SC2006 FDAE TEAM 2 LAB 4

Black Box Testing

Equivalence Class and Boundary Value Testing

For Black Box Testing, we will be testing the SignUpController control class. SignUpController is responsible for receiving new user details from the SignUpForm, then creating a user and saving it in the database. It checks that the username and email submitted are not already associated with any account in the database. It also ensures that the email is in the correct format, and the password must be at least 8 characters long with a combination of uppercase letters, lowercase letters, numbers and symbols. Email, Username and Password are all discrete values, so they have 1 valid and 1 invalid equivalence class (EC).

Email:
Valid EC: Email with correct format and not already in use.
Invalid EC: Email with incorrect format and/or already in use.

Username:
Valid EC: Username is not already in use.
Invalid EC: Username is already in use.

Password:
Requirements list (tick if fulfilled, cross if not):

1. Password must be at least 8 characters long.
2. Password must include at least one special character.
3. Password must contain at least one number.
4. Password must have at least one uppercase letter.
5. Password must have at least one lowercase letter.

Valid EC: At least 8 characters long with a combination of uppercase letters, lowercase letters, numbers and symbols (requirements 1-5 fulfilled).
Invalid EC: Less than 8 characters long and/or does not contain at least 1 uppercase letter, 1 lowercase letter, 1 number and 1 symbol (at least one of the requirements is not fulfilled).

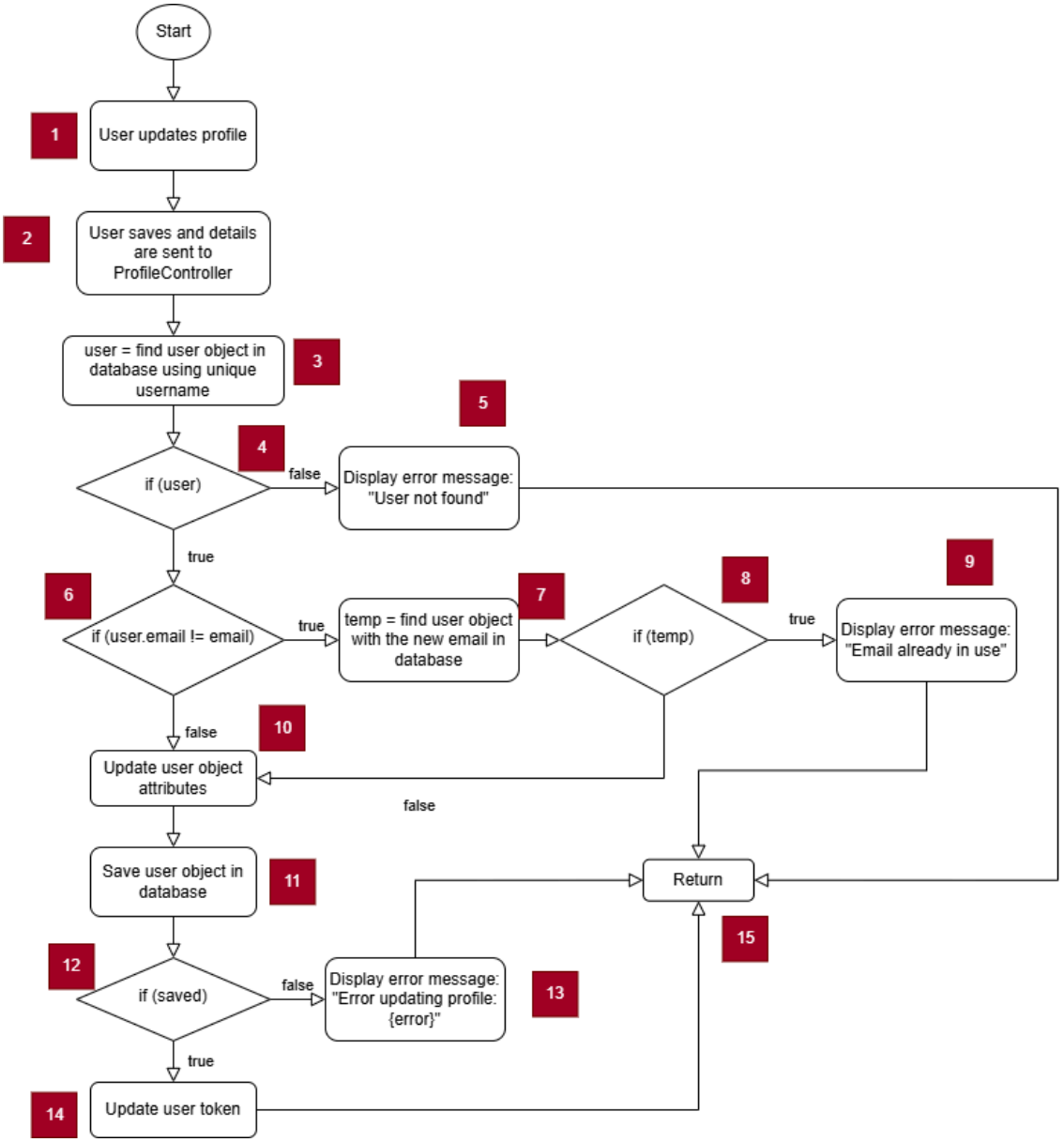
Test Input			Expected Output	Actual Output
Email	Username	Password		
newemail@gmail.com	newuser	Testing@123	No error, redirect to homepage	No error, redirect to homepage
