ParkPing Project Starter Package with UI, Flask Backend, SQLite, and Google Maps

ParkPing – Real-Time Street Parking Alerts (Starter Project)

Overview

ParkPing is a developer-friendly starter project for a real-time street parking alert platform. It provides a complete, responsive web application that lets users find and book nearby parking spots in real time. The project is structured with a **Bootstrap 5** front-end (styled similar to Uber/Uber Eats flows) and a **Flask** back-end, using **SQLite** for data storage. Key features include:

- **Responsive UI** – A mobile-friendly interface with a source/destination input and a live Google Map for nearby parking spots.
- **Google Maps Integration** – Displays the user's current location and available parking spots on an interactive map.
- **User Accounts** – Registration and login system backed by SQLite (passwords hashed for security).
- **Booking & Payment Flow** – Users can reserve a parking spot, see cost calculated per hour, and simulate payment completion.
- **Notifications** – Popup alerts (and placeholders for email notifications) inform users of important events (e.g., spot booked, payment confirmation).
- **Scalable Architecture** – Flask REST API organized in a microservices-style layout for auth, parking, and payment logic, ready to expand.

Features

- **Uber-Style UI:** The interface follows a familiar ride-hailing layout with an address input (destination) and a map. Users can input a destination and see parking spots near that location. The design uses Bootstrap for a clean, modern look and is fully responsive.
- **Real-Time Map Updates:** The app uses the Google Maps JavaScript API to show available parking spots around the user. A marker indicates the user's current position and other markers show free parking spots. The map auto-centers on the user's location (if permitted) or a default city center.
- **User Registration & Login:** Visitors can create an account and login. Authentication is handled via Flask sessions. All user info is stored in a local SQLite database. Passwords are stored securely (hashed via Werkzeug).
- **Parking Spot Management:** The back-end defines a simple model for parking spots (with fields like address, coordinates, hourly rate, and availability). Some sample spots are pre-loaded for demonstration. Users can fetch available spots and book them, which flips their availability status.
- **Booking Confirmation & Mock Payment:** When a user books a spot, they specify the duration (hours) and the cost is calculated (rate × hours). The booking is recorded in the database and the spot is marked as unavailable. Users are then prompted to "pay" – the payment is simulated by an API call that marks the booking as paid. This flow mimics a real booking confirmation and payment process.
- **Notifications:** The front-end provides feedback via alerts/confirmation dialogs for actions like successful booking or payment. The project is set up to easily integrate more advanced notifications (e.g., Bootstrap toasts for non-intrusive alerts – *"Toasts are lightweight notifications designed to mimic the push notifications…"*getbootstrap.com). The back-end also includes a placeholder print statement to simulate sending an email confirmation, indicating where an email API or SMTP call would occur (*Python's built-in `smtplib` can send emails via SMTP*realpython.com).

Project Architecture

This starter project is split into **front-end** and **back-end** components, organized for clarity and scalability. The Flask back-end uses Blueprints (modular route groupings) to simulate a microservices architecture: there are separate blueprint modules for authentication, parking, and payments. This keeps the code organized by concern, which is important as the application growsdigitalocean.comdigitalocean.com. In a larger project, each module could even become an independent service. Flask's blueprint system helps achieve this separation of concerns (*Flask provides a feature called blueprints to organize your application into components for each feature*digitalocean.com).

Project structure:

```
plaintext
ParkPing/
├── backend/
│   └── app.py            <- Flask app with REST API (auth, parking, payment)
├── frontend/
│   ├── templates/        <- HTML files (index, login, register)
│   └── static/           <- Static assets (JavaScript, CSS)
└── ParkPingStarter.zip   <- Packaged project zip (for download)
```

- **Front-end:** Built with HTML5 and Bootstrap 5 (no Tailwind). The front-end is a single-page app for logged-in users (`index.html`) plus separate pages for login and registration. It communicates with the Flask API via AJAX (Fetch API calls in `lib.js`). The UI layout uses a Bootstrap navbar (with the app brand and a logout button) and a search bar for destination input. Below that is a full-width map. The design is mobile-responsive using Bootstrap's grid and utility classes.

- **Back-end:** A Flask application (`app.py`) provides RESTful endpoints under an `/api/*` namespace. It uses Flask's session for user login state and SQLite for persistent storage. The code is organized into logical sections (or Blueprints) for different domains (Auth, Parking, Payment), which makes the app easier to maintain and extenddigitalocean.com. For simplicity, all back-end code resides in one file in this starter, but it's structured with clear separation of functions for each feature.

- **Database:** SQLite is chosen for its zero-configuration nature. The schema has three tables: **users** (for login credentials), **parking_spots** (for spot info and availability), and **bookings** (to track reservations, times, and payment status). The Flask app uses a lightweight approach to database connections, opening a connection when needed and ensuring it's closed after each request (*using Flask's application context `g` to hold a connection per requestflask.palletsprojects.com*). On startup, the app initializes the database schema if not already present, and inserts a few sample parking spots for testing.

