Campus Parking Organizer

Web Application Project

Project members:

Programmers:

- Bazar Jayp S

Documentation:

- Ubante Jay Marion Bristol

Project report:

Executive Summary

Campus Parking Organizer is a web app designed to help students, faculty, and staff easily find and reserve parking spots on campus. With limited parking and the constant challenge of finding available spaces, this app gives users the ability to check real-time parking availability and navigate to open spots quickly.

The app features real-time updates on available parking spaces, a map for easy navigation, and the ability to reserve spots in advance. The goal is to reduce time spent searching for parking and keep campus parking organized, making the process more efficient for everyone.

Implemented Features

1. **User Authentication**

   Secure Login and Logout:
   - Users must log in with valid credentials to access the system.
   - Sessions are managed securely to prevent unauthorized access.

   Password Hashing:
   - User passwords are hashed using industry-standard algorithms, ensuring that stored passwords are not kept in plain text.

2. **Role-Based Access Control (RBAC)**

   Role Definitions: Two primary roles:
   - **Admin:** Has full access to user management, bookings, and parking lot management.
   - **Regular User (Employee/Student):** Can view and book parking lots, but has limited access to administrative functions.

   Access Restrictions:
   - Admins can add, edit, or delete users.
   - Regular users can only manage their own profiles and make bookings.

3. **User Management**

   Admin-Only Features:

   - **User Registration:** Admins can create accounts for new users.
   - **View User List:** Admins can view a complete list of all users.
   - **User Deletion:** Admins can delete user accounts when necessary.

4. **Profile Management**

   Editable User Profiles:

   - Users can update their personal information, including email, phone number, and profile picture.

   Password Change:

   - Users can change their password through their profile settings.

   Profile Picture Upload:

   - Users can upload profile pictures, which are stored and displayed on their profiles.

5. **Parking Lot Management**

   Parking Lot Status:

   - Admins can update the status of parking lots (Available, Reserved, Occupied).

   Location Tracking:

   - Each parking lot entry records latitude and longitude coordinates for easy identification and navigation.

6. **Bookings System**

   Booking Reservations:

   - Users can reserve parking lots by selecting available slots.
   - Booking start and end times are recorded for each reservation.

   Client Linkage:

   - Bookings are linked to specific users via ClientID, ensuring accountability and traceability.

7. **Security Measures**

   Cross-Site Request Forgery (CSRF) Protection:

   - Forms are protected against CSRF attacks, ensuring that only legitimate users can submit data.

   Session Management:

   - User sessions expire after a period of inactivity, reducing the risk of session hijacking.

   Cache Control:

   - Sensitive data is prevented from being cached by browsers, enhancing security.

System Design Decisions and Architectural Choices

- **Framework and Language**:
  We used **Python** with **Flask** for its lightweight, modular design and ease of use. Flask allowed us to quickly develop and scale features while benefiting from its extensive documentation and community support.
- **Database**:
  We chose **MySQL** for its simplicity and seamless integration with the app. Its lightweight nature provides scalability and better handling of complex queries while remaining compatible with our application's requirements.
- **Security**:
  Security was a priority, so we implemented **password hashing** with a secure algorithm like bcrypt to protect stored passwords. **CSRF protection** was added to prevent unauthorized form submissions, and secure cookies ensured safe user sessions.
- **Frontend**:
  The frontend of the Access Control System leverages **HTML templates** for structure, **CSS** (Cascading Style Sheets) for responsive design, and **Bootstrap** for streamlined and consistent styling. **JavaScript** adds interactivity to enhance user experience.

  CSS and Bootstrap:
    - Bootstrap ensures the application is responsive and visually appealing across different screen sizes.
    - Custom CSS is applied to further tailor the design and layout.

  JavaScript Enhancements:
    - Interactive features such as dynamic maps and email auto-suggestions are implemented using JavaScript.
    - This approach ensures the frontend remains lightweight and intuitive, avoiding unnecessary complexity while maintaining functionality.

Security Measures Implemented

- **Password Hashing** - We used **password hashing** with a secure algorithm like bcrypt to make sure user passwords aren't stored in plain text. This helps protect user data if the system is ever compromised.

- **CSRF Protection** - We added **CSRF protection** to prevent attacks where malicious users could trick others into submitting forms they didn't intend to. This makes sure that all form submissions come from the right users.

- **Cache Control Headers** - We use **cache control headers** to stop sensitive data from being stored in the browser cache, which helps keep user information private and secure.

- **Role-Based Access Control (RBAC)** - We set up **role-based access control** using custom decorators to make sure users can only access the pages or features they're allowed to. This keeps unauthorized users out of restricted areas, like the admin panel.

