

**REPUBLIC OF CAMEROON**

**Peace-Work-Fatherland**

**MINISTER OF HIGHER EDUCATION**

**FACULTY OF ENGINEERING**

**AND TECHNOLOGY**

**REPUBLIQUE DU CAMEROON**

**PAIX-Travail-Patrie**

**MINISTRE DE L’ENSEIGNEMENT SUPERIEUR**

**FACULTE D’INGINERIE**

**ET TECHGNOLOGIE**

**CEF440**

**INTERNET AND MOBILE PROGRAMMING**

**DESIGN OF TAXI APP**

**Presented by:**

**HEUYA NDJANSEB KEVIN KENNEDY FE20A049**

**YUVEN BRENDARD WIYFOFE FE20A124**

**INSTRUCTOR**: **Dr. NKEMENI VALERY**

**March 2023**

**Introduction:**

This report presents the design and implementation of “Ways” database that helps taxi drivers find passengers in a locality. The app will utilize a database to store information about drivers, passengers, and their locations. The database will support the core functionalities of the app such as the ability to match drivers with passengers in real-time and provide efficient routing and navigation for drivers.

**Advantages of Firebase for Ways Database**

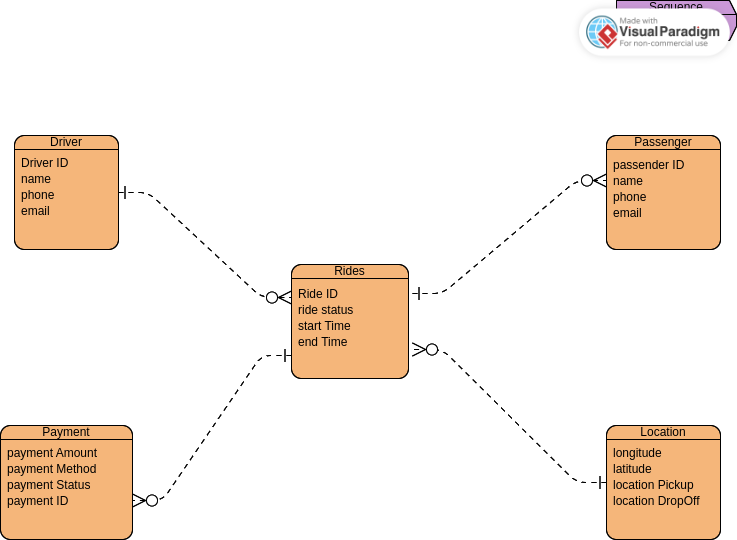
There are several advantages to using Firebase in the implementation of Way;

1. **Real-time updates:** Firebase Realtime Database allows for real-time updates to data, which is ideal for a taxi app where drivers and passengers are constantly on the move. This means that drivers and passengers can see updates to ride requests and location information in real-time, without the need to refresh the app.
2. **Scalability:** Firebase is a cloud-based platform that can scale to meet the needs of large or small applications. This means that the app can handle a large number of drivers and passengers, without the need for additional hardware or infrastructure.
3. **Security:** Firebase provides built-in security features such as authentication and access controls, which can help ensure that only authorized users can access the app's features and data.
4. **Ease of integration:** Firebase provides easy integration with other Google services, such as Google Maps and Google Cloud Messaging. This can help developers implement additional features and functionality quickly and easily.
5. **Analytics:** Firebase provides built-in analytics tools that can help developers track user behavior and app usage. This information can be used to optimize the app's performance and improve the user experience.
6. **Cost-effective:** Firebase offers a flexible pricing model that allows developers to pay for only the services they use. This can help keep costs down, especially for smaller apps with limited budgets.

Overall, Firebase provides a range of features and benefits that can help developers quickly and easily implement a mobile app that helps taxi drivers find passengers, while also providing a secure and scalable solution that can grow with the app's needs.

**Entity Relationship Diagram**

**(ERD):**



The ERD for the database implementation of the mobile app is presented below:

The ERD consists of the following entities:

1. **Driver:** This entity stores information about the registered drivers such as their name, phone number, email, and driver ID. The driver ID is the primary key for this entity.
2. **Passenger:** This entity stores information about the registered passengers such as their name, phone number, email, and passenger ID. The passenger ID is the primary key for this entity.
3. **Location:** This entity stores information about the location of the drivers and passengers. It includes attributes such as latitude, longitude, and location ID. The location ID is the primary key for this entity.
4. **Ride:** This entity stores information about the rides requested by passengers and accepted by drivers. It includes attributes such as ride ID, driver ID, passenger ID, pickup location ID, drop-off location ID, ride status, and ride start and end time. The ride ID is the primary key for this entity.
5. **Payment:** This entity stores information about the payment made by the passenger for each ride. It includes attributes such as payment ID, ride ID, payment amount, payment method, and payment status. The payment ID is the primary key for this entity.

