Phase 5-IOT

SMART PARKING

TOPICS

- **Project Overview**
- **Requirements**
- **Architecture and Design**
- **Development**

Objectives:

- Develop a user-friendly mobile app for Android platforms.
- Implement real-time parking space availability tracking.
- Enable secure and convenient payment processing.
- Create an admin dashboard for parking facility management.
- Improve user experience and reduce parking congestion in urban areas.

Introduction

In the ever-evolving urban landscape, one of the most persistent challenges we face daily is finding a parking spot. Whether it's a congested downtown area, a bustling shopping mall, or a crowded event venue, the search for available parking spaces can be a frustrating and time-consuming ordeal. In response to this common urban dilemma, we present the Smart Parking Application, a modern and innovative solution designed to transform the way we park.

The Problem

With the increasing number of vehicles on the road, the demand for parking spaces has reached unprecedented levels. Traditional parking management methods often fall short, resulting in congestion, pollution, and wasted time for both drivers and parking facility operators. Finding a parking spot should not be a hassle, and managing parking facilities should not be a guessing game. There is a pressing need for a smarter, more efficient approach to parking.

The Solution

The Smart Parking Application offers an elegant solution to the parking challenge by harnessing the power of technology. By seamlessly integrating real-time data, user-friendly interfaces, and efficient communication, this application promises to:

- **Simplify Parking:** Finding an available parking space will be as easy as tapping your smartphone screen. Real-time data will guide you to the nearest available spot, reducing the stress of circling the block.
- **Save Time:** No more wasting valuable time in endless searches. The application enables you to reserve parking spaces in advance, so you can plan your day with confidence.
- **Enhance Efficiency:** Parking facility operators can manage their spaces more effectively, optimizing space allocation and increasing revenue through online reservations.
- **Promote Sustainability:** By reducing unnecessary driving and congestion, the application contributes to a more sustainable urban environment with less pollution and fuel consumption.

User Experience

Our application is designed with you in mind. It provides an intuitive and user-friendly interface that simplifies the entire parking process. From finding and reserving a parking space to secure and hassle-free payment, the Smart Parking Application is set to revolutionize the way you park.

In this document, we will delve into the comprehensive features, architecture, and requirements of the Smart Parking Application, shedding light on the technology that makes it possible. We invite you to explore the world of hassle-free parking and a brighter, more efficient urban future with our smart parking solution.

Requirements:

1. Arduino (Sensor Integration)

Functional Requirements:

- **Sensor Integration:** Connect sensors (e.g., ultrasonic, infrared) to Arduino for monitoring parking space occupancy.
- **Data Collection:** Collect sensor data and transmit it to the Raspberry Pi for processing.
- **Reliability:** Ensure accurate and reliable sensor readings.
- **Low Power Mode:** Implement a low-power mode when parking spaces are unoccupied to save energy.

2. Android Application

Functional Requirements:

- **Parking Space Reservation:** Allow users to reserve parking spaces in advance.
- **Feedback and Support:** Offer a feedback mechanism and support contact for users.
- **User Interface:** Create an intuitive and user-friendly mobile app interface.

Non-Functional Requirements:

- **Performance:** Ensure fast response times, even during peak usage..
- **Cross-Platform Compatibility:** Develop the app for both Android smartphones and tablets..

3. Raspberry Pi (Backend)

Functional Requirements:

- **Data Collection:** Raspberry Pi should be capable of collecting real-time data from parking sensors (connected via Arduino) or other data sources.
- **Data Processing:** Implement data processing to monitor parking space availability and update it in real-time.
- **Communication:** Establish a communication mechanism to send parking data to the Android app.
- **Database:** Store and manage historical data, including reservations, user information, and parking history.
- **Data Security:** Ensure data encryption and secure communication.

4. Arduino (Sensor Integration)

Functional Requirements:

- **Sensor Integration:** Connect sensors (e.g., ultrasonic, infrared) to Arduino for monitoring parking space occupancy.
- **Data Collection:** Collect sensor data and transmit it to the Raspberry Pi for processing.
- **Reliability:** Ensure accurate and reliable sensor readings.
- **Low Power Mode:** Implement a low-power mode when parking spaces are unoccupied to save energy.