newemail	newuser	Testing@123	Error message: “Please include an ‘@’ in the email address.’newuser’ is missing an ‘@’.”	Error message: “Please include an ‘@’ in the email address.’newuser’ is missing an ‘@’.”
existingemail@gmail.com	newuser	Testing@123	Error message: “Email already in use”	Error message: “Email already in use”
newemail@gmail.com	existinguser	Testing@123	Error message: “Username already exists”	Error message: “Username already exists”
newemail@gmail.com	newuser	Tes@123	Only requirement 1 (8 characters) is not fulfilled. Error message: “Please match the requested format.”	Only requirement 1 (8 characters) is not fulfilled. Error message: “Please match the requested format.”
newemail@gmail.com	newuser	Testing123	Only requirement 2 (1 special character) is not fulfilled. Error message: “Please match the requested format.”	Only requirement 2 (1 special character) is not fulfilled. Error message: “Please match the requested format.”
newemail@gmail.com	newuser	Testing@	Only requirement 3 (1 number) is not fulfilled. Error message: “Please match the requested format.”	Only requirement 3 (1 number) is not fulfilled. Error message: “Please match the requested format.”
newemail@gmail.com	newuser	testing@123	Only requirement 4 (1 uppercase letter) is not fulfilled. Error message: “Please match the requested format.”	Only requirement 4 (1 uppercase letter) is not fulfilled. Error message: “Please match the requested format.”
newemail@gmail.com	newuser	TESTING@123	Only requirement 5 (1 lowercase letter) is not fulfilled. Error message: “Please match the requested format.”	Only requirement 5 (1 lowercase letter) is not fulfilled. Error message: “Please match the requested format.”