Frontend: UI & Google Maps Integration

The front-end uses **Bootstrap 5** for styling, ensuring the app looks good on desktop and mobile. The layout resembles Uber's approach: a top bar for navigation and input, and the main area dedicated to the map.

- **Map Display:** The app integrates Google Maps via the Maps JavaScript API. In the HTML, a script tag loads the Google Maps API with a callback to `initMap`. (You must obtain a Google Maps API key and enable the Maps JS API to use this – *Google's documentation explains how to get an API key via Google Cloud Consoledevelopers.google.com*). When the API script loads, it calls the `initMap()` function defined in our `lib.js`. This function creates a new map centered on a default location and then tries to get the user's actual location via the browser's Geolocation API.

- **Geolocation:** If the user grants permission, the app uses `navigator.geolocation.getCurrentPosition` to get the device's latitude/longitude and recenters the map there, placing a special marker ("You are here") to denote the user's position. (Note: For security, many browsers require that geolocation is used only on secure contexts – e.g., https or localhost – otherwise it may failstackoverflow.com. When developing, run the app on `http://localhost` or consider enabling HTTPS for testing to allow geolocation). If geolocation is unavailable or denied, the map stays at the default center (configurable; in this starter, it's set to a downtown area).

- **Parking Markers:** After initializing the map, the front-end script fetches the list of available parking spots from the back-end (`GET /api/parking/spots`). For each spot returned, a Google Maps Marker is placed on the map. Clicking a parking spot marker opens an info window showing the spot's address and hourly rate, along with a "Book this spot" button. This provides a real-time visualization of parking availability around the user.

- **Destination Search:** The UI includes a destination search field. A user can enter an address (e.g., their destination), and on clicking "Find Parking", the app will geocode the address using Google's Geocoding service (available via the Maps API). If found, the map centers on that location and drops a "destination" marker (blue pin) to mark it. Currently, all parking spots are still shown (the example back-end does not yet filter spots by proximity to the destination), but this feature sets the stage for adding such filtering or even integrating Google Maps Directions in the future. *(Creating a map with a marker involves obtaining an API key, an HTML container, and adding the map via JavaScriptdevelopers.google.com – which our project handles through the included script and `initMap` function.)*

- **Responsive Design:** Thanks to Bootstrap, the layout adjusts to different screen sizes. The map container uses a flexible height (75vh – 75% of the viewport height) to ensure it's visible without scrolling on most devices. The input and buttons stack on smaller screens. The use of Bootstrap's classes (like grid columns, `.btn`, `.form-control`, etc.) means the UI is immediately usable on phones and tablets.

Backend: Flask API & SQLite Database

The back-end is a **Flask** application (`app.py`) that serves both the front-end pages and the JSON APIs. It's configured to serve the HTML from a `templates` folder and static files (JS/CSS) from a `static` folder. When the Flask server is running, navigating to the root (`/`) will render `index.html` if the user is logged in (determined by a session cookie), or redirect to the login page if not.

Key Flask components and routes:

- **Application Structure:** We use a single Flask app with multiple route groupings. In a larger application, we would use Flask Blueprints to organize routes by module (auth, parking, etc.). In this starter, for simplicity, the routes are defined in one file but logically separated by feature (comments and ordering). This modular approach makes the code easier to navigate and scale; each part of the app (auth, parking, payment) can be understood in isolationdigitalocean.com.

- **Database Connection:** The app uses a simple pattern to handle the SQLite database: a `get_db()` function opens a connection (stored in Flask's `g` object) on first use per request, and a `teardown_appcontext` closes it after the request endsflask.palletsprojects.com. This pattern avoids having to open/close connections manually in every route. The database file is `parkping.db` (automatically created in the project directory). The schema is set up in `init_db()` which is called at startup – it creates tables if they don't exist.

- **Data Models:** There are three tables:

    - **users:** (id, email, password) – stores user credentials. The password is hashed for security using Werkzeug's utility.

- **parking_spots:** (id, address, lat, lng, rate, is_available) – stores parking spot info. In a real app, this might be populated via sensors or an admin interface; here we seed a few rows for demo. `is_available` is a boolean (1/0) indicating if the spot is free.

- **bookings:** (id, user_id, spot_id, start_time, end_time, cost, paid) – records each booking/reservation. It tracks which user booked which spot, the start/end times of the reservation, the calculated cost, and whether payment is completed.

- **Auth Endpoints:**

  - `POST /api/auth/register` – accepts JSON with email & password. Creates a new user account (after hashing the password). If the email already exists, it returns an error. On success, returns a JSON message. (In a production system, you'd likely also validate the email format and enforce password strength.)

  - `POST /api/auth/login` – accepts JSON credentials, verifies the user (checking the hashed password). On success, it uses Flask's session to store the user's ID (logged-in state) and returns a success message. On failure, returns an error JSON. The session approach means subsequent requests from the same client will remain authenticated (session cookie is sent automatically).

  - `POST /api/auth/logout` – logs the user out by clearing the session.

