PARK IT

parkit.website

Presented by:
SCSF-J4

Rashi Ojha (U2323323D)
Sheth Shaivi Sachin (U2323665B)
Dylan Quek Zhi En (U2321239L)
Lee Zhuo Yang, Nicholas (U2423112F)
Richard Chong Wing Liong (U2222047G)

PROBLEM STATEMENT

Currently, there are no integrated parking apps which enable Singaporeans to

1. View available parking lots
2. Navigate to the selected carpark
3. Track the costs of each parking session

Existing solutions address problems separately, lacking integration.

Drivers must switch between apps for parking costs, availability, and directions.

Navigation

Waze

Google Maps

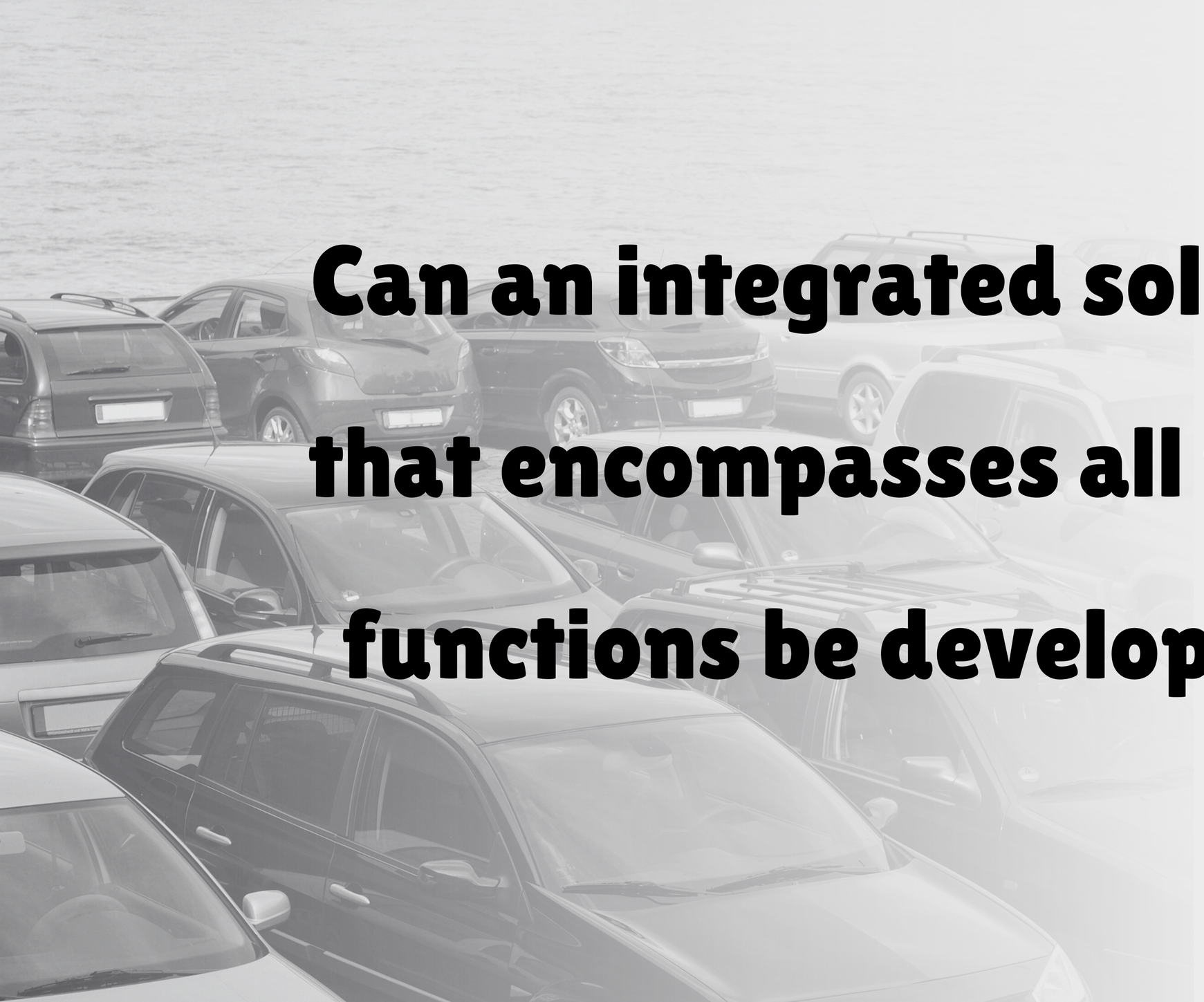
Parking Availability

Carpark Rates

Track parking costs

Parking.sg

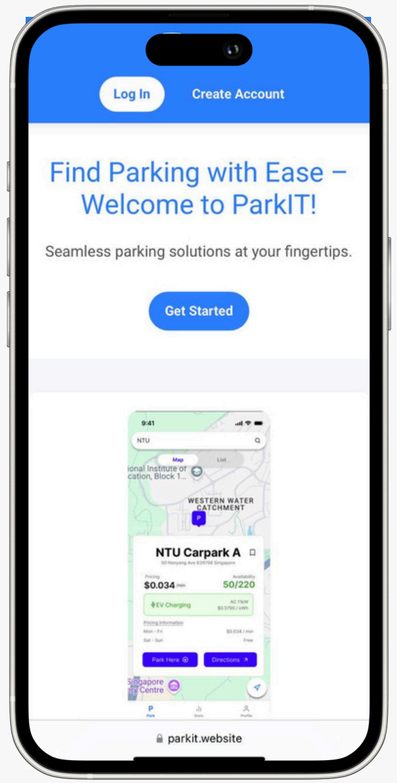
Parkopedia

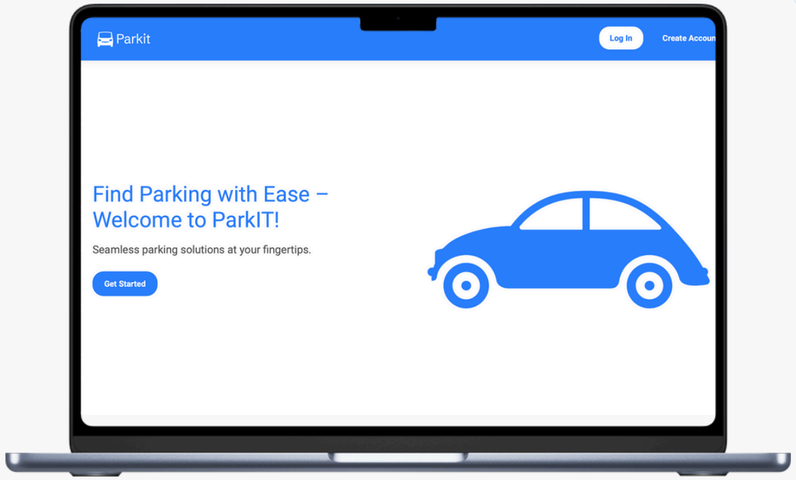


Can an integrated solution that encompasses all these functions be developed?