Challenges Faced and Solutions Applied

- **Challenge: Integrating role-based access control**
  Solution: One of the major challenges was ensuring that different types of users, such as students, employees, and admins, only had access to the appropriate pages and features.
  We addressed this by implementing session-based access control in Flask. User roles are stored in session variables upon login, allowing the application to dynamically check the user's role before granting access to specific routes and pages. This ensures that, for example, only admin users can access the admin panel, while regular users can only access the parking reservation features relevant to them.

- **Challenge: Ensuring the login system was user-friendly**
  Solution: We wanted to make the login process as easy and intuitive as possible, especially for users who might struggle with entering their email correctly each time. To solve this, we added email autosuggestions, which helps speed up the process by suggesting previously entered emails. This feature not only saves users time but also reduces the chance of login errors caused by mistyped emails.

- **Challenge: Ensuring security and privacy of user data**
  Solution: Protecting user data, especially passwords and personal information, was a high priority. We implemented **password hashing** using a secure algorithm to prevent plain-text storage of passwords. Additionally, session security features were added, including **secure cookies** and **CSRF protection** to mitigate unauthorized access and ensure data privacy throughout the user's interactions with the application.

Future Improvements and Recommendations

- Add **Two-Factor Authentication (2FA)** for extra security.
- Implement a **password reset feature** via email.
- Consider modernizing the frontend with frameworks like **React** or **Vue.js** for a more dynamic interface.

Technical Documentation

System Architecture Diagram

The app follows a **Flask MVC** (Model-View-Controller) structure, which is a common way to organize web applications. The main parts are:

- **Models**:
  These define how our data is structured and how different pieces of the app (like users and reservations) interact with the database. It ensures everything is organized and stored correctly.
- **Views/Routes**:
  These are responsible for handling user requests and deciding what to show them. They process information and render HTML pages. For example, when someone checks available parking, a route sends them the updated data.
- **Templates**:
  These are the HTML pages that get filled with dynamic content. They use **Jinja2**, a template engine that lets us insert Python variables directly into the HTML, like showing available parking spots on the map.
- **Static Files**:
  This folder holds things like **CSS**, **JavaScript**, and **images** that help style and make the app interactive. Static files are separated from the dynamic content to help with performance.

Database Schema

The app uses **MySQL** for the database. It's simple and works well with Flask. The main tables in the database are:

- **Users**:
  This table stores user information such as name, email, role, and profile details. student, faculty, admin.
  - Columns:
    - **ID:** A unique identifier for each user.
    - **FirstName:** The user's first name.
    - **MiddleName:** The user's middle name (optional).
    - **LastName:** The user's last name.
    - **Role:** Specifies whether the user is an admin, employee, or student.
    - **CspcEmail:** The user's organizational email.
    - **PhoneNumber:** Contact number of the user.
    - **Password:** Hashed password for secure storage.
    - **ProfilePhoto:** Path to the user's profile picture.

- **Parking lots:**
  This table tracks parking lot locations and availability status.
  - **Columns:**
    - **name:** Name of the parking lot.
    - **latitude1 & longitude1:** Coordinates for one boundary point.
    - **latitude2 & longitude2:** Coordinates for the opposite boundary point.
    - **status:** Indicates if the lot is available, reserved, or occupied.
- **Bookings:**
  This table records parking lot reservations and user bookings.
  - **Columns:**
    - **lotName:** The name of the parking lot being booked.
    - **bookingDate:** The date of the reservation.
    - **timeStarted:** Start time of the booking.
    - **timeEnded:** End time of the booking.
    - **ClientID:** Links the booking to a specific user (foreign key from users table).

Third-Party Libraries and Tools Used

We used several third-party tools to make the app work smoothly:

- **Flask**:
  This is the main framework we used to build the app. It's lightweight and easy to use, and it helped us set up routes and handle user requests.
- **Flask-Login**:
  This extension helps manage user logins and sessions. It makes sure users can log in and stay logged in while they navigate through the app.
- **mysql-connector-python:**
  This library allows the application to interact with the MySQL database directly using Python. It enables seamless database connections, execution of SQL queries, and retrieval of results without the need for an Object-Relational Mapper (ORM).
- **Hashlib:**
  We use **hashlib** for securely hashing user passwords. This ensures sensitive information is protected by hashing passwords before storage, making it difficult for unauthorized users to access plain-text credentials.
-

User Manual

1. Prerequisites:

- Python (3.7 or later)
- Flask and required dependencies (pip install -r requirements.txt)
- MySQL or SQLite database set up