- **Parking Endpoints:**

  - `GET /api/parking/spots` – returns a JSON list of all currently available parking spots. This route is protected (requires login; it checks for `user_id` in session). The response format is `{ "spots": [ {id, address, lat, lng, rate}, ... ] }`. The front-end uses this to plot markers on the map. (In a real scenario, this could accept query parameters for filtering by location/radius).

  - `POST /api/parking/book` – used to book/reserve a parking spot. Expects JSON like `{ "spot_id": <ID>, "hours": <number> }`. This is also protected (requires login). The route checks the requested spot in the database: if it's available, it calculates the cost (`rate * hours`), marks the spot as unavailable, and creates a booking entry with start time = now and end time = now + given hours. The `paid` flag in the booking starts as 0 (not paid yet). The API responds with the booking ID and cost (JSON). If the spot was already taken or any error occurs, an error JSON is returned. After booking, the server prints a notification message to the console as a placeholder for sending an email notification (e.g., "Confirmation email sent to user") – this is where you could integrate an email service in the future.

  (The booking flow simulates reserving a spot in real-time. The cost calculation is straightforward in this demo, but it could be extended (e.g., different rates for different spots or dynamic pricing).)

- **Payment Endpoint:**

  - `POST /api/payment/complete` – simulates completing a payment for a booking. Expects JSON `{ "booking_id": <ID> }` and requires login. It checks that the booking exists and belongs to the current user, and that it isn't already paid. Then it updates the booking's `paid` status to 1 (paid). In a real application, this is where integration with a payment gateway (Stripe, PayPal, etc.) would happen. Here we just assume payment is successful and return a confirmation JSON. (The spot remains marked as unavailable; one could add logic to automatically free the spot after the end time or allow the user to manually release it.)

All API responses are JSON for easy consumption by the front-end. HTTP status codes are used to indicate success (200 OK or 201 Created) or errors (400 for bad input, 401 for unauthorized, etc.).

Security Notes: This starter uses basic session-based auth (Flask's signed cookie). It's sufficient for a demo, but for production you'd ensure to use HTTPS to protect session cookies, possibly implement token-based auth for a more decoupled client, and handle password reset flows, etc. The password hashing uses Werkzeug's safe hash (PBKDF2 by default) which is a solid start; in a real project you might choose an even stronger hashing like bcrypt.

Notifications & Alerts

Currently, user-facing notifications are implemented with simple JavaScript alerts and `confirm` dialogs. For example, after booking a spot, the front-end will `confirm()` with the calculated cost asking the user to proceed with payment. Alerts are used to show error messages (e.g., if booking fails or payment error) or success confirmation (payment success). This approach makes it clear when an action occurs, but it's blocking and not stylized.

As an enhancement, the project could use **Bootstrap Toasts** or modals for a nicer UX. *Bootstrap toasts are ideal for this: "Toasts are lightweight notifications designed to mimic push notifications"* getbootstrap.com, which could slide in non-intrusively instead of alert boxes. The starter code is structured to allow easy insertion of such components. For example, one could replace the `alert()` calls in `lib.js` with functions that generate toast elements and show them.

For **real-time alerts** (for instance, notifying a user that a new parking spot just became free near them), a further step would be integrating WebSockets or Server-Sent Events. With that, the server could push a notification to the client without the client constantly polling. This is noted in the roadmap as a future improvement. In the current project, the assumption is that "real-time" is achieved by refreshing or by the user initiating a search, due to scope.

The back-end has a placeholder for email notifications. After a booking is made, it prints a message indicating an email would be sent. In practice, you could plug in an email module (such as Flask-Mail or an SMTP library) to send an actual email to the user confirming their reservation.

Python's standard library `smtplib` could be used for SMTP communication (*"Python comes with the built-in smtplib module for sending emails using SMTP."*realpython.com). Since this is a starter template, actual email sending isn't configured, but the extension points are there.

API Endpoints Summary

For clarity, here's a list of the provided API endpoints and their purposes (all prefixed by `/api/` and expecting/returning JSON):

- **`POST auth/register`** – Create a new user account. **Request:** JSON `{email, password}`. **Response:** JSON message or error. *(Returns 201 on success, or 400 if the email is already registered.)*
- **`POST auth/login`** – Log in a user. **Request:** JSON `{email, password}`. **Response:** JSON success or error message. On success, a session cookie is set. *(Returns 200 on success, 401 on invalid credentials.)*
- **`POST auth/logout`** – Log out the current user. **Request:** none (uses session). **Response:** JSON message. *(Session is cleared server-side; cookie invalidated.)*
- **`GET parking/spots`** – Retrieve available parking spots. **Request:** none (requires user to be logged in). **Response:** JSON `{spots: [ {...}, ...] }` with each spot's id, address, coordinates, and hourly rate. *(Returns 200 with data, or 401 if not logged in.)*
- **`POST parking/book`** – Book a parking spot. **Request:** JSON `{spot_id, hours}`. **Response:** JSON with booking confirmation `{booking_id, cost, message}` or an error. Books the spot for the specified duration (in hours) starting now, marks it unavailable, and calculates the cost. *(Returns 200 on success, 400/404 for errors like invalid spot or spot already booked, 401 if not logged in.)* 【26†output】
- **`POST payment/complete`** – Complete payment for a booking. **Request:** JSON `{booking_id}`. **Response:** JSON message (and cost) or error. Marks the booking as paid. *(Returns 200 on success, 400 if already paid, 404 if booking not found, 401 if not logged in.)*

These endpoints together implement the core logic: account management, listing parking spots, booking a spot (reservation), and simulating payment. The front-end `lib.js` contains the logic to call these endpoints and handle responses to provide a smooth user experience.