Our Solution: ParkIT

- ParkIT is a user-friendly web app using real-time government APIs for efficient parking.
- Users can search locations to view nearby car parks with live availability and pricing.
- It personalizes the experience by storing parking session data in a database.





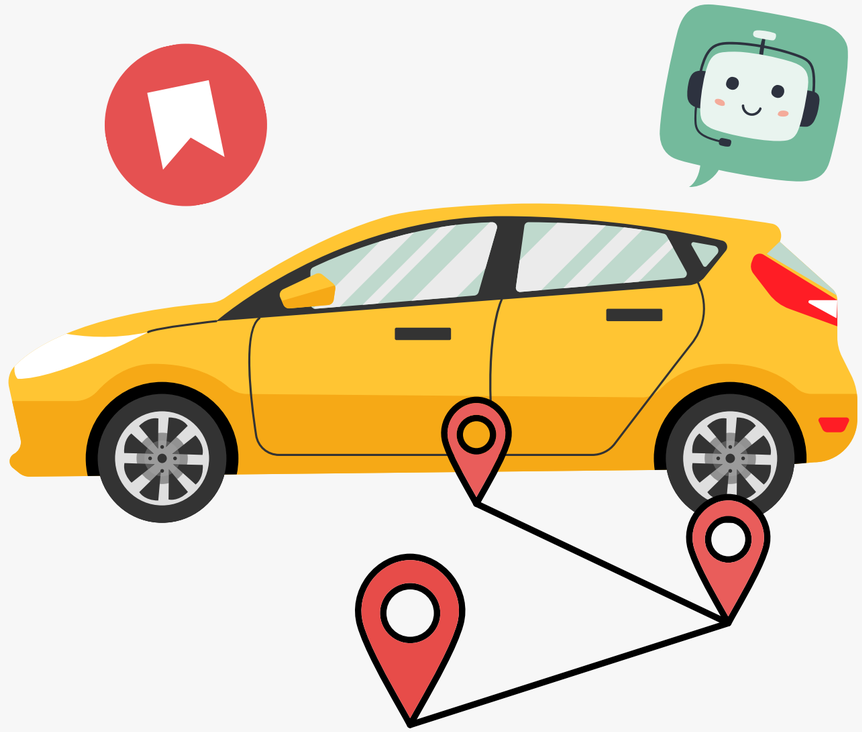
TECH STACK

Front End	Back End
React	Java
Vite	MySQL
Tailwind CSS (for styling)	Spring Boot
Leaflet	Gradle

URA API OneMap API Gemini API

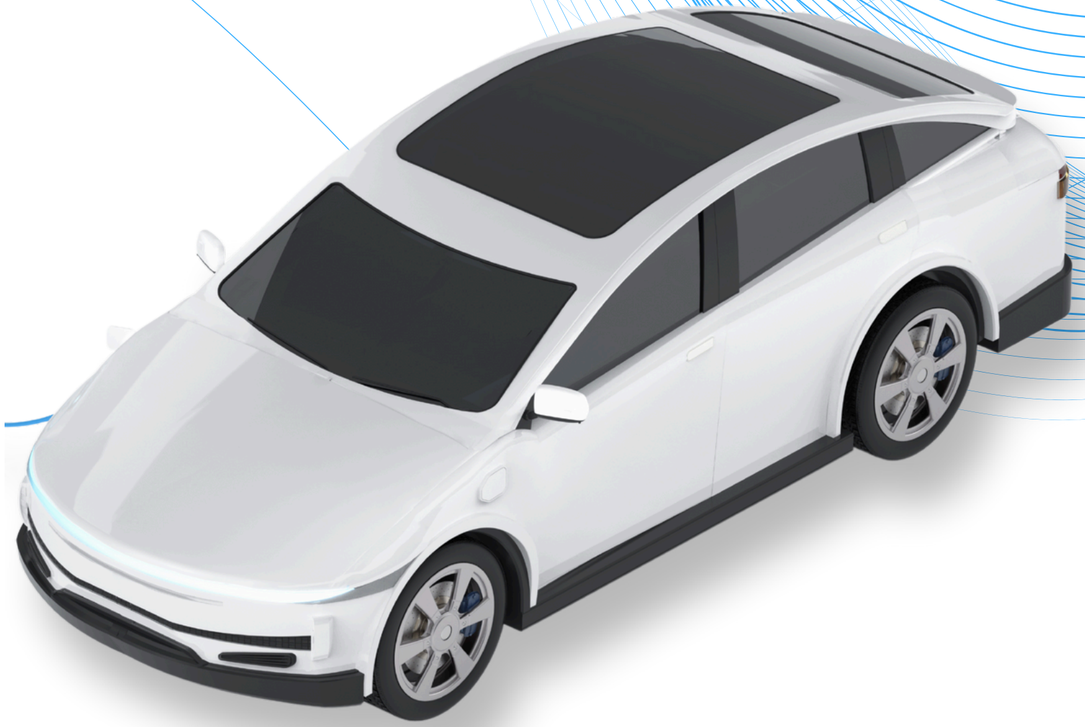
FEATURE LIST

1. AUTHENTICATION
2. SEARCH DESTINATION
3. APPLY FILTER
4. NAVIGATE TO YOUR PARKING
5. RECORD PARKING
6. SAVE FAVOURITES
7. VIEW DASHBOARD
8. EDIT PROFILE
9. QUERY CHATBOT