2.Installaion steps:

- Clone the project repository.
- Set up a virtual environment

```
python -m venv venv
source venv/bin/activate (Linux/Mac)
venv/Scripts/activate (Windows)
```

- Install dependencies:

```
pip install -r requirements.txt
```

- Initialize the database:

```
flask db init
flask db migrate -m "Initial migration."
flask db upgrade
```

- Run the application:

```
flask run
```

It will be available at http://127.0.0.1:5000/

2. User Guide for Different Roles

- **Admin User:**
    1. **Login:** Just log in with your admin credentials.
    2. **Dashboard:** Once logged in, you'll have access to the control panel where you can manage users and monitor parking.
    3. **Manage Users:** Admins can add, remove, or update users and their roles.
    4. **Monitor Parking:** You can also see a full overview of parking usage and make adjustments if needed.

- **Regular User (Student/Faculty/Staff):**
    1. **Sign Up/Register:**
        1. On the homepage, click on the "Sign Up" button.
        2. Fill in your details (username, email, and password).
        3. Hit "Register" to create your account.
        4. You'll get a confirmation email (if we've set up email validation). Follow the link to activate your account.
    2. **Login:** Once your account is activated, log in using your email and password.
    3. **Check Parking Availability:** After logging in, you can check out available parking spaces on campus in real-time.
    4. **Reserve Parking:** Select an open spot and reserve it for a specific time.
    5. **Navigation:** Use the interactive map to find your reserved parking spot.

3. Troubleshooting Section

- **Problem: The app won't load after running `flask run`.**
    - **Solution:** Make sure all dependencies are installed by running:

    ```
    pip install -r requirements.txt
    ```

    - Check if the database is set up by running:

    ```
    flask db upgrade
    ```

- **Problem: Can't log in as a regular user.**
    - **Solution:** Check if the user exists in the database, and make sure the password is hashed correctly. If you're an admin, you can check the user details in the control panel.
- **Problem: Reservations aren't saving.**
    - **Solution:** Check the database for missing reservation data. Make sure your routes for making reservations are working properly.

If you're still having trouble, feel free to reach out to **[support_email]**.

Testing Documentation

Test Cases and Scenarios

Testing is a crucial part of ensuring the app works as expected. We've written several test cases to cover different features of the Campus Parking Organizer. Below are the main test cases and scenarios we used:

1. **Login System**
   - Test Case 1: Verify that a registered user can log in with a valid email and password.
   - Test Case 2: Verify that an unregistered user cannot log in and receives an error message.
   - Test Case 3: Check that the password field masks the password entered.
   - Test Case 4: Ensure the system suggests previously entered emails for faster login.
2. **Registration System**
   - Test Case 1: Verify that a new user can register with a valid username, email, and password.
   - Test Case 2: Ensure that a user cannot register with an already existing email.
   - Test Case 3: Ensure that the password field has a minimum length (e.g., 8 characters).
   - Test Case 4: Verify that the email entered is in a valid email format.
3. **Profile Management**
   - Test Case 1: Verify that users can update their profile details successfully.
   - Test Case 2: Ensure that users can change their password securely.
   - Test Case 3: Test that profile pictures can be uploaded and displayed correctly.
4. **Role-Based Access Control (RBAC)**
   - Test Case 1: Verify that only admin users can access the admin panel.
   - Test Case 2: Ensure regular users can only access the profile management and general features.
   - Test Case 3: Verify that trying to access restricted pages (as a regular user) results in a 403 Forbidden error.
5. **Database Operations**
   - Test Case 1: Verify that user registration data is saved correctly in MySQL.
   - Test Case 2: Ensure that deleted users are removed from the database.
   - Test Case 3: Test that admin actions (e.g., adding users) reflect accurately in the database.

Test Results and Coverage Report

After running all our tests, we ensured that the system works as intended. Below is a summary of the results:

- **Login System**: All test cases passed successfully. Users can log in, and error messages are shown for invalid attempts.
- **Registration System**: Test cases for registration, including email validation and password length, passed. Duplicate email registrations were prevented.

- **Profile Management**: Profile updates and password changes worked as expected. Profile pictures uploaded and displayed correctly.
- **Role-Based Access Control (RBAC)**: Security measures were correctly enforced. Admin access was restricted, and regular users only had access to their designated features.
- **Database Operations**: All database interactions were successful, with accurate updates to user data.

Coverage Report:

- We tested over 90% of the system's functionalities, focusing on core features like authentication, RBAC, and database operations.
- For future updates, we plan to add more edge-case tests to cover unexpected situations, like system failures or invalid data inputs.