Running the Project

To get started with this project on your local machine:

1. **Prerequisites:** Install Python 3 and ensure you have an environment where you can install packages. Clone or unpack the project files. (All dependencies are standard Python libraries except Flask. You can install Flask with `pip install Flask`.)
2. **Google API Key:** Sign up for a Google Cloud account if you haven't, and enable the **Google Maps JavaScript API** for your project. Obtain an API key. *(Google's documentation shows how to get an API key in the Cloud consoledevelopers.google.com.)* In the `frontend/templates/index.html` file, find the script tag that includes the Maps API and replace `YOUR_API_KEY` with your actual key.
3. **Initialize Database:** The first run will create a SQLite database file (`parkping.db`). The app will do this automatically. If you want to adjust initial data, you can edit the sample spots inserted in `app.py` (near the bottom) or run your own SQL. No separate migration step is needed.
4. **Run the Server:** In a terminal, navigate to the `backend/` directory and run `python app.py`. Flask will start the development server on http://127.0.0.1:5000 by default. (The app is configured with `debug=True`, so you'll see live reloads and detailed error traces in the console if any.)
5. **Open the App:** Visit **http://localhost:5000** in your web browser. You should see the login page. (If you see an error about Google Maps, double-check that your API key is correctly inserted and that the API is enabled for that key.)
6. **Test the Features:** Register a new user on the Register page. After submitting, it will redirect you to login – log in with the credentials. Once logged in, you'll see the main map page. The map will attempt to locate you (look for a permission prompt in your browser). You can use the search bar to center the map on a location of your choice (e.g., try a city name or address). Markers in the map show the seeded parking spots. Click a marker to see its info and book it. When you click "Book this spot", the app will prompt for number of hours – enter a value (say 1 or 2) – then it will show a confirmation with the cost. If you accept, it will simulate payment. You should see an alert that payment was successful. The marker for that spot will disappear (as it's now booked/unavailable). You can open the browser console or server log to see the debug prints (e.g., the email notification placeholder).
7. **Inspect Data:** If curious, open the `parkping.db` file (using SQLite Browser or command-line sqlite3). You'll find the tables updated with your user, the booking, etc. This can help you understand the state changes as you use the app.

Development tips: This project is a starting point. You can modify the front-end HTML/JS to change the look or add new UI elements (Bootstrap's components make it easy to add things like dialogs or cards). On the back-end, you can add new routes or extend existing ones. The code is meant to be straightforward to read and extend by developers exploring the project.

Development Roadmap

This starter project covers the basics, but there are many opportunities to enhance it into a production-ready application:

- **Real-Time Updates:** Currently, the client would have to poll the server to know if new spots become available. Implementing WebSocket support (using Flask-SocketIO, for example) or Server-Sent Events could allow the server to push notifications to the client when, say, a parking spot becomes free. This would truly enable real-time parking alerts (for instance, notify users "Spot #5 is now available!" without them refreshing). Alternatively, integrating a messaging/queue system for background tasks (e.g., to automatically free up spots whose time has expired) could be useful.

- **Routing & Directions:** To fully realize the "source-to-destination" flow like Uber, integrate the Google Directions API. After the user chooses a parking spot to book, the app could show directions from the user's current location to that parking spot, or from the parking spot to the final destination. This could be done by calling the Directions service and displaying the route on the map. (Google Maps Platform allows drawing polylines for routes, etc.) The current UI already gathers a destination; using that, the app could also suggest the closest parking spot to that destination in the future.

- **Filtering Spots:** If this were deployed in a large area, you'd want to fetch only nearby spots (to save bandwidth and declutter the map). You could enhance `GET /parking/spots` to accept parameters like `?near_lat=…&near_lng=…&radius=…` to return only spots within a certain distance. The front-end could send the user's location or destination to filter spots. This would be an easy extension to the current design.

- **Payment Integration:** Replace the mock payment with a real integration. For example, use Stripe Checkout or their API to process payments securely. Upon booking, you might redirect the user to a payment page, or integrate a credit card form in a modal. Once payment is confirmed by the gateway, update the booking as paid. This involves securely handling payment tokens and perhaps storing transaction IDs in the database.

- **UI/UX Improvements:** Use Bootstrap components to refine the user experience:

  - Implement **Bootstrap Toasts** for notifications instead of alert boxes (non-blocking and can auto-dismiss)getbootstrap.com.

  - Add a **loading spinner** or disable the booking button while a request is in progress to avoid duplicate actions.

  - Use **Bootstrap forms** for login/register with validation feedback. Currently, basic HTML5 required fields are used, but custom validation messages or stronger rules could be added.