White Box Testing

Update User Details

Since the methods for updating user details in the controller and view are separate, we'll only do the control flow for the controller class. Users are allowed to modify their name, email and car plate number but not their username since the username is a unique key used to identify users in the database. Error checking is done in the view class and the inputs received are in the correct format.

Control Flow Diagram



Basis Path Testing

Cyclomatic Complexity
= |decisionpoint| + 1 = 4 + 1 = 5 (since all decision points are binary)

Basis Paths

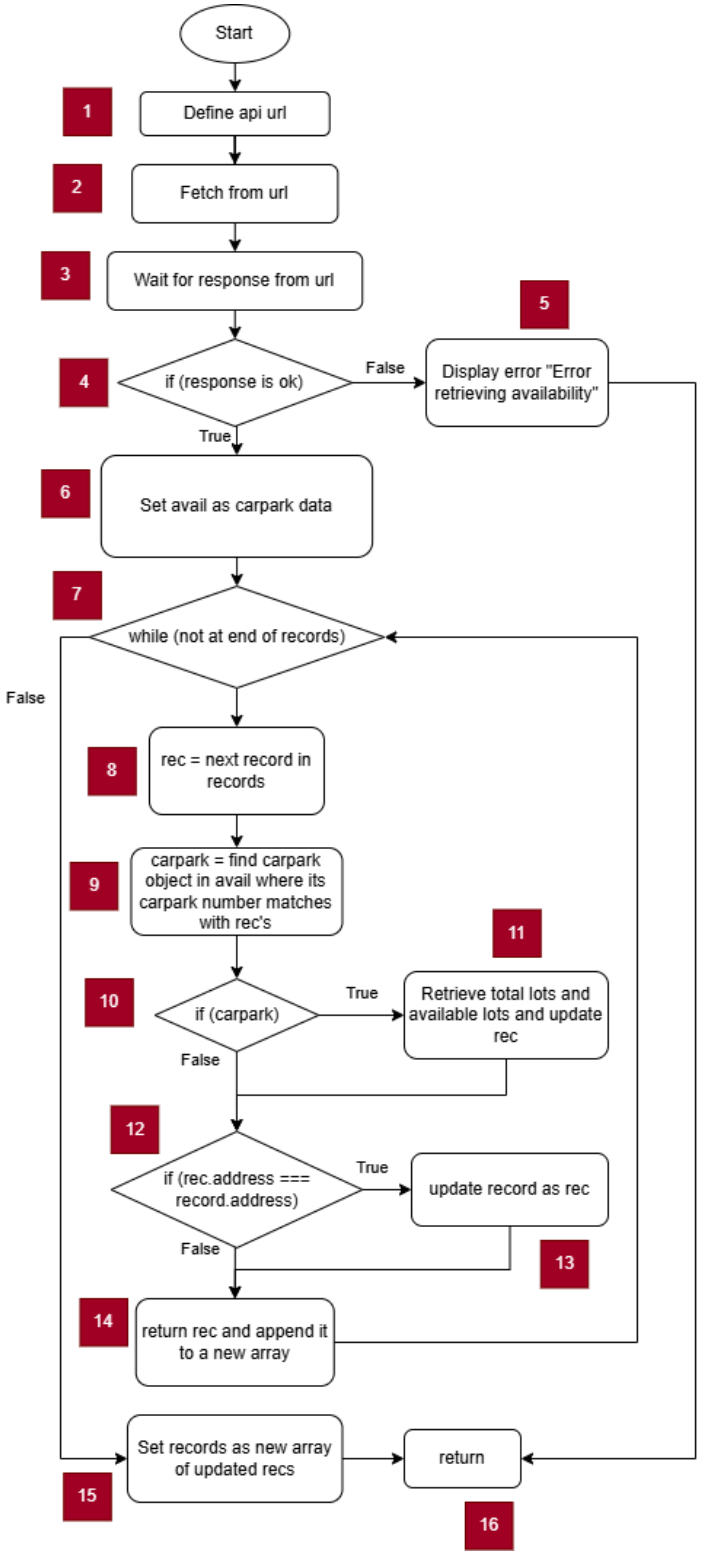
Basis Path no.	Basis Path	Description
1 (Baseline)	1, 2, 3, 4, 6, 10, 11, 12, 14, 15	Normal update flow (no email change, save successful)
2	1, 2, 3, 4, 5, 15	User not found in Database
3	1, 2, 3, 4, 6, 7, 8, 9, 15	New email already taken
4	1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 14, 15	Email change successful
5	1, 2, 3, 4, 6, 10, 11, 12, 13, 15	Save failed due to DB error

Basis path	Test Input	Expected Output	Actual Output
1	Email: sameemail@gmail.com Name: New Name Car Plate No.: ABC-1234	User updated	User updated
2	Email: sameemail@gmail.com Name: New Name Car Plate No.: ABC-1234 Setup: User record deleted from DB before test. Input fields remain the same.	“User not found”.	“User not found”.
3	Email: exisitngemaill@gmail.com Name: New Name Car Plate No.: ABC-1234	“Email already in use”	“Email already in use”
4	Email: newemaill@gmail.com Name: New Name Car Plate No.: ABC-1234	User updated	User updated
5	Email: sameemail@gmail.com Name: New Name Car Plate No.: ABC-1234 Setup: Turn database offline at node 6	“Error updating profile: {error}”	“Error updating profile: {error}”

Fetching Carpark Availability Data

“records” is the array of carpark objects from HDB carpark information api. “record” is the carpark the user is currently viewing. “avail” is the carpark availability data fetched from the external api.

Control Flow Diagram



Basis Path Testing

Cyclomatic Complexity
= |decisionpoint| + 1 = 4 + 1 = 5 (since all decision points are binary)
Basis Paths

Basis Path no.	Basis Path	Description
1 (Baseline)	1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 7, 15, 16	Normal update flow (successful fetch, match found, updates made to records array and current record being accessed)
2	1, 2, 3, 4, 5, 16	Error receiving response from API.
3	1, 2, 3, 4, 6, 7, 15, 16	The records array is empty.
4	1, 2, 3, 4, 6, 7, 8, 9, 10, 12, 13, 14, 7, 15, 16	Carpark not found in availability data.
5	1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16	The address of the record being updated (rec in records) is not the same as the current record being accessed / viewed by user (record).