Database

**Implementation:**

The database is implemented using MySQL, a popular open-source relational database management system. The database schema is created using SQL, and the database is populated with sample data for testing purposes.

The following is the SQL code used to create the database schema:

sql

CREATE DATABASE TaxiApp;

USE TaxiApp;

CREATE TABLE Driver (

DriverID INT PRIMARY KEY,

Name VARCHAR(255) NOT NULL,

Phone VARCHAR(20) NOT NULL,

Email VARCHAR(255) NOT NULL

);

CREATE TABLE Passenger (

PassengerID INT PRIMARY KEY,

Name VARCHAR(255) NOT NULL,

Phone VARCHAR(20) NOT NULL,

Email VARCHAR(255) NOT NULL

);

CREATE TABLE Location (

LocationID INT PRIMARY KEY,

Latitude DECIMAL(10, 8) NOT NULL,

Longitude DECIMAL(11, 8) NOT NULL

);

CREATE TABLE Ride (

RideID INT PRIMARY KEY,

DriverID INT NOT NULL,

PassengerID INT NOT NULL,

PickupLocationID INT NOT NULL,

DropoffLocationID INT NOT NULL,

RideStatus ENUM('requested', 'accepted', 'in progress', 'completed') NOT NULL,

StartTime DATETIME NOT NULL,

EndTime DATETIME,

FOREIGN KEY (DriverID) REFERENCES Driver(DriverID),

FOREIGN KEY (PassengerID) REFERENCES Passenger(PassengerID),

FOREIGN KEY (PickupLocationID) REFERENCES Location(LocationID),

FOREIGN KEY (DropoffLocationID) REFERENCES Location(LocationID)

);

CREATE TABLE Payment (

PaymentID INT PRIMARY KEY,

RideID INT NOT NULL,

PaymentAmount DECIMAL(10, 2) NOT NULL,

PaymentMethod ENUM('momo', 'bank card', 'cash') NOT NULL,

PaymentStatus ENUM('pending', 'completed', 'cancelled') NOT NULL,

FOREIGN KEY (RideID) REFERENCES Ride(RideID)

);

Conclusion:  
In conclusion, The ERD and database schema have been designed to support the core functionalities of the app such as real-time matching of drivers and passengers, efficient routing and navigation for drivers, and secure payment processing. The implementation has been done using MySQL, and the database has been populated with sample data for testing purposes.

The ERD presented in the report captures the relationships between the entities in the system. The Driver and Passenger entities represent the users of the app, while the Location entity stores the latitude and longitude coordinates of the drivers and passengers. The Ride entity captures the details of each ride requested by a passenger and accepted by a driver, including the pickup and drop-off locations, ride status, and ride start and end times. The Payment entity stores the information about the payment made by the passenger for each ride.

The ERD design ensures that the database is normalized, which prevents data redundancy and inconsistencies. For example, the Driver and Passenger entities have their own respective tables, which avoids duplicating the user's information for each ride they take. Similarly, the Location entity is separate from the Ride entity, which prevents the need to repeat the location information for each ride.

The database schema presented in the report is implemented using SQL, which is a standard language for relational databases. The schema includes the necessary data types, constraints, and foreign key relationships between the tables. For example, the Ride table has foreign key relationships with the Driver, Passenger, PickupLocation, and DropoffLocation tables, which ensures that only valid data is stored in the Ride table. Additionally, the Payment table has a foreign key relationship with the Ride table, which allows payments to be associated with their respective rides.

In practice, the database would be hosted on a server and accessed by the mobile app through an API (Application Programming Interface). The API would provide the necessary endpoints for the app to interact with the database, such as creating new rides, updating ride status, and processing payments. The database itself would be secured using appropriate measures such as encryption, access control, and data backups.

Overall, the database implementation presented in the report provides a solid foundation for the mobile app to help taxi drivers find passengers in a locality. The design ensures that data is organized efficiently and accurately, which supports the app's core functionalities and enables a seamless user experience.