  - Possibly integrate a front-end framework (React, Vue) if the project grows, but for a simple start Bootstrap + jQuery/JS is sufficient.

- **Email Notifications:** Build out the email notification system. For example, send an email when a booking is confirmed (the placeholder is there in the code). You could use Flask-Mail extension or external services (SendGrid, etc.). Similarly, if a spot becomes available, users who opted in could get an email or SMS. These features require background job processing in a real app (to avoid delaying the response while sending emails).

- **Microservice Separation:** Right now the app runs as one Flask service. In a microservices architecture, you might split it into separate services (e.g., an Auth service, a Parking service, a Payment service) that communicate via APIs. Given the current scale, this is not necessary, but the code structure (with clear separation of concerns) makes it easier to break apart if needed. For instance, you could separate `auth`, `parking`, and `payment` into different Flask apps and have a gateway or client orchestrate calls between them.

- **Security Enhancements:**

  - Use HTTPS in deployment (critical for protecting logins and enabling geolocation in production).

  - Implement rate limiting on API endpoints to prevent abuse (Flask-Limiter can help).

  - Use Flask's CSRF protection if you introduce forms that are served by Flask (for pure JSON API, not as relevant).

  - Possibly implement JSON Web Tokens (JWT) if you want to decouple the front-end from server sessions (e.g., if turning into a mobile app back-end).

  - Improve password handling: require strong passwords, allow password reset via email, etc.

- **Testing & Quality:** Add unit tests or integration tests for the API endpoints. Use tools like Postman or automated test frameworks (pytest) to test booking logic, etc. This ensures future changes won't break existing functionality.

- **Performance:** For a small app, Flask + SQLite is fine. As usage grows, you might move to a more robust database (PostgreSQL/MySQL) and possibly a WSGI server or ASGI server for concurrency. Also, caching frequently requested data (like parking spots that don't change often) could be considered to reduce DB hits.

With these enhancements, ParkPing can evolve from a basic starter project to a production-grade service. The current codebase is intentionally kept straightforward and commented for ease of understanding, so new developers can quickly grasp where to add new features or adjust behaviordigitalocean.com.

Project Files

Below is the complete project code structure with core files. Developers can use this as a reference or starting point for extension:

`backend/app.py`

```
python
from flask import Flask, request, session, redirect, jsonify, g, render_template
import sqlite3
from datetime import datetime, timedelta
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__, template_folder='../frontend/templates', static_folder='../frontend/static')
app.config['SECRET_KEY'] = 'dev-secret-key'  # Use a secure key in production

# Database configuration
DATABASE = 'parkping.db'

def get_db():
    """Opens a new database connection if there is none yet for the current application context."""
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
        db.row_factory = sqlite3.Row  # to retrieve rows as dict-like objects
    return db

@app.teardown_appcontext
def close_connection(exception):
    """Closes the database again at the end of the request."""
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()

def init_db():
    """Initialize the database with required tables if not present."""
    con = sqlite3.connect(DATABASE)
    cur = con.cursor()
    # Create users table
    cur.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            email TEXT UNIQUE NOT NULL,
            password TEXT NOT NULL
        )
    ''')
    # Create parking_spots table
    cur.execute('''
        CREATE TABLE IF NOT EXISTS parking_spots (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            address TEXT,
            lat REAL,
            lng REAL,
            rate REAL,
            is_available INTEGER
        )
    ''')
    # Create bookings table
    cur.execute('''
        CREATE TABLE IF NOT EXISTS bookings (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER,
            spot_id INTEGER,
            start_time TEXT,
            end_time TEXT,
            cost REAL,
            paid INTEGER,
            FOREIGN KEY(user_id) REFERENCES users(id),
            FOREIGN KEY(spot_id) REFERENCES parking_spots(id)
        )
    ''')
    con.commit()
    con.close()

# Initialize database at startup
init_db()

# Define authentication routes
@app.route('/api/auth/register', methods=['POST'])
def register():
    data = request.get_json()
    if not data or not data.get('email') or not data.get('password'):
        return jsonify({'error': 'Email and password are required'}), 400
    email = data['email']
    password = data['password']
    hashed_pw = generate_password_hash(password)
    try:
        db = get_db()
        cur = db.cursor()
        cur.execute("INSERT INTO users (email, password) VALUES (?, ?)", (email, hashed_pw))
        db.commit()
```

```
    except sqlite3.IntegrityError:
        return jsonify({'error': 'User already exists'}), 400
    return jsonify({'message': 'User registered successfully'}), 201

@app.route('/api/auth/login', methods=['POST'])
def login():
    data = request.get_json()
    if not data or not data.get('email') or not data.get('password'):
        return jsonify({'error': 'Email and password are required'}), 400
    email = data['email']
    password = data['password']
    db = get_db()
    cur = db.cursor()
    cur.execute("SELECT id, password FROM users WHERE email = ?", (email,))
    row = cur.fetchone()
    if row is None:
        return jsonify({'error': 'Invalid credentials'}), 401
    user_id = row['id']
    stored_password = row['password']
    if not check_password_hash(stored_password, password):
        return jsonify({'error': 'Invalid credentials'}), 401
    # Credentials valid, set session
    session['user_id'] = user_id
    session['user_email'] = email
    return jsonify({'message': 'Login successful'}), 200

@app.route('/api/auth/logout', methods=['POST'])
def logout():
    session.clear()
    return jsonify({'message': 'Logged out'}), 200

# Define parking routes
@app.route('/api/parking/spots', methods=['GET'])
def get_spots():
    if 'user_id' not in session:
        return jsonify({'error': 'Unauthorized'}), 401
    db = get_db()
    cur = db.cursor()
    cur.execute("SELECT id, address, lat, lng, rate FROM parking_spots WHERE is_available = 1")
    rows = cur.fetchall()
    spots = []
    for row in rows:
        spots.append({
            'id': row['id'],
            'address': row['address'],
            'lat': row['lat'],
            'lng': row['lng'],
            'rate': row['rate']
        })
    return jsonify({'spots': spots}), 200

@app.route('/api/parking/book', methods=['POST'])
def book_spot():
    if 'user_id' not in session:
        return jsonify({'error': 'Unauthorized'}), 401
    data = request.get_json()
    if not data or 'spot_id' not in data or 'hours' not in data:
        return jsonify({'error': 'spot_id and hours are required'}), 400
    spot_id = data['spot_id']
    try:
        hours = float(data['hours'])
    except:
        return jsonify({'error': 'Invalid hours value'}), 400
    if hours <= 0:
        return jsonify({'error': 'Hours must be greater than 0'}), 400
    user_id = session['user_id']
    db = get_db()
    cur = db.cursor()
    # Check spot availability
    cur.execute("SELECT rate, is_available FROM parking_spots WHERE id = ?", (spot_id,))
    spot = cur.fetchone()
    if spot is None:
        return jsonify({'error': 'Parking spot not found'}), 404
    if spot['is_available'] == 0:
        return jsonify({'error': 'Parking spot is already booked'}), 400
    rate = spot['rate']
    cost = rate * hours
    # Reserve the spot (mark as unavailable)
    cur.execute("UPDATE parking_spots SET is_available = 0 WHERE id = ?", (spot_id,))
    # Create a booking record
    start_time = datetime.now()
    end_time = start_time + timedelta(hours=hours)
    cur.execute(
        "INSERT INTO bookings (user_id, spot_id, start_time, end_time, cost, paid) VALUES (?, ?, ?, ?, ?, ?)",
```

```
        (user_id, spot_id, start_time.strftime("%Y-%m-%d %H:%M:%S"), end_time.strftime("%Y-%m-%d %H:%M:%S"), cost, 0)
    )
    booking_id = cur.lastrowid
    db.commit()
    # Placeholder for sending notification (e.g., email)
    user_email = session.get('user_email')
    if user_email:
        print(f"[Notification] Parking spot {spot_id} booked by {user_email}. Confirmation email sent.")
    return jsonify({'message': 'Spot booked successfully', 'booking_id': booking_id, 'cost': cost}), 200

# Define payment route
@app.route('/api/payment/complete', methods=['POST'])
def complete_payment():
    if 'user_id' not in session:
        return jsonify({'error': 'Unauthorized'}), 401
    data = request.get_json()
    if not data or 'booking_id' not in data:
        return jsonify({'error': 'booking_id is required'}), 400
    booking_id = data['booking_id']
    db = get_db()
    cur = db.cursor()
    cur.execute("SELECT cost, paid, spot_id FROM bookings WHERE id = ? AND user_id = ?", (booking_id, session['user_id']))
    booking = cur.fetchone()
    if booking is None:
        return jsonify({'error': 'Booking not found'}), 404
    if booking['paid'] == 1:
        return jsonify({'error': 'Payment already completed for this booking'}), 400
    # Process payment (mock)
    cur.execute("UPDATE bookings SET paid = 1 WHERE id = ?", (booking_id,))
    db.commit()
    # In a real app, you'd integrate with a payment gateway here
    cost = booking['cost']
    return jsonify({'message': 'Payment completed successfully', 'cost': cost}), 200

# Frontend routes for pages
@app.route('/')
def index_page():
    # If user is logged in, show main app, otherwise show login
    if 'user_id' in session:
        return render_template('index.html')
    else:
        return render_template('login.html')

@app.route('/login')
def login_page():
    # If already logged in, go to main page
    if 'user_id' in session:
        return redirect('/')
    return render_template('login.html')

@app.route('/register')
def register_page():
    if 'user_id' in session:
        return redirect('/')
    return render_template('register.html')

# Seed some sample parking data if none exists (for demo purposes)
with app.app_context():
    db = get_db()
    cur = db.cursor()
    cur.execute("SELECT COUNT(*) as cnt FROM parking_spots")
    cnt = cur.fetchone()['cnt']
    if cnt == 0:
        sample_spots = [
            ("Downtown Parking Lot 1", 51.046, -114.065, 2.5, 1),
            ("Downtown Parking Lot 2", 51.045, -114.066, 3.0, 1),
            ("Mall Parking Garage", 51.050, -114.070, 1.5, 1)
        ]
        for address, lat, lng, rate, avail in sample_spots:
            cur.execute("INSERT INTO parking_spots (address, lat, lng, rate, is_available) VALUES (?, ?, ?, ?, ?)",
                        (address, lat, lng, rate, avail))
        db.commit()

if __name__ == '__main__':
    app.run(debug=True)
```