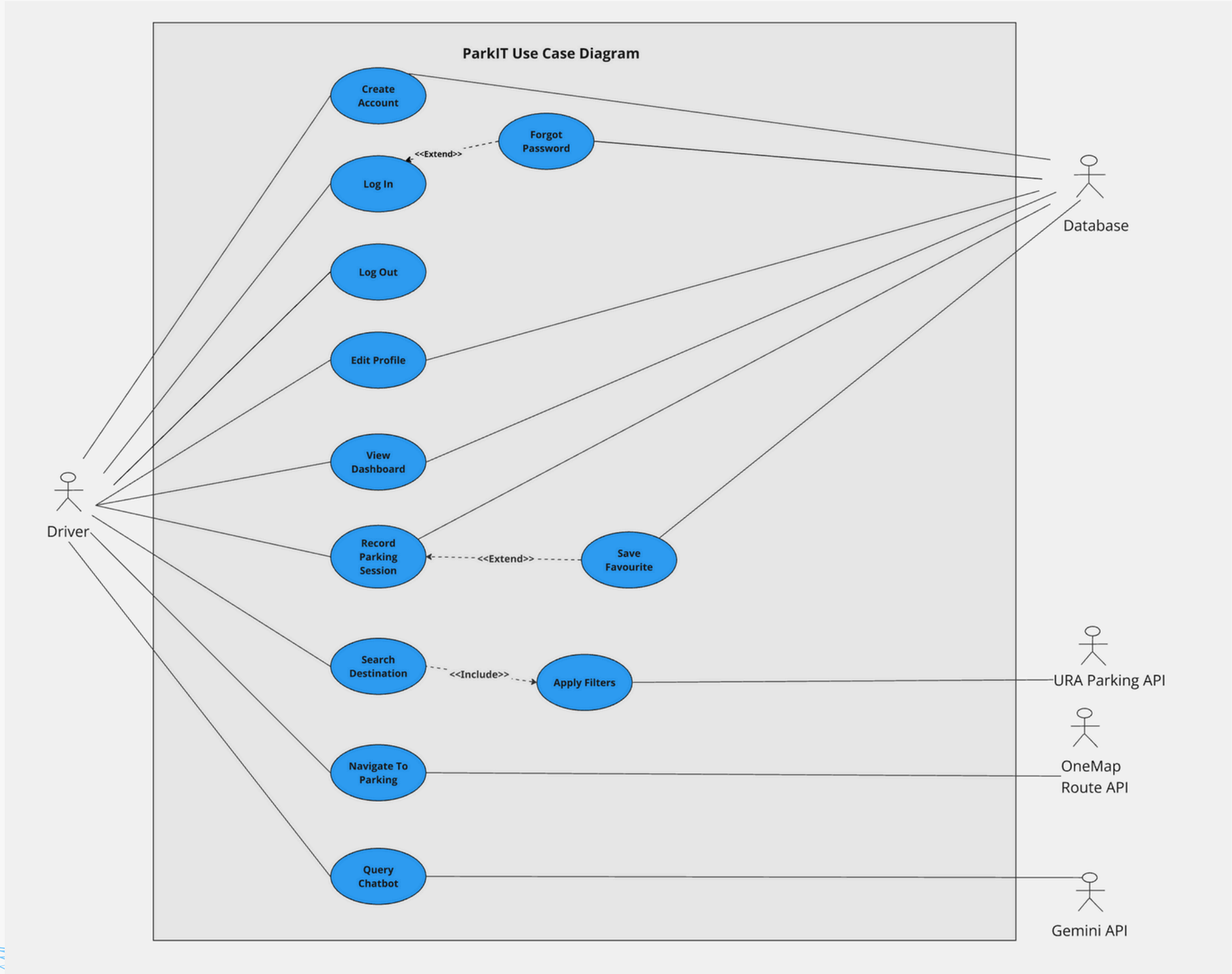


LIVE DEMO

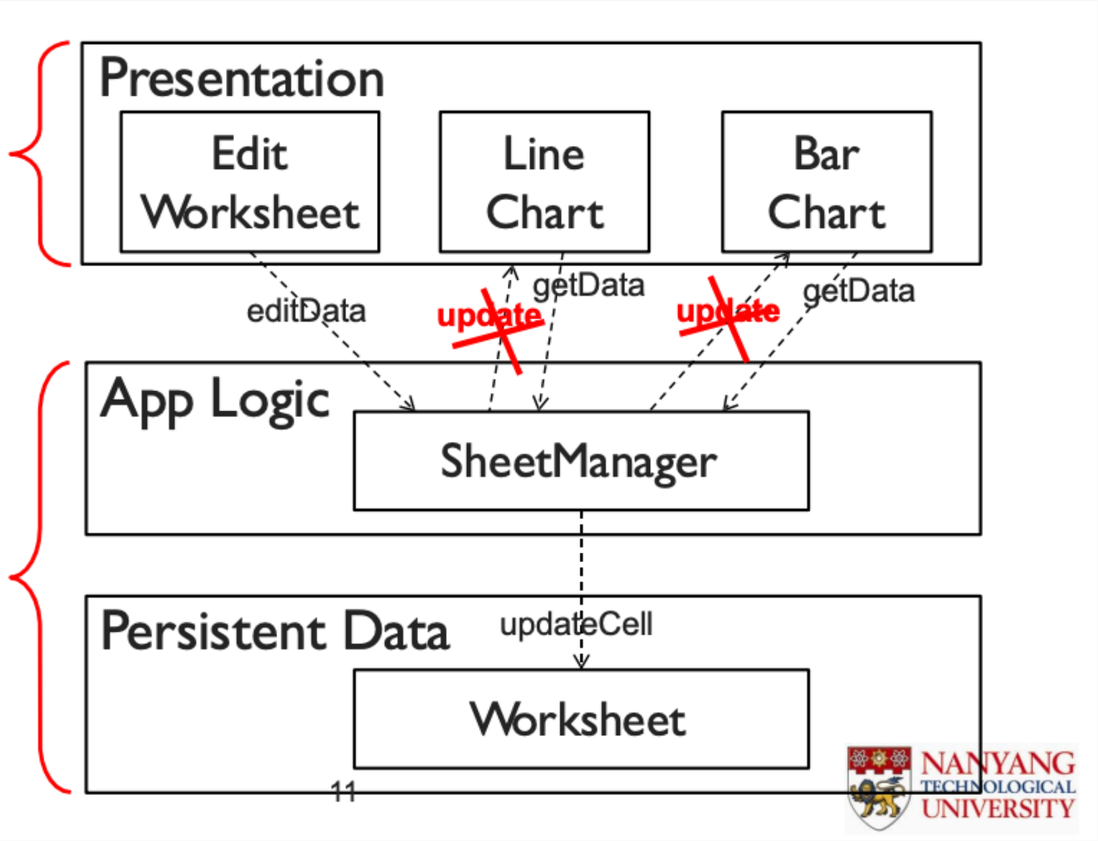
parkit.website



USE CASE DIAGRAM



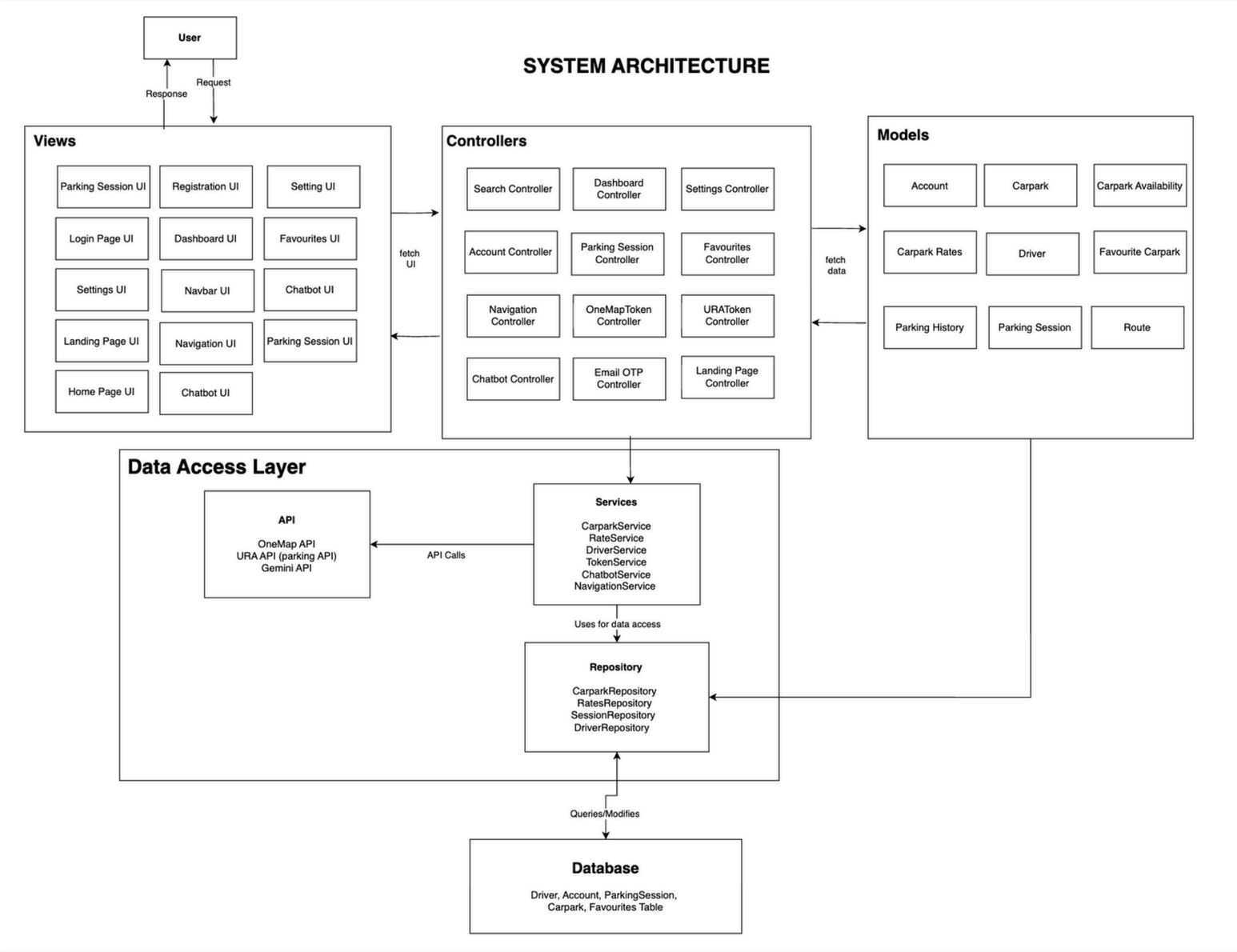
WHY MVC ARCHITECTURE?



- The same data can be presented differently
- Changes to UI must be easy and even possible at runtime