Architecture:

- **Client-Server Model:** Implement a client-server architecture where the mobile app (client) interacts with a server for data processing, storage, and real-time updates.
- **Components:**
- **Mobile App (Client):** The front-end of the application, developed in Kotlin for Android, responsible for user interactions, user interface, and communication with the server.
- **Server:** The back-end of the application, implemented in Python or another appropriate server-side language, responsible for data processing, storage, and communication with IoT devices and external services.
- **Database:** Store information related to parking spaces, reservations, user accounts, transaction history, and more. Consider using a relational database (e.g., MySQL) or a NoSQL database (e.g., MongoDB) depending on your needs.
- **IoT Devices (e.g., Arduino):** Devices that collect real-time data from parking sensors and relay it to the server. The server processes this data to update parking availability.
- **External Services:** Integrate with external services for payment processing, map services, and notification delivery.
- **Communication:**
- Use RESTful APIs for communication between the mobile app and the server. These APIs allow for seamless data exchange.
- Implement real-time updates using WebSockets or a similar technology to keep users informed about parking availability changes.
- **Scalability:**
- Design the server to be scalable, so it can handle increased traffic as more users and parking spaces are added.
- Utilize load balancing to distribute requests evenly across multiple server instances.

Design:

- **User Interface (UI):**
- Create an intuitive and user-friendly UI with easy navigation and quick access to core features.
- Use a mobile-first design approach to ensure that the application functions well on various screen sizes.

- Include features such as a map view of parking spaces, search and filter options, reservation functionality, and user profile management.
- **Authentication and Authorization:**
- Implement a secure and user-friendly authentication system to protect user accounts and data.
- Use industry-standard authentication protocols like OAuth for user sign-in.
- **Parking Space Map:**
- Display a map with real-time updates on parking space availability. Use color-coding or icons to indicate available and occupied spaces.
- Allow users to zoom in and out, pan, and search for parking spaces based on location.

The development of a smart parking application involves multiple steps, including coding, testing, and deployment.

Requirement Analysis:

- Review the functional and non-functional requirements outlined earlier.
- Ensure that you have a clear understanding of the application's purpose and scope.

Front-End Development (Android Application):

- Design the user interface based on the UI/UX design principles.
- Develop the Android app using Kotlin.
- Implement user registration, authentication, and profile management.
- Create the parking space search feature, reservation flow, payment integration, and notification system.
- Test the app on various devices and screen sizes.

Back-End Development (Server-Side):

- Design the server architecture and APIs.
- Develop the server application using Python or your chosen server-side language.
- Implement data processing, storage, user management, and communication with IoT devices and external services.
- Set up a secure authentication system.
- Ensure the server can handle data in real-time.

IoT Device Integration:

- Program the IoT devices (e.g., Arduino) to collect data from parking sensors.
- Develop communication protocols for transmitting data to the server.
- Ensure that the IoT devices can handle data in real-time and are reliable.

Deployment:

- Deploy the server application on a reliable server infrastructure.
- Publish the Android app on the Google Play Store.
- Ensure that the application is available to users.

STEPS:

User Registration and Login: This is essential to track users and their parking reservations.

1. Create a New Android Project:

First, create a new Android project in Android Studio.

2. Design the User Interface:

Design your app's user interface using XML layout files. Here's an example of a simple layout for a parking application with a search feature:

XML CODE:

```
<!-- activity_main.xml -->
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <EditText
      android:id="@+id/searchEditText"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="Enter location or address"/>

   <Button
      android:id="@+id/searchButton"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Search"/>
    <!-- Add a ListView or RecyclerView to display search results -->
</RelativeLayout>
```

3. Create Java Code:

Now, create the Java code to handle user interactions and display search results.

JAVA CODE:

```
public class MainActivity extends AppCompatActivity {

    private EditText searchEditText;
    private Button searchButton;
    private ListView resultsListView; // You can also use RecyclerView for better performance

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        searchEditText = findViewById(R.id.searchEditText);
        searchButton = findViewById(R.id.searchButton);
        resultsListView = findViewById(R.id.resultsListView);

        searchButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
```

```
            String query = searchEditText.getText().toString();
        }
    });
}}
```

4. Handle API Requests:

In a real-world application, you'd make API requests to a backend server that manages parking data and returns search results. Here's a simplified example of making an API request using the Retrofit library.

