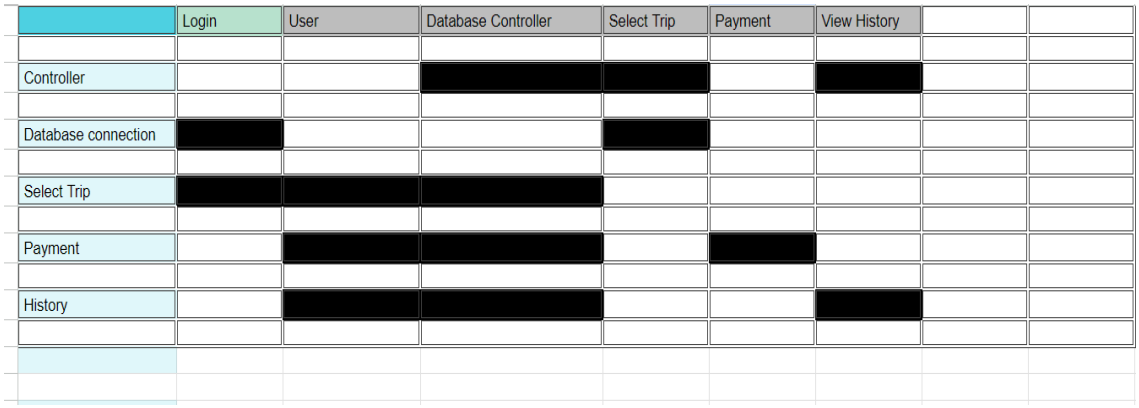



*Figure showing the Traceability Matrix*

**Login:**

The login enables the user to access the mobile application and communicates with the database connection to determine whether the username and password are valid. It likewise utilizes the select excursion connection point to provoke the client to sign in and select where he/she needs to go.

**User:**

The User is the main branch of the domain models.

**Database Controller:**

Throughout the system's lifespan, it is in charge of keeping in touch with the database and making attempts to reconnect if the connection is lost. Additionally, it must respond to each function call requesting database access. The connection serves as the channel for all queries.

**Select Trip:**

Evolved from a controller that dealt with specific controller locations or users.

**Payment:**

The user's method of payment, whether by cash or credit card, is the responsibility of the Payment.

**History:**

Users can view past and current payments made and tickets purchased using the user's method of history.

## 7. DATA STRUCTURES

The Bus Commute Companion application will utilize several data structures, namely arrays, linked lists, hash tables, and JavaScript Object Notation (JSON). The following section provides a detailed explanation of each of these data structures.

### 1. Arrays

In the Bus Commute Companion application, arrays will be used to store and retrieve information such as the list of available bus routes, bus schedules, and ticket prices. This data structure is useful because it allows a single variable to store multiple values of the same type, making it easier to manage and manipulate the data. The PHP language will be used as a back end to interact with the database and retrieve the data, such as the list of available routes or schedules, which will be stored in arrays. When a user searches for a bus route, the array will be queried, and the matching route details will be displayed to the user.

### 2. Linked Lists

Linked Lists will be used in Bus Commute Companion to manage the data of passengers, trips, and bookings. For example, a linked list of passengers can be used to store the information of all passengers on a particular trip. Each node in the linked list will contain information such as the passenger's name, age, and contact information. Linked list will also manage the queue of passengers waiting to purchase a ticket and track the history of bookings and trips for each passenger. Each passenger will have their own linked list containing their booking history, which will be updated when they make a new booking or modify existing ones.

### 3. Hash Tables

In the Bus Commute Companion application, hash tables will be useful for storing and accessing data related to routes and schedules, such as the departure time, arrival time, and stop locations. Since hash tables can search for data quickly based on the key, it will improve the performance and efficiency of the application, making it more user-friendly for customers.

### 4. JavaScript Object Notation

JavaScript Object Notation (JSON) will be used in Bus Commute Companion to facilitate data exchange between the front-end and back-end of the application. This data structure will be used to store and transmit complex data in a lightweight and efficient manner. For example, when a user submits a search query for available buses, the front-end will encode the data into a JSON object and send it to the back end via an HTTP request. The back end will then process the request and respond with a JSON object containing the relevant data, such as bus schedules and pricing information. The front-end will then decode the JSON object and display the data to the user in a user-friendly format.