- The application display and behavior must reflect data changes immediately
- The same presentation can have different look and feel
- The application reacts to user input differently

WHAT PATTERN CAN BE USED HERE?

SYSTEM ARCHITECTURE



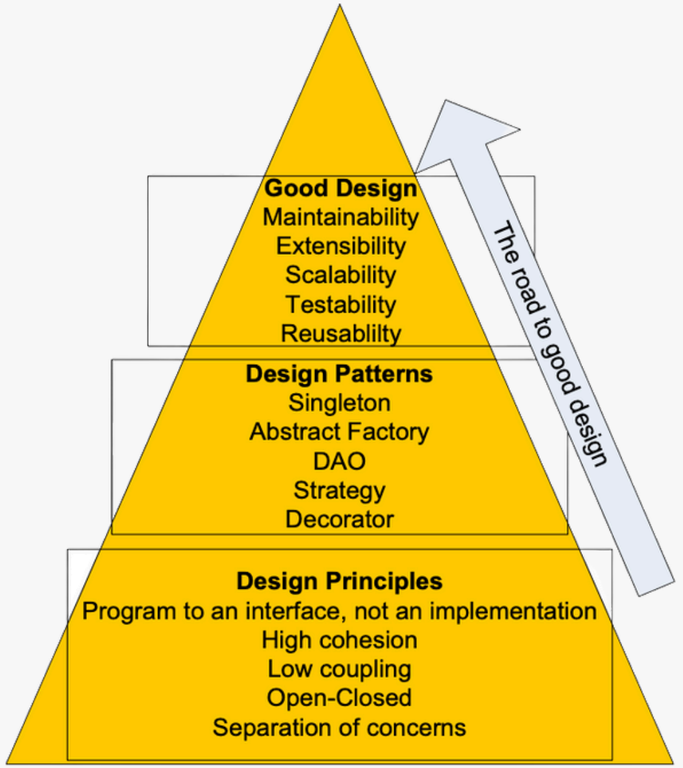
MVC ARCHITECTURE

Advantages :