**Measures taken to secure the database.**

Securing the database is crucial to protect the sensitive information stored in it. Here are some measures that can be taken to secure the database for the mobile app that helps taxi drivers find passengers in a locality:

1. **Encryption:** All sensitive information such as user passwords, payment details, and personal information should be encrypted when stored in the database. Encryption ensures that even if someone gains unauthorized access to the database, they won't be able to read the sensitive data.
2. **Access control:** Only authorized users should be able to access the database. This can be achieved by implementing user authentication, authorization, and access control mechanisms. User authentication ensures that only authorized users can log in to the app and access the database. Authorization determines what actions a user can perform within the app, while access control restricts access to certain parts of the database.
3. **Data backups:** Regular backups of the database should be taken to prevent data loss in case of a system failure or a security breach. The backups should be stored securely and off-site so that they can be easily recovered in case of a disaster.
4. **Regular updates and patches:** The database management system and all software used in the database environment should be regularly updated with the latest security patches and updates. This helps to prevent vulnerabilities that could be exploited by attackers.
5. **Auditing and logging:** The system should be set up to log all activities related to the database, including login attempts, data access, and modifications. This helps to track any unusual activity and detect potential security threats.
6. **Network security:** The database should be hosted on a secure network that is protected by firewalls, intrusion detection systems, and other security measures to prevent unauthorized access and data breaches.

By implementing these measures, the database for the mobile app that helps taxi drivers find passengers in a locality can be secured against potential security threats and data breaches.

**Ensuring that only authorized users can access the app.**

To ensure that only authorized users can access the mobile app, you can implement user authentication and authorization mechanisms. Here are some common approaches to user authentication and authorization:

1. **Password-based authentication:** This is the most common form of authentication, where users are required to enter a username and password to access the app. To ensure that passwords are secure, you can enforce password requirements such as length, complexity, and expiration. Additionally, you can use password hashing and salting to protect passwords from being compromised.
2. **Two-factor authentication (2FA):** This adds an extra layer of security to password-based authentication by requiring users to provide a second form of authentication, such as a fingerprint scan, a one-time code sent via SMS or email, or a security token. 2FA makes it harder for attackers to gain unauthorized access even if they have stolen a user's password.
3. **Social media authentication:** This allows users to log in to the app using their social media accounts such as Facebook, Twitter, or Google. Social media authentication is convenient for users and can simplify the registration process.

Once users are authenticated, you can implement authorization mechanisms to determine what actions they can perform within the app. Authorization can be based on user roles, permissions, or access levels. For example, a driver may have access to driver-specific features such as accepting rides, while a passenger may have access to passenger-specific features such as requesting rides and making payments.

To ensure that user authentication and authorization are implemented securely, you should follow best practices such as using secure protocols (e.g., HTTPS), enforcing strong password policies, and regularly reviewing and updating access control policies. Additionally, you should implement logging and auditing to keep track of user activity and detect any potential security breaches.

**helps drivers find passengers in a specific locality.**

One important consideration is the accuracy and currency of the location information stored in the database. Since the app relies on location data to match drivers with passengers, it's crucial to ensure that the location information in the database is accurate and up to date. This can be achieved by using reliable location data sources and regularly updating the location information in the database.

Another consideration is the ability to handle real-time updates to driver locations and passenger requests. Since the app will be used in real-time, the database must be able to handle fast and frequent updates to driver locations and passenger requests. This can be achieved by using appropriate indexing and caching techniques to optimize the database's performance.

Finally, it's essential to ensure that the database schema is flexible enough to accommodate future changes and updates to the app's functionality. As the app evolves and new features are added, the database schema may need to be modified to support new data structures and relationships. By designing a flexible and scalable database schema, developers can ensure that the app can be easily updated and maintained over time.

Overall, designing a database for a taxi app that helps drivers find passengers in a specific locality is a complex process that requires careful consideration of many factors. However, with the right design and implementation, the database can provide a solid foundation for the app to provide a smooth and reliable experience for drivers and passengers alike.

**Ability to handle complex queries and data analysis.**

Since the app will be used by many drivers and passengers at the same time, it's essential to ensure that the database can handle complex queries and data analysis in real-time. For example, the app may need to perform queries to find the nearest available driver to a passenger's location or analyze ride data to identify trends and patterns.

To achieve this, the database can be optimized by using appropriate indexing techniques and implementing efficient query algorithms. Additionally, it may be necessary to use data warehousing and business intelligence tools to perform complex data analysis and provide insights into the app's performance and usage.

Another consideration is the integration of third-party services and APIs. Many taxi apps rely on third-party services and APIs to provide additional functionality, such as payment processing or real-time traffic updates. To integrate these services into the app, the database schema must be designed to support the necessary data structures and relationships.