5. **Criteria used to choose the Data Structure**

The criteria used to choose these data structures for Bus Commute Companion include:

1. Efficiency and speed of data retrieval: Since the application is targeting availability and reliability, search queries should yield almost immediate results. As a result, arrays will be used because they are fast and easy to traverse.
2. Security: To ensure that user data is secure, we will use hash tables to store sensitive information such as passwords. Hash tables are more secure than arrays as they use a hashing function to store data in a way that makes it difficult to reverse engineer.
3. Scalability: To ensure the app can handle a large number of users and data, arrays and hash tables can be easily scaled up by adding more memory or processing power, while linked lists can be optimized for specific use cases to improve performance.
4. Flexibility: JSON will be used for exchanging data between the server and the client, as it provides a flexible and lightweight way to encode and transmit data.

## 8. DESIGN OF TESTS

We will be testing our five Main Use-Cases. We need these tests so we can evaluate what the user will input. Overall, we cannot test everything that a user might want to do, so the following will be what we anticipate the user might try.

### UC #10 – View Booking History

Test Case: Test for a Valid User ID

Description: This test case checks if the function displays the correct booking history for a valid user ID with existing bookings.

Test Steps:

a. Set a valid user ID that exists in the database with a booking history.
b. Call a function with a valid user ID.
c. Capture the output of the function using output buffering.
d. Compare the expected output with the actual output of the function.
e. Assert that the desired output and actual output are equal.

Test Case Name: Test for an invalid User ID

Description: This test case checks if the function displays the correct message when an invalid user ID that does not exist in the database is provided.

Test Steps:

a. Set an invalid user ID that does not exist in the database.
b. Call a function with a invalid user ID.
c. Capture the output of the function using output buffering.
d. Compare the expected output with the actual output of the function.
e. Assert that the desired output and actual output are equal.

Test Case Name: Test for an empty string

Description: This test case checks if the function displays the correct message when an empty string is provided as the user ID.

Test Steps:

a. Set an empty string as the user ID.
b. Call the function with the empty string user ID.
c. Capture the output of the function using output buffering.
d. Compare the expected output with the actual output of the function.
e. Assert that the desired output and actual output are equal.

| Test Case Function | Inputs | Actual Result | Expected Result |
|---|---|---|---|
| testValidUserIdWithHistory() | $userid = “1”; | True | False |
| testInvalidUserIdNonExistent() | $userid = “5678”; | False | False |
| testInvalidUserIdEmptyString() | $userid = “” | False | False |

All of these will be using the PHP unit function assertEqual(). In order for the test case to pass, the actual result and the expected result must be the same. We made sure the expected results and actual results are the same which creates an overall result where everything passes.

**UC #9 – View Payment History**

Test Case: Test for a Valid User ID

Description: This test case checks the function’s behaviour when a valid user ID is provided. It expects the function to return a string containing payment information for the user with ID 1.

Test Case Name: Test for an invalid User ID

Description: This test case checks the function’s behaviour when an invalid user ID (i.e., a non-existent user ID) is provided. It expects the function to return a string indicating that no payment information was found for the user.

Test Case Name: Empty Payment Information

Description: This test case checks the function’s behaviour when a valid user ID is provided, but no payment information is associated with the user. It expects the function to return a string indicating that no payment information was found for the user.

Test Case Name: Null user-id

Description: This test case checks the function’s behaviour when a null user ID is provided. It expects the function to return a string indicating that no payment information was found for the user.

| Test Case Function | Inputs | Actual Result | Expected Result |
|---|---|---|---|
| testValidUserId() | paymentinfo(1) | True | True |
| testInvalidUserId() | paymentinfo(1000) | False | False |
| testNoPaymentForUser() | paymentinfo(2) | False | False |
| testNullUserId() | paymentinfo(0) | False | False |

All of these will be using the PHP unit function assertEqual(). In order for the test case to pass, the actual result and the expected result must be the same. We made sure the expected results and actual results are the same which creates an overall result where everything passes.

**UC #4 – Edit User Profile**

Test Case: Update User Info with Valid Input

Description: This test case checks if the function "edithprofile" updates the user's information with valid inputs (i.e., user ID, first name, last name, phone number, and address) and returns true, indicating that the update was successful.

Test Case Name: Update User Info with Empty First Name

Description: This test case checks if the function "edithprofile" returns false when the first name is empty, indicating that the update was unsuccessful.

Test Case Name: Update User Info with Invalid User Id

Description: This test case checks if the function "edithprofile" returns false when an invalid user ID is provided, indicating that the update was unsuccessful.

