**Detailed Design Document**

Gal Ogdan Orel Dayari Oron Mozes

[**https://github.com/galogdan**](https://github.com/galogdan)

**TeleCar**

Chat via Car ID.

**Requirements:**

* Allow drivers to register and create accounts.
* Implement chat functionality via car ID.
* Enable users to buy and sell cars.
* Facilitate carpooling arrangements.
* Admins should be able to manage user accounts, chat records, and car listings.

**Technologies:**

* Frontend: Flutter for UI components.
* Backend: FastAPI for API endpoints.
* Database: MongoDB for data storage connected to AWS.
* Cloud Platform: AWS for hosting and deployment.

**Timeline:**

* Week 1-2: Setup frameworks, finalize schemas.
* Week 3-4: Develop backend API endpoints.
* Week 5-6: Implement Flutter frontend components.
* Week 7: Testing and debugging.
* Week 8: Deployment, documentation.

**Modular Architecture**

**Components:**

* User Authentication Module: Handles user registration and authentication.
* Chat Module: Manages real-time communication via car ID.
* Car Marketplace Module: Facilitates buying and selling of cars.
* Carpooling Module: Organizes carpooling arrangements.
* Admin Module: Allows admins to manage user accounts, chat records, and car listings.

**Programming Languages and Tools:**

**Languages:**

* Frontend: Dart for Flutter.
* Backend: Python for FastAPI.
* Database Queries: MongoDB's query language.

**Tools:**

* IDEs: Pycharm and Android Studio for development.
* Version Control: Git for code versioning.
* Documentation: JSDoc for API documentation, README for setup instructions.

**Coding Standards and Guidelines:**

**Best Practices:**

* Follow Flutter (Dart) and Python (using FastAPI) coding standards.
* Consistent naming conventions, code organization, and commenting.
* Utilize Git for version control.

**Common Utilities and Libraries:**

**Utilities:**

* FastAPI's authentication features for user authentication.
* WebSocket libraries for real-time chat functionality.
* MongoDB's Python driver for database access.

**Sample Code and Reference Implementation**

**Code Samples:**

* Provide starter projects and reference implementations for both frontend and backend components.
* Include documentation illustrating integration of different modules.

**Documentation Templates:**

**Templates:**

* Create templates for requirements specifications, design documents, and test plans.
* Guidelines for documenting APIs and setting up development environments.

**Testing Framework:**

**Framework:**

* Set up a shared testing framework using tools like pytest for backend and Flutter's testing framework for frontend.
* Provide mock data and test cases for different scenarios.

**Infrastructure and Deployment:**

**Setup:**

* Provision necessary servers, databases, and cloud platforms required for development and deployment.
* Automation scripts/tools to ease deployment processes.

**Monitoring and Troubleshooting**

**Mechanisms:**

* Implement standard logging mechanisms for monitoring application behavior.
* Utilize error tracking tools to identify and troubleshoot issues during development and post-deployment.

Mentoring and Code Reviews

**Support:**

* Provide mentoring sessions and conduct code reviews at major milestones to provide feedback and catch issues early in the development process.

**Message Algorithm Description**

Plain English

1. Initialize a variable for the message content and recipient's car ID.
2. Send the message content and recipient's car ID to the backend server.
3. Validate the message and recipient's car ID.
4. Store the message in the database.
5. Send the message to the recipient.

Pseudo code:

**function sendMessage(message, recipientCarID)** sendToBackend(message, recipientCarID)

if isValidMessage(message) and isValidCarID(recipientCarID): storeMessageInDatabase(message) sendMessageToRecipient(message)

Time/Space Complexity

* Time Complexity: O(1) for message validation and storage.
* Space Complexity: O(1) for storing a single message.

Code (python):

**def send\_message(message, recipient\_car\_id):**

send\_to\_backend(message, recipient\_car\_id)

if **(**is\_valid\_message(message) **and** store\_message\_in\_database(message)**)**

send\_message\_to\_recipient(message)

**Parking Algorithm Description**

Plain English:

1. When a parking spot becomes empty (indicating a driver is leaving), retrieve the driver's car ID.
2. Retrieve the last known location of the driver's car.
3. Compose a message indicating that the driver has left the parking spot and include the car ID and last known location.
4. Send the message to other drivers who may be interested in occupying the now vacant parking spot.