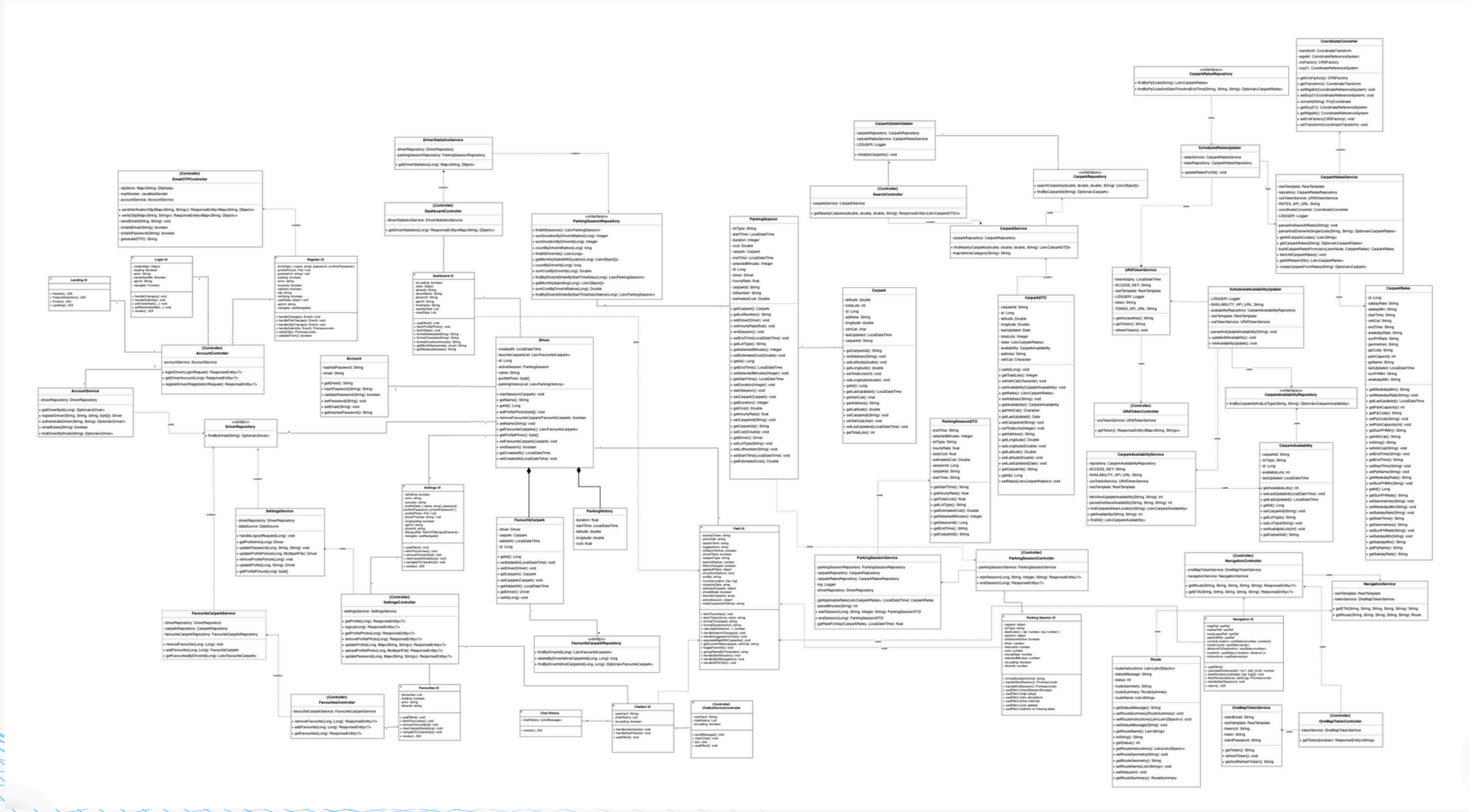
- Simultaneous Development:
- Multiple Views for a Model:
- Uses Design Patterns :
  - Model uses the Observer Pattern to keep the View(s) updated.
  - View and Controller use the Strategy Pattern

Disadvantages

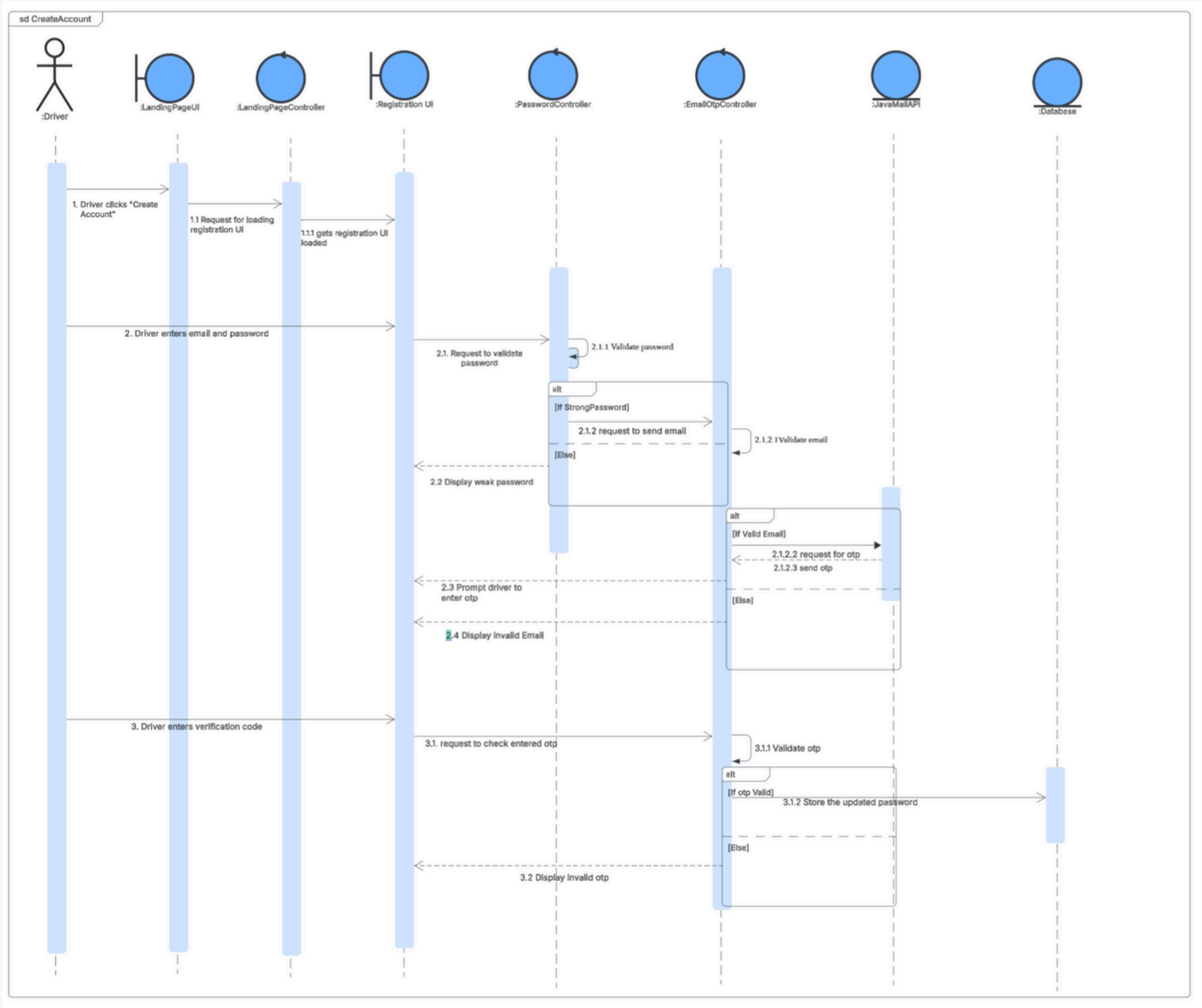
- Code Navigability
- Learning Curve:



CLASS DIAGRAM

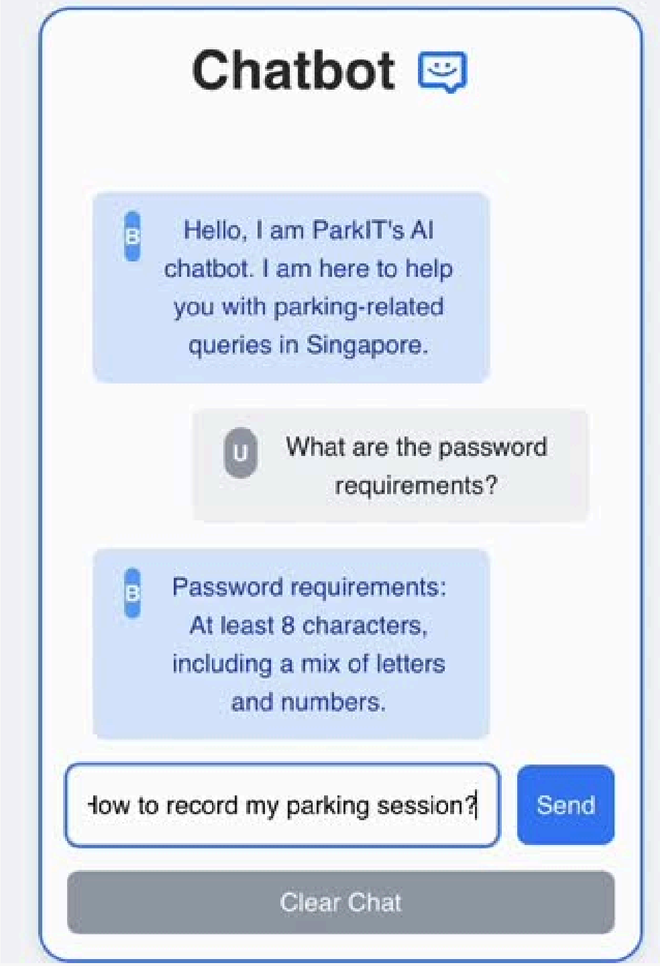


CREATE ACCOUNT SEQUENCE DIAGRAM



CHATBOT - GEMINI FLASH 1.5

Approach	Brief Description	Pros	Cons
RAG	Combines a retriever to fetch relevant context and a generator (Gemini API) to answer based on that context.	-No need to retrain models - Works well with long documents	- More complex system architecture - Performance depends on retrieval quality
Fine-Tuning + Prompt Engineering	Customizes a base model via additional training (fine-tuning) and tailors output using specific prompt patterns.	- Faster inference (no retrieval step) - Fine-tuned on specific domains	-Hard to keep up-to-date - Risk of overfitting



GOOD SWE PRACTICES

1. DOCUMENTATION
2. GOOD CODE PRACTICES
3. REUSABILITY AND REFACTORING
4. SCRUM
5. CLEAN ARCHITECTURE

DOCUMENTATION - README

Getting Started

Prerequisites

- **Java 21** (JDK)
- **Node.js** (includes npm)
- **MySQL version 5.7.24**

Setting up a Local MySQL Database

1. **Open Terminal**

   On Windows, **avoid using PowerShell or CMD**. Use

2. **Downloading MySQL**

   For macOS users,

   ```
   brew install mysql
   ```

   To initiate MySQL,

   ```
   brew services start mysql
   ```

   Set a root password

   ```
   mysql_secure_installation
   ```

3. **Creating the Database**

   To connect to MySQL,

   ```
   mysql -u root -p
   ```

   Create the ParkIT DB

   ```
   CREATE DATABASE parkit_db;
   ```

Running frontend + backend servers

1. **Navigate to Project Root**

   ```
   cd ParkIT
   ```

2. **Ensure .env file is in the /ParkIT directory**

   ```
   URA_API_KEY=???
   ONEMAP_API_EMAIL=???
   ONEMAP_API_PASSWORD=???
   ```

3. **Run startup script**

   ```
   ./start.sh
   ```

   The shell script populates the database with carpark data, the DB tables have been initialised, the URA API will be call

4. **Open Browser** http://localhost:5173/

Debugging

- **Backend (view logs)**
  - `cd backend`
  - `./gradlew bootRun`
- **Frontend (view logs)**
  - `cd frontend`
  - `npm run dev`
- **Manual Database Querying**

  After connecting to MySQL, to use the Database,

  ```
  USE parkit_db;
  ```

  To show Database tables,

  ```
  SHOW TABLES;
  ```

  To query Database tables,

  ```
  SELECT * FROM TABLE_NAME;
  ```

ParkIT Application Skeleton

Frontend (React)

File	Description
`src/pages/main.jsx`	The main entry point for the React application, renders the root App component.
`src/pages/App.jsx`	Defines the core application structure, routing, and protected routes.
`src/pages/Landing.jsx`	Component for the public landing page.
`src/pages/Login.jsx`	Component for the user login page and logic.
`src/pages/Register.jsx`	Component for the user registration page and logic, including OTP.
`src/pages/Park.jsx`	Main component displaying the map and list view for finding carparks.
`src/pages/ParkingSession.jsx`	Component managing an active or completed parking session view.
`src/pages/Navigation.jsx`	Component handling map-based navigation/routing to a selected carpark.
`src/pages/Dashboard.jsx`	Component displaying driver statistics, charts, and recent parking history.
`src/pages/Favourites.jsx`	Component for viewing and managing the user's favorite carparks.
`src/pages/Settings.jsx`	Component for managing user profile information, password, and photo.
`src/pages/Chatbot.jsx`	Component providing the AI chatbot interface.
`src/pages/ChatHistory.jsx`	Reusable component to display the list of chat messages.
`src/pages/Navbar.jsx`	Reusable component for the bottom navigation bar in the authenticated app section.
`src/services/chatbotService.js`	Service providing function to interact with the chatbot (Gemini) API.

Backend (Java - Spring Boot)

File	Description
Application Entry Point	
`Application.java`	Main Spring Boot application class; the entry point for the backend server.
Controllers (API Endpoints)	
`controller/AccountController.java`	Handles API requests for user registration and login.
`controller/DashboardController.java`	Handles API requests for retrieving driver statistics for the dashboard.
`controller/EmailOTPController.java`	Handles API requests related to sending and verifying email OTPs for registration and password reset.
`controller/FavouritesController.java`	Handles API requests for adding, removing, and retrieving favorite carparks for a driver.
`controller/NavigationController.java`	Handles API requests for getting route information (geometry, instructions, ETA) from OneMap.
`controller/OneMapTokenController.java`	Handles API requests for retrieving the OneMap API access token.
`controller/ParkingSessionController.java`	Handles API requests for starting and ending parking sessions.
`controller/SearchController.java`	Handles API requests for searching nearby carparks based on location and filters.
`controller/SettingsController.java`	Handles API requests related to user settings (profile update, password change, photo upload/delete, logout).
`controller/URATokenController.java`	Handles API requests for retrieving the URA Data Service API token.
Data Transfer Objects (DTOs)	
`model/CarparkDTO.java`	DTO combining data from Carpark, CarparkAvailability, and CarparkRates for search results.
`dto/DriverResponse.java`	DTO for structuring driver data in API responses, includes static factory method `fromDriver`.