Test Case Name: Update User Info with Invalid Phone Number

Description: This test case checks if the function “edithprofile” returns false when an invalid phone number is provided, indicating that the update was unsuccessful.

| Test Case Function | Inputs | Actual Result | Expected Result |
|---|---|---|---|
| testUpdateUserInfoWithValidInput() | $userid = 1;<br>$fname = ‘walter’;<br>$lname = ‘white’;<br>$phone = ‘4443333844’;<br>$address = ‘123 Aroura Lane’; | True | True |
| testUpdateUserInfoWithEmptyFirstName() | $userid = 1;<br>$fname = ‘’;<br>$lname = ‘doe’; | False | False |

| | | | |
|---|---|---|---|
| | $phone = ‘123456789’;<br>$address = ‘123 Main St’; | | |
| testUpdateUserInfoWithInvalidUserId() | $userid = invalid;<br>$fname = ‘John’;<br>$lname = ‘doe’;<br>$phone = ‘1234567890’;<br>$address = ‘123 Main St’; | False | False |
| testUpdateUserInfoWithInvalidPhoneNumber() | $userid = null;<br>$fname = ‘John’;<br>$lname = ‘doe’;<br>$phone = ‘Invalid’;<br>$address = ‘123 Main St’; | False | False |

Each test case includes input values, expected results, and actual results. The PHPUnit framework is used to perform the assertions, which compare the expected and actual results to ensure that the function "edithprofile" is working as intended passes.

**UC #6 – Search Trip**

Test Case: No trips found

Description: This test case checks if the searchTrips function returns the "No trips found" message when no trips are found for the given input parameters.

Test Case Name: Invalid location IDs

Description: This test case checks if the searchTrips function returns the "No trips found" message when invalid location IDs are given as input parameters.

Test Case Name: Invalid time format

Description: This test case checks if the searchTrips function returns the "No trips found" message when invalid time formats are given as input parameters.

Test Case Name: Empty parameters

Description: This test case checks if the searchTrips function returns the “No trips found” message when empty parameters are given as input.

| Test Case Function | Inputs | Actual Result | Expected Result |
|---|---|---|---|
| testValidSearch() | searchTrips(“08:00:00”, “08:55:00”, 1, 2); | True | True |
| testNoTripsFound() | searchTrips(“09:00:00”, “11:00:00”, 2, 3); | False | False |
| testInvalidLocationId() | searchTrips(“09:00:00”, “11:00:00”, “invalid”,”location”2); | False | False |
| testInvalidTimeForm() | searchTrips(“9:00”, “11:00”, 1, 2); | False | False |
| testEmptyParameters() | searchTrips(“”, “”, “”, ””); | False | False |

These test cases cover various scenarios that could occur when using the searchTrips function, and they will help ensure that the function works correctly and reliably.

**UC #12 – Reschedule Ticket**

Test Case: Valid Input

Description: This test case ensures the update trip function returns true with valid input. In this case, valid unput is defined as a non-empty ticket and bus schedule id.

Test Case Name: Invalid Bus Schedule Id

Description: This test case ensures the update trip function returns false when an empty bus schedule id is provided. This simulates the scenario where the user forgets to enter a bus schedule id value, and the function is called with an empty string.

Test Case Name: Invalid Id

Description: This test case ensures the update trip function returns false when an empty ticket id is provided. This simulates the scenario where the user forgets to enter a ticket id value, and the function is called with an empty string.

| Test Case Function | Inputs | Actual Result | Expected Result |
|---|---|---|---|
| testValidInput() | $ticketid = 1;<br>$busscheduleid = 1; | True | True |
| testInvalidBusScheduleId() | $ticketid = 1;<br>$busscheduleid = “”; | False | False |
| testInvalidId() | $ticketid = “”;<br>$busscheduleid = 5; | False | False |

These test cases aim to validate the correct behaviour of the updatetrip function in different scenarios and ensure that the function handles invalid input gracefully by returning false instead of causing errors or exceptions.

**Strategy to Conduct Unit Testing**

My strategy for conducting integration testing would involve the following steps:

a. Identify the components: The first step is to identify the different components of the system that need to be tested.

b. Define integration points: Once the components have been identified, the next step is to define the integration points, which are the places where the members interact with each other.

c. Develop integration test cases: Based on the integration points, develop a set of integration test cases that verify the interaction between the components. The test cases should be designed to cover all possible scenarios during the interaction between the members.

d. Prioritize test cases: Prioritize the integration test cases based on their importance and risk level. Test cases that cover critical functionality or high-risk areas of the system should be given higher priority.

e. Conduct tests: Conduct the integration tests, starting with the highest-priority test cases first. The tests should be conducted in a controlled environment to ensure that the results are reliable and accurate.

f. Debug and fix issues: If any issues are identified during the integration tests, they should be logged, investigated, and fixed before proceeding with the next set of tests.