frontend/templates/index.html

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
```

```
    <title>ParkPing - Find Parking</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
    <link rel="stylesheet" href="/static/style.css">
  </head>
  <body>
    <!-- Navigation Bar -->
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">ParkPing</a>
        <div class="d-flex">
          <button id="logoutBtn" class="btn btn-outline-light">Logout</button>
        </div>
      </div>
    </nav>
    <!-- Search bar -->
    <div class="container my-3">
      <div class="row">
        <div class="col-md-8">
          <input type="text" id="destination" class="form-control" placeholder="Enter destination address">
        </div>
        <div class="col-md-4">
          <button class="btn btn-primary w-100" id="searchBtn">Find Parking</button>
        </div>
      </div>
    </div>
    <!-- Map -->
    <div class="container-fluid px-0">
      <div id="map" style="height: 75vh; width: 100%;"></div>
    </div>

    <!-- Include JS at bottom for faster load -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
    <script src="/static/lib.js"></script>
    <!-- Google Maps API (replace YOUR_API_KEY with actual key) -->
    <script async defer src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initMap"></script>
  </body>
  </html>
```

frontend/templates/login.html

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>ParkPing - Login</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="/static/style.css">
</head>
<body class="d-flex align-items-center justify-content-center" style="min-height: 100vh;">
  <div class="container" style="max-width: 400px;">
    <h2 class="mb-4 text-center">Sign In to ParkPing</h2>
    <form id="loginForm">
      <div class="mb-3">
        <label for="email" class="form-label">Email address</label>
        <input type="email" id="email" class="form-control" required>
      </div>
      <div class="mb-3">
        <label for="password" class="form-label">Password</label>
        <input type="password" id="password" class="form-control" required>
      </div>
      <button type="submit" class="btn btn-primary w-100">Login</button>
      <p class="mt-3 text-center">No account? <a href="/register">Register here</a></p>
      <div id="loginError" class="text-danger mt-2 text-center"></div>
    </form>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
  <script src="/static/lib.js"></script>
</body>
</html>
```

frontend/templates/register.html

```
html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>ParkPing - Register</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="/static/style.css">
```

```
</head>
<body class="d-flex align-items-center justify-content-center" style="min-height: 100vh;">
  <div class="container" style="max-width: 400px;">
    <h2 class="mb-4 text-center">Create an Account</h2>
    <form id="registerForm">
      <div class="mb-3">
        <label for="regEmail" class="form-label">Email address</label>
        <input type="email" id="regEmail" class="form-control" required>
      </div>
      <div class="mb-3">
        <label for="regPassword" class="form-label">Password</label>
        <input type="password" id="regPassword" class="form-control" required>
      </div>
      <button type="submit" class="btn btn-success w-100">Sign Up</button>
      <p class="mt-3 text-center">Already have an account? <a href="/login">Sign in</a></p>
      <div id="registerError" class="text-danger mt-2 text-center"></div>
      <div id="registerSuccess" class="text-success mt-2 text-center"></div>
    </form>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
  <script src="/static/lib.js"></script>
</body>
</html>
```

