Demo Script

[Slide 1] Hi everyone! Ever found yourself circling endlessly for a parking spot? I'm sure all who drive understand the feeling. 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 our 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.

If an email and username has already been registered, the user is prompted to enter a valid email. If an email isn't in the correct format. Before completing registration, users must also agree to our user agreement and privacy policy. This ensures transparency

and keeps your data safe. 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, in low-light environments, 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 his 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. **His login remains active for up to an hour.** Show new link keep remain login
Sign out and open the link again and show is sign out

Finally, he may also delete his account.

Note that our routes are protected—users that are not logged in cannot access profile page, post or comment in forums.

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.

if a user types in a random address, it will always redirect to the homepage.
Finally, ParkSmart is fully responsive across all viewports — and it looks great on any screen size.

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)

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: **(Click)** 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 . 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 from 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.

(click)

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.

Now, I’ll pass the time to Wen Rong.

(next slide)

Wen Rong’s Script

We implemented features to fulfill our non-functional requirements—reliability, security, usability, maintainability, and error handling.
Our JWT token generation follows a consistent structure, enhancing both security and maintainability. The token includes essential user information, allowing users to remain

logged in across sessions. It is signed using a secret key, which ensures the integrity of the token and prevents tampering or forgery.

(next slide)

To improve usability, we added a light/dark mode toggle (click) and language selection (click). Our frontend is fully responsive across all devices and adjusts accordingly to the screen dimensions. (click)

For maintainability, we use reusable components like headers and footers, allowing easier updates and design consistency. (click)

We use a useEffect hook to poll updates every 60 seconds (click), and passwords are securely hashed using bcrypt in order to maintain security, resulting in an irreversible string stored in the database. (click)

We also implemented try-catch blocks across the backend—for example, during password changes, where errors are caught and handled with clear feedback.

(click) To measure how well our features link back to requirements, we'll now look at the Sign Up and Profile Page use case traceability.

So, how can we assess the quality of traceability? We will demonstrate this through our User Sign Up and Profile Page use cases. (click)

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. (click)
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

Slide 33: Overall sequence diagram for Sign Up

Here you can see the overall sequence diagram for the sign up use case

Slide 34: Creating Account

User opens app and is at the Login page, then clicks the signup form and enters theirs details (name, email, userID, password, car plate)

Slide 35: Username exists and password error Alt Flow

Here you can see alternate flows if a username already exists this message will pop up. If the password strength requirement is not met, there will be a “X”.

Slide 36: Email format or already exists

Here you can see the alternate flow if an email already exists or if the email format is wrong. The following messages will be displayed.

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 SignUp use case. 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 - Class Diagram
Moving to the requirements analysis phase, this class diagram maps the relationships between our system objects. This shows the flow between classes involved in the update profile use case.

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

This is an overview of our sequence diagram. (click)

We start with the ProfileController's role—enabling users to edit their details. This is the UI that will be displayed for the edit profile details page, with details like the name, email address, car plate number and username The user can edit whichever field they want before clicking on save changes.

(click)

After clicking on the save changes, call updateprofile method will run, followed by the update of the user profile as json-ified data will be sent from our front end to our Mongodb backend.

(click)

Finally, the app will confirm that changes have been successfully updated. (click)

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. As there are 4 decision points, the cyclomatic complexity has a value of 5. For basis path testing, the first and fourth test cases has a path that reaches a successful updates,

while test case 2, 3 and 5, which test invalid inputs, have paths that reach an unsuccessful updates. As shown, all of our test cases have passed. 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.