JAVA CODE:

```
public class RetrofitClient {

    private static Retrofit retrofit;
    private static final String BASE_URL = "https://your-api-url.com/";

    public static Retrofit getClient() {
        if (retrofit == null) {
            retrofit = new Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build();
        }
        return retrofit;
    }
}
```

5. Define API Endpoints:

Define your API endpoints using Retrofit annotations.

6. Handle API Responses:

Handle the API responses and update the UI with the search results.

JAVA CODE:

```
ParkingApiService apiService = RetrofitClient.getClient().create(ParkingApiService.class);


Call<List<ParkingLocation>> call = apiService.searchParking(query);

call.enqueue(new Callback<List<ParkingLocation>>() {

   @Override

   public void onResponse(Call<List<ParkingLocation>> call,
Response<List<ParkingLocation>> response) {

      if (response.isSuccessful()) {

         List<ParkingLocation> parkingLocations = response.body();

         // Update the ListView with the parking locations

      } else {

         // Handle error

      }

   }


   @Override

   public void onFailure(Call<List<ParkingLocation>> call, Throwable t) {

      // Handle network error

   }

});
```

7. Create the ParkingLocation Model:

Create a model class to represent parking locations.

JAVA CODE:

```
public class ParkingLocation {
    private String name;
    private String address;
    private double latitude
  private double longitude;
}
```

USER INTERFACE(UI FOR MOBILE APPLICATION):

CIRCUIT:

To collect data from sensors connected to a Raspberry Pi and send that data to a cloud server or a mobile app server, you'll need to write Python scripts that use suitable libraries and APIs for data

transmission. Below is a basic example of Python scripts for collecting data from sensors (assuming ultrasonic distance sensors) and sending it to a cloud server.

CIRCUIT CODE:

```
const int trigPin1 = 27;

const int echoPin1 = 26;


const int trigPin2 = 2;

const int echoPin2 = 15;


const int trigPin3 = 18;

const int echoPin3 = 5;


const int ledPin1 = 13;

const int ledPin2 = 12;

const int ledPin3 = 14;


long duration;

int distance;


void setup () {
  pinMode(trigPin1, OUTPUT);
  pinMode(echoPin1, INPUT);


  pinMode(trigPin2, OUTPUT);
  pinMode(echoPin2, INPUT);


  pinMode(trigPin3, OUTPUT);
  pinMode(echoPin3, INPUT);
```

```
   pinMode(ledPin1, OUTPUT);
   pinMode(ledPin2, OUTPUT);
   pinMode(ledPin3, OUTPUT);

  Serial.begin(9600);
}
void loop() {
  digitalWrite(trigPin1, LOW);
  delayMicroseconds(2);

  digitalWrite(trigPin1, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin1, LOW);

  duration = pulseIn(echoPin1, HIGH);

  distance = duration * 0.034 / 2;

if (distance < 200) {
  digitalWrite(ledPin1, LOW);
  Serial.println("Parking Space 1 : Occupied");
}else {
  digitalWrite(ledPin1, HIGH);
  Serial.println("Parking Space1 : Vacant");
}
delay(1000);
```

```
  Serial.print("Distance");
  Serial.println(distance);

   digitalWrite(trigPin2, LOW);
  delayMicroseconds(2);

  digitalWrite(trigPin2, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin2, LOW);

  duration = pulseIn(echoPin2, HIGH);

  distance = duration * 0.034 / 2;

if (distance < 200) {
  digitalWrite(ledPin2, LOW);
  Serial.println("Parking Space 2 : Occupied");
}else {
  digitalWrite(ledPin2, HIGH);
  Serial.println("Parking Space2 : Vacant");
}
delay(1000);

  Serial.print("Distance");
  Serial.println(distance);

   digitalWrite(trigPin3, LOW);
  delayMicroseconds(2);
```

```
  digitalWrite(trigPin3, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin3, LOW);

  duration = pulseIn(echoPin3, HIGH);

  distance = duration * 0.034 / 2;

if (distance < 200) {
  digitalWrite(ledPin3, LOW);
  Serial.println("Parking Space 3 : Occupied");
}else {
  digitalWrite(ledPin3, HIGH);
  Serial.println("Parking Space3 : Vacant");
}
delay(1000);

  Serial.print("Distance");
  Serial.println(distance);
}
```

Python script that collects data from an ultrasonic sensor and sends it to a cloud server:

```
import RPi.GPIO as GPIO
import time
import requests
TRIG_PIN = 23
ECHO_PIN = 24
```