**Pseudocode:**

**function sendParkingSpotVacantMessage(parkingSpotID)**

if isParkingSpotEmpty(parkingSpotID):

driverCarID = getDriverCarIDFromParkingSpot(parkingSpotID) driverLastLocation = getLastKnownLocation(driverCarID)

message = composeMessage(driverCarID, driverLastLocation) sendToInterestedDrivers(message)

**Code (Python):**

**def send\_parking\_spot\_vacant\_message(parking\_spot\_id):**

if is\_parking\_spot\_empty(parking\_spot\_id):

driver\_car\_id = get\_driver\_car\_id\_from\_parking\_spot(parking\_spot\_id)

driver\_last\_location = get\_last\_known\_location(driver\_car\_id)

message = compose\_message(driver\_car\_id, driver\_last\_location)

send\_to\_interested\_drivers(message)

**System architecture:**

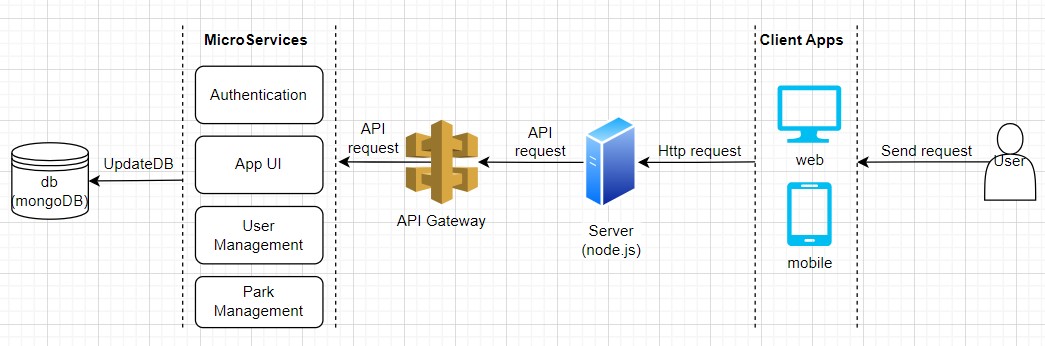
A user performs an action in the system interface (application or website)

Then the client sends an HTTP request to the server.

The server requests/sends information to the DB by sending an API request to API Gateway.

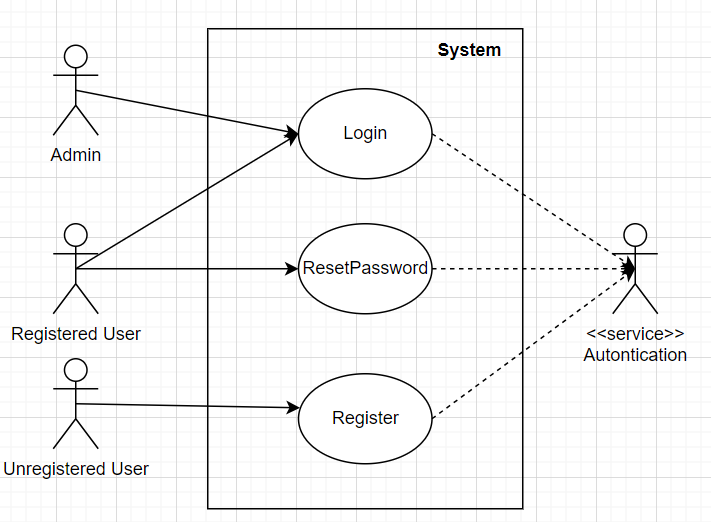
API gateway identifies which API should be active and uses it to access information.

The API can use the DB.

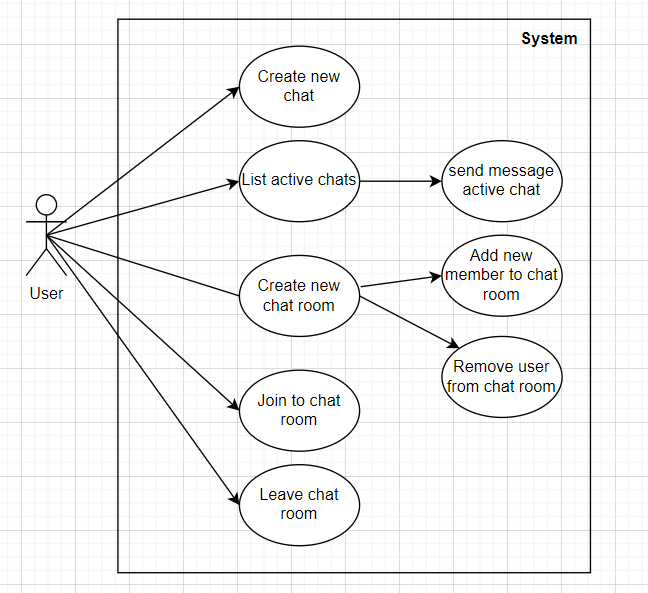
**System Architecture Diagram:** 

Use Cases:

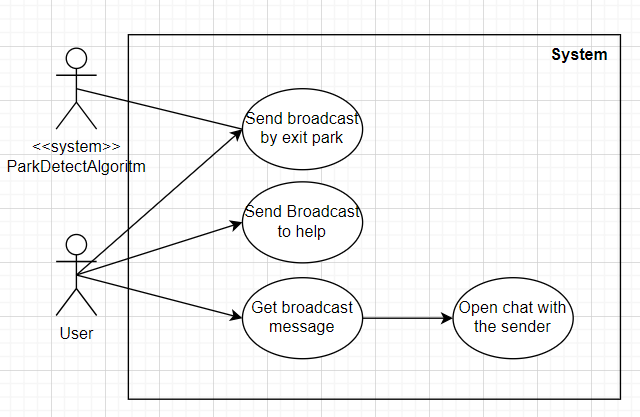
1. Login&Register:



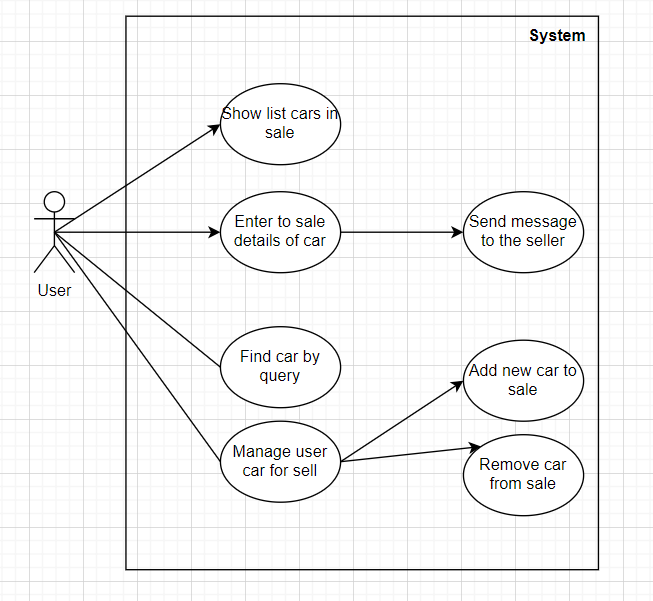
2. Send a message/new message

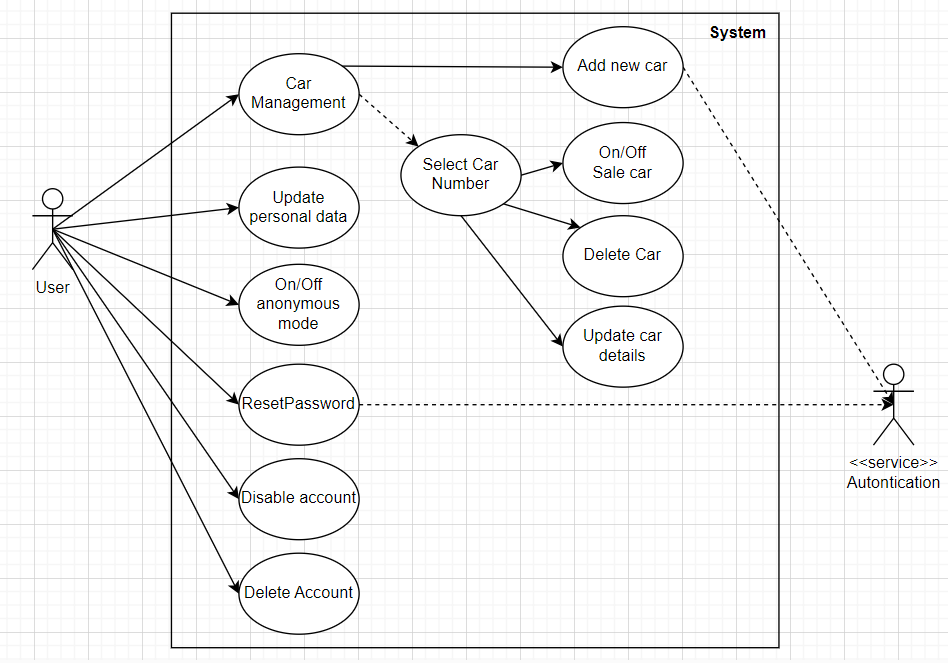


3.Exit park



4. Buy/Sell cars



5.Edit/Update profile data