Basis path	Test Input	Expected Output	Actual Output
1	url = api_url records = [carpark1] record = carpark1	carpark1 in records and record has total lots and available lots information added	carpark1 in records and record has total lots and available lots information added
2	url = “” records = [carpark1] record = carpark1	“Error retrieving availability”	“Error retrieving availability”
3	url = api_url records = [] record = carpark1	“No records to update”	“No records to update”
4	url = custom_url where response = [] records = [carpark1] record = carpark1	carpark1 not updated	carpark1 not updated
5	url = api_url records = [carpark1] record = carpark2	carpark1 in records updated, record (carpark2) not updated	carpark1 in records updated, record (carpark2) not updated

Demo Script

[Slide 1] Hi everyone! Ever found yourself circling endlessly for a parking spot? We've all been there. Good afternoon professor and friends, I'm Jia Yuun, and with me are Brandon, Wen Rong, Aditi Qi Cheng, and Bradley. We're excited to introduce ParkSmart—an app designed to take the stress out of parking.

[Slide 2] Here's a quick overview of today's presentation. We'll cover the problem, our solution, system design, software traceability, and future enhancements.

[Slide 3] Let's begin by looking at the problem we set out to solve.

[Slide 4] In Singapore, parking rates and restrictions are usually only visible at the carpark entrance. This makes it hard for drivers to find suitable lots quickly. Drivers need a way to access carpark details ahead of time for a smoother experience.

[Slide 5] And that brings us to our solution.

[Slide 6] Introducing ParkSmart! An app built for all drivers—whether you're a daily commuter or new driver. It focuses on three key pillars:
Data Security – with encrypted and salted passwords.
Real-Time Data – live carpark info from APIs.
User Connectivity – a community forum for user interaction.

[Slide 7] We use 3 main APIs and a dataset:
HDB Availability API – updates parking space availability every 60 seconds.
OneMap API – live maps and location data, helping users estimate distance and travel time.
HDB Carpark Dataset – a CSV file with static carpark info like type and gantry height.

[Slide 8] Our tech stack keeps ParkSmart fast and reliable:
Frontend: React + TailwindCSS for responsive design.
Backend: Node.js for real-time processing and API management.
Database: MongoDB Atlas for secure, scalable cloud storage of user and carpark data.

I'll now pass the time over to __ for a live demonstration of ParkSmart.

live demo

So how does our app work? Let's take the example of a new user, Joe.

To start, Joe may register as a new user by signing up. Here, new users enter their email, car plate number, username and password. As shown, a user may only proceed if the password fulfills the required conditions. Before completing registration, users must also agree to our user agreement and privacy policy. This ensures transparency and keeps your data safe.

If an email has already been registered, the user is prompted to enter a valid email. After successfully entering the required information, Joe can successfully sign up as a new user.

And there! After logging in, Joe is brought to the ParkSmart homepage. This is where all the main features of the app come together.

On the navigation menu, Joe can change the language based on his preferred language. This feature makes the app more accessible to a diverse user base. Another accessibility feature? Dark mode. This improves accessibility, especially for comfort use in low-light environments, particularly when driving at night.

And of course, from the homepage, we can navigate to the main function of the app, the carpark search page. After Joe allows the browser to access his location, he can see a map. The map is automatically loaded to his current location, shown by a marker. Below the map, we also see cards displaying information about each car parks organised by the ones nearest to him. The cards include carpark location, availability and height limit. When we click View More, we see additional information, including payment. Additionally, red, green and yellow have been used to visually indicate carpark availability for better accessibility.

Now let's say Joe wants to find parking at the Bugis MRT. Joe can simply search for the desired location on the search bar and select the suggested result. And accordingly, the list of carparks will be organised based on the distance to Bugis MRT. Using the slider, we can dictate the number of carparks to be shown.

Additionally, he may filter the carparks based on his requirements, including free parking, night parking, availability and gantry height. And the carparks will be accordingly filtered.

Next, let's check the forum. When Joe navigates to the forum page, he can view posts from users sharing their experiences. He may comment on posts, edit comments and also report posts to be reviewed by the ParkSmart Team. Similarly, he may make his post by sharing his experience and may attach files. Once published, he may edit and delete the post.

Next, let's check the support page. Here he may see FAQs. He can also submit a feedback. Here he may enter his details, choose a subject, write his concern, rate our service and submit the form. This will be sent to be reviewed by the ParkSmart team. Additionally, we have email and phone support. When clicking email support, Joe will be directed to his email, where a draft email is started to send to our team.

