SMART PARKING

DEVELOPING SMART PARKING SYSTEM

Developing a smart parking system using Arduino is another viable option for creating a costeffective and customizable solution. Here's a basic guide on how to develop a smart parking system using Arduino: Components and Materials: Arduino Board: Choose a suitable Arduino board (e.g., Arduino Uno, Arduino Mega) depending on the number of parking spaces and sensors you plan to use. Proximity Sensors: Use ultrasonic sensors or IR sensors to detect vehicle presence in parking spaces. LED Displays: LED displays or status lights to show the availability of parking spots. Wi-Fi/Bluetooth Module: To connect your Arduino to the internet and enable data transmission and remote monitoring. Mobile App/Web Interface: Develop a mobile app or web interface for users to check parking availability and make reservations. Database: Set up a database to store information about parking spaces, reservations, and user data. Server: Develop a server application that communicates with the Arduino and the database. You can use platforms like Raspberry Pi or a cloud-based server. Development Steps: Set up Arduino: Load the Arduino IDE and install the necessary libraries. Configure the Arduino board and Wi-Fi/Bluetooth module for network connectivity. Sensor Setup: Connect proximity sensors to the Arduino board. Write Arduino code to read data from the sensors and detect vehicle presence. Database and Server: Set up a database to store information about parking spaces, reservations, and user data. Develop a server application that communicates with the Arduino and the database. You can use platforms like Raspberry Pi or a cloud-based server. User Interface (App or Web): Create a mobile app or web interface that allows users to check parking availability and make reservations.

 Integrate payment gateways for online payments if needed. Data Processing and Display: Write code on the Arduino to send data about parking space availability to the server. Develop code to update LED displays or status lights to indicate the availability of parking spots. Security and Access Control: Implement security measures to protect the system from unauthorized access. Create user authentication and authorization processes. Testing and Debugging: Thoroughly test the system to ensure accurate sensor readings, data transmission, and user interface functionality. Debug any issues that arise during testing. Deployment: Install the Arduino and sensors in the parking area. Ensure a stable power supply. User Training and Support: Educate users on how to use the smart parking system. Provide support channels for user inquiries and issues. Monitoring and Maintenance: Continuously monitor the system's performance and resolve any issues that may arise. Plan for regular maintenance and updates to the system. Consider scalability, security, and the specific needs of your parking area when developing a smart parking system using Arduino. Regularly update and maintain the system to ensure its continued functionality and reliability.

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:

```xml
<!-- 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;
}
```

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:

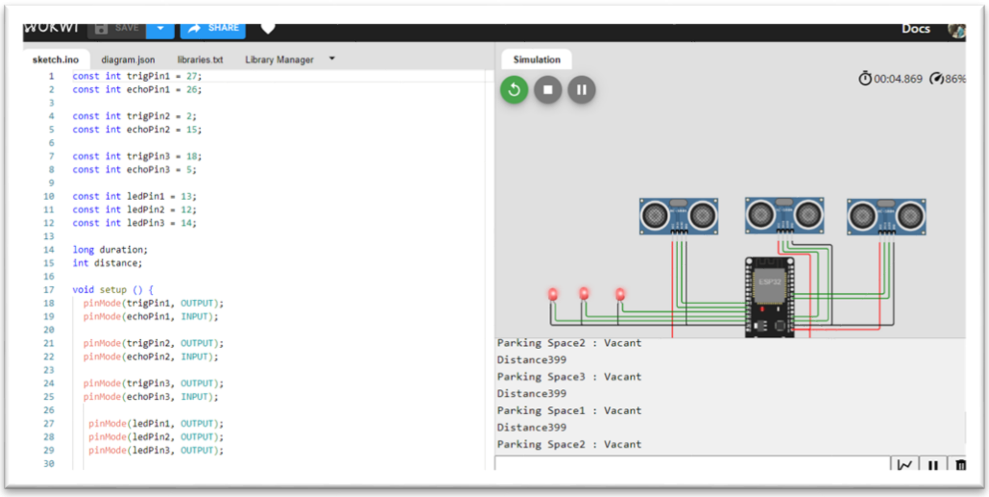
```
const int trigPin1 = 27;

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()
```



```
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);

```

```
Parking Space2 : Vacant
Distance399
Parking Space3 : Vacant
Distance399
Parking Space1 : Vacant
Distance399
Parking Space2 : Vacant
```