Finally, it's important to consider the scalability and availability of the database. As the app grows in popularity and usage, the database must be able to handle increasing volumes of data and transactions. This can be achieved by using scalable database solutions, such as cloud-based databases, and implementing appropriate backup and recovery procedures to ensure the availability of data in the event of a failure.

In conclusion, designing a database for a taxi app that helps drivers find passengers in a specific locality requires careful consideration of many factors, including data accuracy, real-time updates, complex queries, third-party integration, scalability, and availability. By designing a flexible, scalable, and efficient database schema, developers can ensure that the app provides a seamless and reliable experience for drivers and passengers alike, even during peak usage periods.

**Data privacy and security.**

Since the app will be handling sensitive information such as personal details, payment information, and location data, it's important to ensure that the database is secure and protected from unauthorized access. This can be achieved by implementing proper access controls, encryption, and other security measures.

For example, the database can be designed to store sensitive data separately from other data, with restricted access only to authorized personnel. Encryption can be used to protect sensitive data both in transit and at rest, and data backups should be encrypted as well.

Additionally, the app can be designed to respect user privacy, such as by providing options to disable location tracking or delete personal data.

Another consideration is compliance with data protection regulations, such as the General Data Protection Regulation (GDPR) or the California Consumer Privacy Act (CCPA). These regulations impose strict requirements on the handling of personal data, including data storage, processing, and disclosure. By designing the database with data protection regulations in mind, developers can ensure that the app is compliant with these regulations and avoid potential legal issues.

Finally, it's important to ensure that the database is regularly audited and monitored for potential security breaches. This can be achieved by implementing appropriate logging and monitoring tools and conducting regular security audits to identify potential vulnerabilities and address them promptly.

In conclusion, designing a database for a taxi app that helps drivers find passengers in a specific locality requires careful consideration of data privacy and security. By implementing appropriate security measures, complying with data protection regulations, and regularly auditing the database for potential security breaches, developers can ensure that the app provides a secure and reliable experience for drivers and passengers alike.

**handle user feedback and ratings.**

Since the app relies on the trust and reputation of drivers and passengers, it's important to provide a mechanism for users to provide feedback and ratings on their experiences. This can be achieved by adding a feedback and rating system to the app and storing the feedback and ratings in the database.

The feedback and rating system can be designed to allow users to rate drivers and passengers based on factors such as professionalism, punctuality, and friendliness. Users can also provide written feedback to provide more detailed information on their experiences.

To store the feedback and ratings in the database, a new table can be created to store this information. This table can have a foreign key to the rides table to link the feedback and ratings to specific rides. It can also have fields to store the user ID, the rating, and the written feedback.

The feedback and rating system can be used to improve the quality of the app's services by identifying problematic drivers or passengers and taking appropriate action. For example, drivers with consistently low ratings may be suspended or removed from the app, while passengers with a history of poor behavior may be banned from using the app.

In addition to improving the quality of the app's services, the feedback and rating system can also provide valuable insights into user behavior and preferences. By analyzing the feedback and ratings data, developers can identify trends and patterns in user behavior and use this information to improve the app's functionality and user experience.

In conclusion, designing a database for a taxi app that helps drivers find passengers in a specific locality requires careful consideration of many factors, including the ability to handle user feedback and ratings. By implementing a feedback and rating system and storing the data in the database, developers can improve the quality of the app's services and gain valuable insights into user behavior and preferences.

Share

**Handle payments and transactions.**

Since the app will be used to facilitate transactions between drivers and passengers, it's important to have a secure and reliable payment system. This can be achieved by integrating with a payment gateway or implementing a payment system within the app.

To store payment and transaction information in the database, a new table can be created to store this information. This table can have a foreign key to the rides table to link the payments to specific rides. It can also have fields to store the payment amount, payment method, payment status, and other relevant information.

The payment system can be designed to support multiple payment methods, such as credit cards, debit cards, and digital wallets. It can also be designed to handle complex transactions, such as splitting a fare between multiple passengers.

To ensure the security of the payment system, it's important to implement appropriate encryption and security measures. For example, sensitive payment information can be encrypted both in transit and at rest, and access to the payment system can be restricted only to authorized personnel.

In addition to handling payments and transactions, the database can also be used to provide valuable insights into the app's financial performance. By analyzing payment data, developers can identify trends and patterns in revenue and use this information to optimize the app's pricing and revenue strategies.

In conclusion, designing a database for a taxi app that helps drivers find passengers in a specific locality requires careful consideration of payment and transaction processing. By implementing a secure and reliable payment system and storing payment and transaction data in the database, developers can ensure the app provides a smooth and seamless experience for drivers and passengers alike while also providing valuable insights into the app's financial performance.