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(TRIG_PIN, GPIO.OUT)
GPIO.setup(ECHO_PIN, GPIO.IN)
def get_distance():
# Send a short pulse to trigger the ultrasonic sensor
GPIO.output(TRIG_PIN, True)time.sleep(0.00001)
GPIO.output(TRIG_PIN, False)
while GPIO.input(ECHO_PIN) == 0:
pulse_start = time.time()
while GPIO.input(ECHO_PIN) == 1:
pulse_end = time.time()

pulse_duration = pulse_end - pulse_start
distance = pulse_duration * 17150 # Speed of sound in cm/s
distance = round(distance, 2)
return distance
try:
while True:
distance = get_distance()
print(f"Distance: {distance} cm")

server_url = " "//URL
payload = {"distance": distance}
headers = {"Content-Type": "application/json"}
response = requests.post(server_url, json=payload, headers=headers)
if response.status_code == 200:
print("Data sent successfully")
else:
```

```
print("Failed to send data")
time.sleep(5)
except KeyboardInterrupt:
GPIO.cleanup()
```

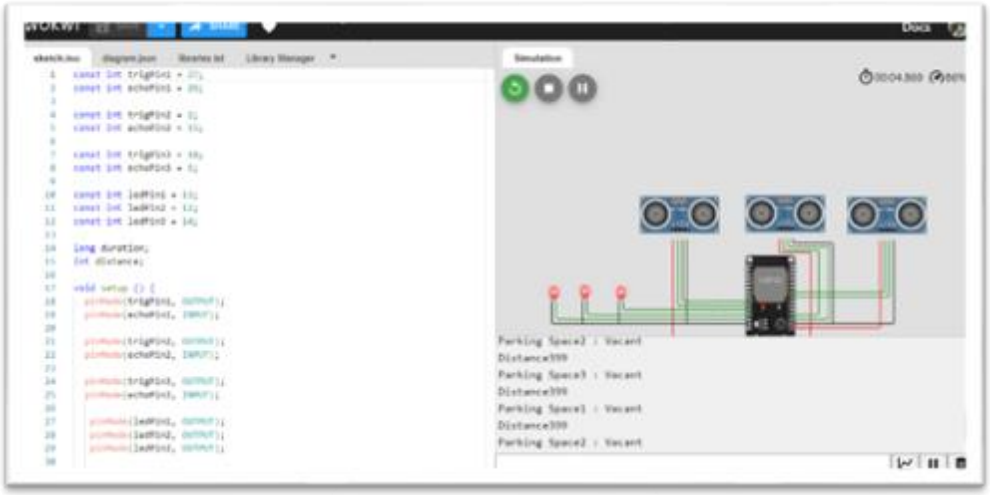
KOTLINE(FOR UI):

```
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class ParkingSpaceSearchActivity : AppCompatActivity() {
    private lateinit var locationEditText: EditText
    private lateinit var searchButton: Button
    private lateinit var resultTextView: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_parking_space_search)

        locationEditText = findViewById(R.id.editTextLocation)
        searchButton = findViewById(R.id.buttonSearch)
        resultTextView = findViewById(R.id.textViewResult)

        searchButton.setOnClickListener {
            val location = locationEditText.text.toString()
            // Call a function to perform the parking space search based on the location input
```

```
            val searchResult = performParkingSpaceSearch(location)
            resultTextView.text = searchResult
        }
    }

    private fun performParkingSpaceSearch(location: String): String {
        <?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    android:gravity="center">

    <TextView
        android:id="@+id/textViewTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Find Parking Space"
        android:textSize="24sp"
        android:layout_gravity="center"/>

    <EditText
        android:id="@+id/editTextLocation"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:hint="Enter Location"
        android:inputType="text"
        android:layout_marginTop="16dp"/>

    <Button
        android:id="@+id/buttonSearch"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Search"
        android:layout_marginTop="16dp"/>

    <TextView
        android:id="@+id/textViewResult"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=""
        android:textSize="18sp"
        android:layout_marginTop="16dp"/>
</LinearLayout>
        return "Found 3 available parking spaces near $location."
    }
}
```

CIRCUIT DESIGN:


Parking Space2 : Vacant
Distance999
Parking Space3 : Vacant
Distance999
Parking Space1 : Vacant
Distance999
Parking Space2 : Vacant