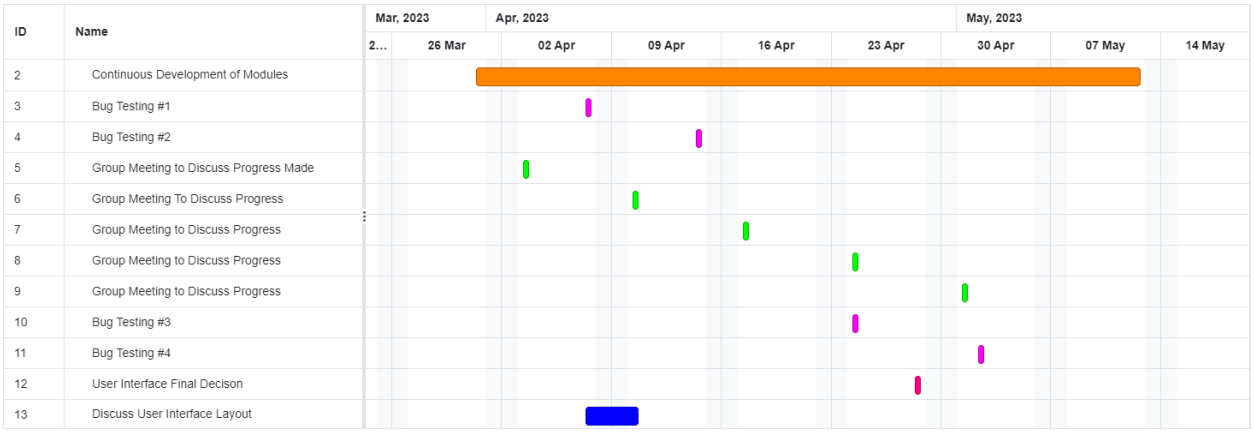
g. Report results: After completing the integration tests, the results should be documented, and a report should be generated. The information should include the tests conducted, any issues identified, and their resolution.

## 9. PROJECT MANAGEMENT & PLAN OF WORK

While completing the report, several issues and complications occurred due to miscommunication, poor grammar, image formats & properties, and maintaining the document's appearance. The central conflict that arose during the development of the document was a miscommunication. When we had initially decided to split the work into multiple sections where multiple individuals would work on a part together, each individually handling a domain model, for example, some members accidentally constructed the domain model of the wrong use case, or they completed the diagram assigned to the other user. However, this did not cause any setbacks in development because correcting this issue was easily solved by allowing the group member to do a different use case. As for any report, grammar was a significant issue regarding the spelling within diagrams and other areas. This was quickly resolved by having the individuals place their work into a grammar checker. Images have always been a problem for us as our diagrams are fairly large, which causes them to fit well in our reports. In report #1, we devised a solution to changing the page orientation, which was done once again.

Currently, several use cases are being implemented, Select Trip, View Payment History, View Booking History, and Reschedule Ticket. The process of completing a bank transaction has not been implemented in any form. Our database tables have been designed, but we are currently discussing whether or not changes will be made to it due to some issues we have observed.

# 9.1 Gantt Chart



| ID | Name |
|---|---|
| 2 | Continuous Development of Modules |
| 3 | Bug Testing #1 |
| 4 | Bug Testing #2 |
| 5 | Group Meeting to Discuss Progress Made |
| 6 | Group Meeting To Discuss Progress |
| 7 | Group Meeting to Discuss Progress |
| 8 | Group Meeting to Discuss Progress |
| 9 | Group Meeting to Discuss Progress |
| 10 | Bug Testing #3 |
| 11 | Bug Testing #4 |
| 12 | User Interface Final Decison |
| 13 | Discuss User Interface Layout |

## 10. BREAKDOWN OF RESPONSIBILITIES

| **Module** | **Team Member** |
|---|---|
| Cancel Ticket | Floyd Ack |
| Generate Ticket | Azriel Cuellar |
| Change Schedule | Victor Tillet (Lead Programmer) |
| Feedback | Daniel Garcia (Lead Programmer) |
| Profile | Levi Coc |

| **Integration Coordination** | **Integration Testing** |
|---|---|
| Victor Tillet | Victor Tillet |
| Daniel Garcia | Daniel Garcia |
| - | Levi Coc |

## 10. REFERENCES

Aweda, Z. I. (2022, March 9). *How to Test PHP Code With PHPUnit*. freeCodeCamp.org. https://www.freecodecamp.org/news/test-php-code-with-phpunit/

Søren Spangsberg Jørgensen. (2021, March 7). *Unit Testing with PHP Unit* [Video]. YouTube. https://www.youtube.com/watch?v=a5ZKCFINUkU