Detailed instructions to run the program for first-time users and future developers, enabling long-term consistency and good development

DOCUMENTATION - CODE COMMENTS

```java
public Driver updateProfile(Long driverId, String name) {

/**
 * Update user's password
 */
@Transactional
public void updatePassword(Long driverId, String currentPassword, String newPassword) {
    Driver driver = driverRepository.findById(driverId)
            .orElseThrow(() -> new EntityNotFoundException("Driver not found with id: " + driverId));

    // Validate the current password
    if (!driver.validatePassword(currentPassword)) {
        throw new IllegalArgumentException("Current password is incorrect");
    }

    // Validate the new password strength
    if (newPassword == null || newPassword.length() < 8) {
        throw new IllegalArgumentException("Password must be at least 8 characters long");
    }

    // Check if the password contains both letters and numbers
    boolean hasLetter = false;
    boolean hasDigit = false;

    for (char c : newPassword.toCharArray()) {
        if (Character.isLetter(c)) {
            hasLetter = true;
        } else if (Character.isDigit(c)) {
            hasDigit = true;
        }
    }

    // Only check for letters and numbers
    if (!hasLetter || !hasDigit) {
        throw new IllegalArgumentException("Password must contain both letters and numbers");
    }

    // Update the password
    driver.setPassword(newPassword);
    driverRepository.save(driver);
}
```

```javascript
// Effect 4: Timer Interval Logic - Start/Stop based on isSessionActive
useEffect(() => {
    // Clear any previous interval before setting a new one
    if (intervalId) {
        clearInterval(intervalId);
        setIntervalId(null);
    }

    if (isSessionActive && session && !session.endTime) {
        console.log("Timer Effect: Session active, starting interval.");
        const id = setInterval(() => {
            // Use functional update to ensure we use the latest timer value
            setTimer(prevTimer => prevTimer + 1);
        }, 60000); // 1 minute
        setIntervalId(id);

        // Cleanup function for this specific interval
        return () => {
             console.log("Timer Effect: Cleaning up interval ID:", id);
             clearInterval(id);
             setIntervalId(null); // Also clear state on cleanup
        };
    } else {
         console.log("Timer Effect: Session inactive or ended. No interval running.");
    }
// eslint-disable-next-line react-hooks/exhaustive-deps
}, [isSessionActive, session]); // Rerun when active status or session object changes

// Effect 5: Update Cost - Depends on timer, rate, and active status
useEffect(() => {
    if (isSessionActive && hourlyRate > 0) {
        // Calculate cost based on float timer value for accuracy before rounding for display
        const newCost = (hourlyRate * timer) / 60;
        setCost(newCost);
    } else if (!isSessionActive && session?.cost !== undefined) {
         // If session ended, use the final cost from session object
         setCost(session.cost);
    }
    // If inactive and no final cost, cost remains 0 or its previous value
}, [timer, hourlyRate, isSessionActive, session]); // Rerun when these change

// Effect 6: Handle missing initial data AFTER loading check is complete
useEffect(() => {
    if (!isLoading) {
        // If NOT restoring an active session AND essential data for starting a new one is missing
        if (!isSessionActive && (!carpark || !lotType || !destination)) {
```

Makes code easier to understand for enhanced collaboration

CONSISTENT CODING & NAMING CONVENTIONS

1. camelCase for variables and methods across frontend and backend
2. PascalCase for component and class names
3. Proper Indentation

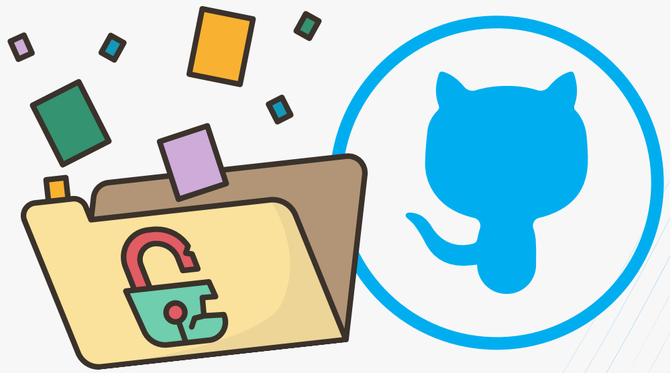
```
public Optional<Driver> authenticateDriver(String email, String password) {
```

```
public class ParkingSessionService {
```

```
public class CarparkAvailabilityService {

    @Value("${ura.api.key}")
    private String ACCESS_KEY;
    private final String AVAILABILITY_API_URL = "https://eservice.ura.gov.sg/uraDataService/invokeUraDS/v1?service=Car_Park_Availability";
    private final CarparkAvailabilityRepository repository;
    private final RestTemplate restTemplate = new RestTemplate();
    private final URATokenService uraTokenService;

    public CarparkAvailabilityService(CarparkAvailabilityRepository repository, URATokenService uraTokenService) {
        this.repository = repository;
        this.uraTokenService = uraTokenService;
    }

    /**
     * Returns the available lots for the given carpark and lot type.
     * Uses cached data if updated within the last 4 minutes.
     */
    public int getAvailability(String carparkId, String lotType) {
        Optional<CarparkAvailability> record = repository.findByCarparkIdAndLotType(carparkId, lotType);
        if (record.isPresent() && Duration.between(record.get().getLastUpdated(), LocalDateTime.now()).toMinutes() < 4) {
            return record.get().getAvailableLots();
        }
        return fetchAndUpdateAvailability(carparkId, lotType);
    }

    private int fetchAndUpdateAvailability(String carparkId, String lotType) {
        try {
            HttpHeaders headers = new HttpHeaders();
            headers.set("AccessKey", ACCESS_KEY);
            headers.set("Token", uraTokenService.getToken());
            HttpEntity<String> entity = new HttpEntity<>(headers);
            ResponseEntity<String> response = restTemplate.exchange(
                    AVAILABILITY_API_URL, HttpMethod.GET, entity, String.class);
            if (response.getBody() != null) {
                return parseAndSaveAvailability(response.getBody(), carparkId, lotType);
            }
        } catch (Exception e) {
            System.err.println("Error fetching Availability API: " + e.getMessage());
        }
        return 0;
    }
```

GITHUB PRACTICES

1. Use of .env file for API credentials. This prevents accidental credential exposure in commits
2. Each team member worked on a separate Git branch to divide development tasks. This allowed parallel progress, avoiding code conflicts and improving productivity



REDUCED DUPLICATION AND REDUNDANCY

- Avoided repeated API calls by checking sessionStorage and caching destination/driver data.
- Used React hooks (useEffect, useMemo) to avoid unnecessary re-renders.
- In the backend, data validation and parsing logic were consolidated to prevent repetition.