Finally, let's look at the Profile Page. Here, Joe can see his personal information, including name, email, car plate number, and last logged in time. Additionally, he may change these details. The changing password has an extra layer of security. Joe must enter his current password, and a new secure password. Let's test out this change by logging out, and logging in with the new password. And as shown, Joe can log in again using the updated password. Finally, he may also delete his account.

Now, to manage all the features, we have a special access Admin account. First, they can edit the about page. Next, the admin account support page is redirected to an admin page. Here, the admin may see recent feedback, the number of feedback and the average rating. Next, they may see user reports. Here we see the one submitted by Joe. Admins may click the view post to review and delete the necessary posts or comments. Admins can also directly access posts by going to the forum page.

And with that, we present ParkSmart. With ParkSmart, we've created an app that not only helps users find parking easily but also builds a community, values user feedback, and ensures accessibility for all.

I will now pass the time over to __ to talk more about the System design of ParkSmart.

[Slide 10] Now let's take a look at the System Design.

(next slide)
Now we will go through our design patterns

DESIGN PATTERN

System architecture
[Slide 11] Here is our system architecture. The ParkSmart system follows a modular 4-layer architecture that separates concerns across distinct layers to ensure scalability and maintainability.
The **Presentation Layer** handles user interactions through pages and components. It captures user input and sends the relevant requests to the business logic layer.
The **Business Logic Layer** – the core operations of the systems. It processes requests, applies business rules, and communicates with the data layer to fetch or modify data.
The **Data Access Layer** abstracts the way how business logic interacts with various data sources, It ensures that the business layer is decoupled from data management details and performs CRUD operations.
Finally, the **Database Layer** handles data storage. Connecting to MongoDB for persistent storage, and managing file storage systems for larger data like media files.
This modular architecture enforces clear boundaries between different layers, and ensures that the system is both testable and easily extensible for future features or improvements.

[Slide 12] Let's look at the further design patterns we've used in our system.

MVC Pattern The MVC pattern structures the system into three core parts: the Model (MongoDB schemas via Mongoose) handles data storage and retrieval **(Click)**, the View (React frontend) manages user interaction and display **(Click)**, and the Controller (Express route handlers) processes user requests and coordinates responses **(Click)**. This separation ensures cleaner, maintainable code and a more scalable application architecture.

(next slide)

Repository Pattern The **Repository Pattern** helps us manage how we access and work with data in a consistent way. These repositories provide a simple interface to access data. For example **(Click)**, our **CarParkRepository** manages accessing car park data fetfrom different sources like external APIs. If we ever need to change how we get the data, we only need to update the repository, and everything else in the system will continue to work smoothly. This makes our system more flexible and easier to maintain.

(next slide)

Strategy Pattern The **Strategy Pattern** is a way of making your system more flexible by allowing it to switch between different methods or strategies. For example, in our system, we used this pattern for the "Change Language" feature. It allows the system to easily switch between languages like English, Chinese, Bahasa-Melayu, and Hindi without modifying the core logic.

GOOD PRACTICES

SOLID Principles

- **Single Responsibility**
- **Open/Closed**
- **Liskov Substitution**
- **Interface Segregation**
- **Dependency Inversion**

Agile Methodology We used two-week sprints with regular stand-ups and sprint reviews, adapting requirements based on feedback.

(next slide)

We also implemented specific features to fulfil our non-functional requirements. We highlight our key qualities that determine how our system performs. These include reliability, security, usability, maintainability, and robust error handling. For instance, our JWT token generation follows a consistent structure, improving both maintainability and security by standardizing how user sessions are managed.

(next slide)

To enhance usability and personalization, we implemented a light and dark mode toggle. (Click) and also the ability to toggle between various languages. (Click) Our frontend is responsive on all viewports, be it the phone or computer.

Next, to ensure maintainability, we have implemented reusable components for frequently used functionalities and UI elements, such as our header and footer, that can be repeatedly called in different front-end pages. This ensures that future updates are easily implemented and consistency across different pages are present. (Click)

Our UseEffect hook regularly polls for updates every 60s, ensuring data is kept current. (Click) while passwords are encrypted via bcrypt. (Click)

We've also added try-catch blocks throughout our code to handle errors, like during password changes, where the try block handles logic and the catch sends an error response to the client.

So how can we determine how good the traceability is? We will be illustrating it with our user Sign Up and Profilepage use case.