frontend/static/lib.js

```
javascript
let map;
let userMarker;
let destMarker;
let spotMarkers = [];

// Initialize Google Map and load parking spots
function initMap() {
  // Default location (if geolocation not available) - using sample location (downtown of a city)
  const defaultLoc = { lat: 51.046, lng: -114.065 };
  map = new google.maps.Map(document.getElementById('map'), {
    center: defaultLoc,
    zoom: 14
  });
  // Try HTML5 geolocation to get user's current location
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(
      (position) => {
        const userLoc = { lat: position.coords.latitude, lng: position.coords.longitude };
        map.setCenter(userLoc);
        // Add marker for user's location
        userMarker = new google.maps.Marker({
          position: userLoc,
          map: map,
          title: "You are here",
          icon: {
            path: google.maps.SymbolPath.CIRCLE,
            scale: 8,
            fillColor: "#4285F4",
            fillOpacity: 1,
            strokeColor: "#ffffff",
            strokeWeight: 2
          }
        });
      },
      (error) => {
        console.warn('Geolocation failed or denied.', error);
      }
    );
  }
  // Load available parking spots from API
  loadSpots();
}

// Fetch spots from backend and display on map
function loadSpots() {
  fetch('/api/parking/spots')
    .then(response => response.json())
    .then(data => {
      if (data.error) {
        console.error("Error loading spots:", data.error);
        return;
      }
      // Remove existing spot markers
      spotMarkers.forEach(m => m.setMap(null));
      spotMarkers = [];
      data.spots.forEach(spot => {
        const pos = { lat: spot.lat, lng: spot.lng };
```

```
          const marker = new google.maps.Marker({
            position: pos,
            map: map,
            title: spot.address || "Parking Spot",
            // could use custom icon for parking spot here
          });
          // Info window content with a Book button
          const infoContent = `
            <div>
              <h6>${spot.address}</h6>
              <p>Rate: $${spot.rate.toFixed(2)}/hour</p>
              <button class="btn btn-sm btn-primary" onclick="bookSpot(${spot.id})">Book this spot</button>
            </div>`;
          const infowindow = new google.maps.InfoWindow();
          marker.addListener('click', () => {
            infowindow.setContent(infoContent);
            infowindow.open(map, marker);
          });
          spotMarkers.push(marker);
        });
      });
  }

  // Book a parking spot for a given number of hours
  function bookSpot(spotId) {
    const hrs = prompt("Enter number of hours to book:");
    if (hrs === null) return; // cancelled
    const hours = parseFloat(hrs);
    if (isNaN(hours) || hours <= 0) {
      alert("Please enter a valid number of hours.");
      return;
    }
    fetch('/api/parking/book', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ spot_id: spotId, hours: hours })
    })
    .then(response => response.json())
    .then(data => {
      if (data.error) {
        alert("Error: " + data.error);
        return;
      }
      const cost = data.cost;
      const bookingId = data.booking_id;
      // Confirm payment
      const payNow = confirm(`Spot booked for $${cost.toFixed(2)}. Proceed to payment?`);
      if (payNow) {
        fetch('/api/payment/complete', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ booking_id: bookingId })
        })
        .then(resp => resp.json())
        .then(payData => {
          if (payData.error) {
            alert("Payment error: " + payData.error);
          } else {
            alert("Payment successful! Enjoy your parking.");
          }
          // Refresh spots availability on map
          loadSpots();
        });
      } else {
        alert("Booking is reserved. Complete payment from your account when ready.");
        // Optionally, could keep spot reserved for some time before auto-release
      }
    });
  }

  // Handle user logout
  function logout() {
    fetch('/api/auth/logout', { method: 'POST' })
      .then(response => response.json())
      .then(data => {
        // On successful logout, redirect to login page
        window.location = '/login';
      });
  }

  // Handle destination search
  function searchDestination() {
    const address = document.getElementById('destination').value;
    if (!address) {
```

```
      alert("Please enter a destination address.");
      return;
    }
    const geocoder = new google.maps.Geocoder();
    geocoder.geocode({ address: address }, (results, status) => {
      if (status === "OK" && results[0]) {
        const loc = results[0].geometry.location;
        map.setCenter(loc);
        map.setZoom(15);
        // Mark the destination on the map
        if (destMarker) destMarker.setMap(null);
        destMarker = new google.maps.Marker({
          position: loc,
          map: map,
          title: "Destination",
          icon: {
            url: "http://maps.google.com/mapfiles/ms/icons/blue-dot.png"
          }
        });
        // In a real app, you might filter available spots near this location or show route
      } else {
        alert("Geocode was not successful: " + status);
      }
    });
  }

  // Setup event handlers on DOM load
  document.addEventListener('DOMContentLoaded', () => {
    // Login form submission
    const loginForm = document.getElementById('loginForm');
    if (loginForm) {
      loginForm.addEventListener('submit', (e) => {
        e.preventDefault();
        const email = document.getElementById('email').value;
        const password = document.getElementById('password').value;
        fetch('/api/auth/login', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ email: email, password: password })
        })
        .then(response => response.json())
        .then(data => {
          if (data.error) {
            document.getElementById('loginError').innerText = data.error;
          } else {
            // Redirect to main page on success
            window.location = '/';
          }
        });
      });
    }
    // Register form submission
    const registerForm = document.getElementById('registerForm');
    if (registerForm) {
      registerForm.addEventListener('submit', (e) => {
        e.preventDefault();
        const email = document.getElementById('regEmail').value;
        const password = document.getElementById('regPassword').value;
        fetch('/api/auth/register', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ email: email, password: password })
        })
        .then(response => response.json())
        .then(data => {
          if (data.error) {
            document.getElementById('registerError').innerText = data.error;
            document.getElementById('registerSuccess').innerText = "";
          } else {
            document.getElementById('registerError').innerText = "";
            document.getElementById('registerSuccess').innerText = "Registration successful! Redirecting to login...";
            // Redirect to login page after a short delay
            setTimeout(() => {
              window.location = '/login';
            }, 2000);
          }
        });
      });
    }
    // Logout button
    const logoutBtn = document.getElementById('logoutBtn');
    if (logoutBtn) {
      logoutBtn.addEventListener('click', () => {
        logout();
```

```
      });
    }
    // Search button
    const searchBtn = document.getElementById('searchBtn');
    if (searchBtn) {
      searchBtn.addEventListener('click', () => {
        searchDestination();
      });
    }
  });
```

frontend/static/style.css

```
css
/* Custom styles for ParkPing */
#map {
  /* Ensure the map fills its container */
  width: 100%;
}
```