REUSABILITY & REFACTORING

1. Utility functions like formatDuration were extracted to avoid duplication.
2. Backend services like CarparkService, SettingsService, AccountService are modularized and reused across controllers.
3. Components like SessionCard, TimerDisplay, and input fields are reused in the frontend.

```
// Format duration in minutes to "Xh Ym" string
const formatDuration = useCallback((mins) => {
    if (isNaN(mins) || mins < 0) return '0h 0min';
    const h = Math.floor(mins / 60);
    const m = Math.round(mins % 60); // Round minutes
    // Handle edge case where rounding makes mins 60
    if (m === 60) {
      return `${h + 1}h 0min`;
    }
    return `${h}h ${m}min`;
}, []);
```

SCRUM


SCRUM MASTER
PRODUCT OWNER
USER STORIES
24
HOURS
REVIEW
IMPLEMENTATION
RETROSPECT
2-4 WEEKS
SPRINT BACKLOG
PLANNING
DEFINITION OF DONE

CLEAN ARCHITECTURE

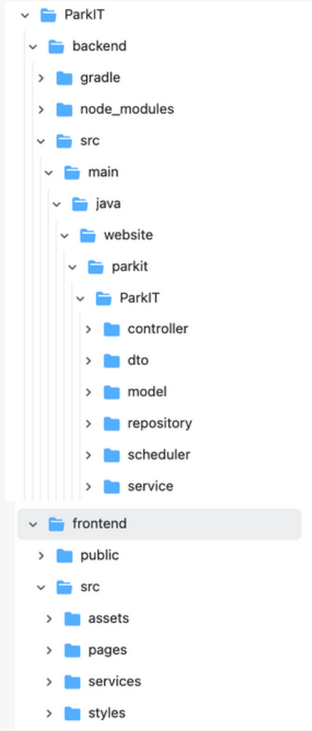
1 MVC Structure

a. Backend

Model → **DTO** → **Service** → **Controller**

b. Frontend

View



2 Single Responsibility Principle

```java
public class CarparkRatesService {
    public void fetchAllCarparkRates() {
        try {
            LOGGER.info("Fetching ALL CarparkRates from URA DataMall...");
            HttpHeaders headers = new HttpHeaders();
            headers.set("AccessKey", uraTokenService.getAccessKey());
            headers.set("Token", uraTokenService.getToken());

            HttpEntity<String> entity = new HttpEntity<>(headers);
            ResponseEntity<String> response = restTemplate.exchange(
                    RATES_API_URL, HttpMethod.GET, entity, String.class);

            String responseBody = response.getBody();
            LOGGER.info("URA API Response: " + responseBody);
            if (responseBody != null) {
                parseAndSaveAllRates(responseBody);
            } else {
                LOGGER.warning("Empty response body from URA for fetchAllCarparkRates().");
            }
        } catch (Exception e) {
            LOGGER.severe("Error fetching ALL CarparkRates from URA: " + e.getMessage());
        }
    }
```

```java
public class ParkingSessionService {
    public ParkingSessionDTO startSession(Long driverId, String carparkId,Integer selectedMinutes, String lotType) {
        Optional<Driver> driverOpt = driverRepository.findById(driverId);
        Optional<Carpark> carparkOpt = carparkRepository.findByCarparkId(carparkId);

        if (!driverOpt.isPresent()) throw new IllegalStateException("Driver not found");
        if (!carparkOpt.isPresent()) throw new IllegalStateException("Carpark not found");

        Driver driver = driverOpt.get();
        Carpark carpark = carparkOpt.get();
        LocalDateTime now = LocalDateTime.now();

        List<CarparkRates> rates = carparkRatesRepository.findByPpCode(carparkId);
        CarparkRates applicableRate = getApplicableRate(rates, now);

        float hourlyRate = getRateForDay(applicableRate, now);
        double estimatedCost = Math.round(((selectedMinutes / 60.0) * hourlyRate) * 100.0) / 100.0;

        ParkingSession session = new ParkingSession();
        session.setDriver(driver);
        session.setCarpark(carpark);
        session.setStartTime(now);
        session.setHourlyRate(hourlyRate);
        session.setLotType(lotType);
        session.setLotNumber("B15"); // Optional
        session.setEstimatedCost(estimatedCost);  // Optional: Add to your model
        session.setSelectedMinutes(selectedMinutes);     // Save expected duration (optional)

        ParkingSession savedSession = parkingSessionRepository.save(session);
        return new ParkingSessionDTO(savedSession);
    }
```

DESIGN PATTERNS

1. Static Factory Pattern
2. Singleton Pattern

STATIC FACTORY PATTERN

A static method that returns an instance of a class, used in DriverResponse.java.

Pros:

- Promotes encapsulation and data hiding
- Ensures separation of concerns

```
public static DriverResponse fromDriver(Driver driver) {
    DriverResponse response = new DriverResponse();
    response.setId(driver.getId());
    response.setEmail(driver.getEmail());
    response.setName(driver.getName());

    if (driver.getProfilePhoto() != null) {
        response.setProfilePhotoBase64(Base64.getEncoder().encodeToString(driver.getProfilePhoto()));
    }

    response.setCreatedAt(driver.getCreatedAt());
    return response;
}
```

SINGLETON PATTERN

- Used in Spring IoC container where beans are created as singletons
- Saves memory by maintaining a single instance
- Risk of race conditions with mutable shared state in multi-threaded environments

```java
@Service
public class FavouriteCarparkService {

    private final FavouriteCarparkRepository favouriteCarparkRepository;
    private final DriverRepository driverRepository;
    private final CarparkRepository carparkRepository;

    @Autowired
    public FavouriteCarparkService(FavouriteCarparkRepository favouriteCarparkRepository,
                                   DriverRepository driverRepository,
                                   CarparkRepository carparkRepository) {
        this.favouriteCarparkRepository = favouriteCarparkRepository;
        this.driverRepository = driverRepository;
        this.carparkRepository = carparkRepository;
    }
```

FUTURE PLANS

ADVANCED RESERVATION SYSTEM

- Pre-book parking slots for specific times
- Real-time availability updates
- Flexible edit/cancel options

IN-APP PAYMENTS & WALLET

- Pay parking fees directly using ParkIT secure wallet in the app
- View transaction history & receipts
- Plan to integerate with Parking.sg

BUSINESS SPECIFIC

- Easily extend to lorry business specific
- Track and analyse multiple lorries sessions under a business
- Use ParkIT AI to help optimise cost savings

CONCLUSION

PARK *SMARTER*. DRIVE *SMOOTHER*. WITH PARKIT