Bradley script

Slide 29: Traceability (Slide Title)

Now we'll walk through our software traceability process - we've ensured that our implementation directly ties back to our requirements.
As you can see in this diagram, we've followed a systematic approach moving from functional requirements through requirements analysis, implementation, and finally testing.
Each phase builds upon the previous one, creating clear connections between user needs and our actual code.

Slide 30: Functional Requirement 1 - SignUpController

Our first key functional requirement is the user sign-up process.
This use case description outlines the flow for new users creating accounts.
The flow follows a logical sequence from opening the app to successfully creating an account, also carefully documenting alternative flows for situations like email validation failures or username conflicts.

Slide 31: Functional Requirement 1 - Use Case Diagram

This diagram visualizes how our sign-up functionality integrates with other components of the system.
You can see the SignUp highlighted here as one of several user interactions.
The diagram shows how our system connects various user actions with external APIs, including OneMap and HDB carpark information APIs.

Slide 32: Requirements Analysis - Class Diagram

Moving to the requirements analysis phase, this class diagram maps the relationships between our system objects. These show the boundary, control, entity classes related to the sign up use case. Maintaining clean separation between functions, ensuring each class has appropriate relationships with others in the system.

Slide 33-36: Requirements Analysis - Sequence Diagrams for SignUpController

These sequence diagrams detail the exact flow of the sign-up process. You can see the interactions between the user, SignUpPage, AuthWrapper, Server, and MongoDB. The main flow shows the validation process, checking for existing accounts, and successful registration with JWT token generation.

Slides 34-36 highlight important validation elements in our implementation. Notice how we're displaying validation errors for issues like "Email already in use" and password format requirements. These error handling flows are critical for providing users with clear guidance during the registration process.

Slide 37: Implementation - Design Principles

In our implementation phase, we've applied several design principles. This slide demonstrates the Single Responsibility Principle as applied to our SignUpController. Each component of our code has clear validation, data processing, persistence, authentication setup, and response handling.
By separating these concerns, we've created more maintainable and testable code.

Slide 38: Implementation - Single Responsibility Principle

Here you can see how we've applied the Single Responsibility Principle in practice. The SignUpController class is responsible only for the logic of signup, while the SignUpPage is responsible for UI elements. This separation ensures that changes to one aspect won't affect the other, making our code more robust and easier to maintain.

Slide 39: Testing - Black Box Testing

For testing our signup functionality, we've implemented black box testing to ensure all flows work correctly. Our test cases cover the main flow for successful registration as well as alternate flows for validation failures. We've tested email format validation, duplicate email scenarios, and username conflicts to ensure proper error messages are displayed to users, and invalid passwords.

Jiayuun script

Slide 40: Functional Requirement 2 - ProfileController

Our second key functional requirement focuses on the profile management functionality. This use case description outlines how users can view and modify their profile information. The precondition is that users must be logged in, and the flow includes selecting the Profile option, viewing current information, making changes, and saving those updates successfully.

Slide 41: Functional Requirement 2 - Use Case Diagram

In this use case diagram, we've highlighted the profile-related functions, including "View profile page" and "Edit Profile Page." These are part of the broader system functionality that gives users control over their account information.

Slide 42-46: Requirements Analysis and Sequence Diagrams for ProfileController

These slides walk through the detailed analysis of our profile management functions. The class diagram shows ProfileController's relationships, while the sequence diagrams illustrate the detailed flow of editing profile information.

Important aspects include enabling edit mode, validating inputs, saving changes, and handling both success and cancellation flows. You can see how we've implemented JWT token verification before allowing profile updates, which is a critical security measure.

Slide 47: Implementation - Design Principles (ProfileController)

For the ProfileController implementation, we've applied the Interface Segregation Principle. Each handler method has a clean, specific purpose - handleDeleteAccount, handleEditCancel, handleSave, etc. This approach makes our code more modular and easier to maintain as we can modify individual handlers without affecting others.

Slide 48: Testing - Basis Path Testing

Finally, we've conducted basis path testing for our ProfileController functionality. The control flow diagram shows various decision points, with each path representing different scenarios like successful updates, validation failures, and database errors. Our testing table documents the input conditions, expected outputs, and actual results for each path, ensuring our implementation performs correctly across all possible user scenarios.

Now, we will finally conclude our presentation with some proposed further upgrades, including introducing AI prediction models for peak period car park availabilities and a Filter for handicap parking spaces. We will also Allow users to book car park spaces & pay on the go. Users will have the option to save their preferred car parks or filters and their passwords.