Software Requirements Specification
Taxi Tap by Git It Done

Contents

1 Introduction

Taxi Tap is a mobile platform designed to revolutionize South Africa's minibus taxi industry by digitizing route information, eliminating the need for constant hooting, and creating a semi-structured booking system while preserving the flexibility that makes taxis an essential mode of transport. The system connects passengers and taxi operators through a location-aware mobile application that facilitates taxi requests, communicates passenger locations, manages payments, and provides real-time vehicle tracking – all without fundamentally changing the existing system's multi-passenger, flexible route nature.

2 User Characteristics

The users of the Taxi Tap system are expected to fit into the following groups:

2.1 Driver User Characteristics

Attribute	**Description**
Familiarity with Mobile Technology	Varies widely; some drivers may be tech-comfortable, while others may struggle with apps.
Access to Reliable Internet and Data	Often limited or inconsistent; drivers operate in areas with poor signal or expensive data costs.
Preferred Language and Communication Style	Preference for local languages (e.g. Zulu, Xhosa, Sesotho).
Attention Capacity While Driving	App must require minimal taps and distractions to ensure safe usage while driving.
Trust and Skepticism Toward New Technology	Some skepticism exists due to concerns about surveillance, manipulation, or job security.
Goals and Incentives for Using the App	Seeking more passengers, quicker pickups, and less idle time, while maintaining their routine.

Table 1: Driver User Characteristics and Considerations

2.2 Passenger User Characteristics

Attribute	Description
Digital Literacy	Ranges from students and workers (tech-savvy) to commuters with limited app experience.
Access to Reliable Internet and Data	Frequently encounters low or no connectivity, especially in transit.
Reasons for Using the Platform	Seeks reliable transport, less waiting, and a safer way to locate and use taxis.
Preferred Language and Communication Style	May prefer local languages (e.g. Zulu, Xhosa, Sesotho).
App Usage Context (Where & When)	Often uses the app in crowded, noisy, or busy settings like taxi ranks.
Concerns Around Trust and Safety	Wants to be sure drivers are legitimate and that their location and personal data are protected.
Platform Interaction Needs	Needs to discover taxis, request rides, track driver arrival, and receive ride notifications.

Table 2: Commuter User Characteristics and Considerations

3 User Stories

3.1 Passenger User Stories

User Story	Acceptance Criteria	Definition of Done
Account Registration & Login	As a passenger, I want to sign up and log in to my account, so that I can securely access and use the Taxi Tap app.	Given that I am on the app's welcome screen, When I choose "Sign Up" or "Log In" and enter valid details, then I should be authenticated and taken to the home screen. Based on my input criteria, I am taken to the home page of Taxi Tap.
View Available Taxis and Routes	As a passenger, I want to view available taxis and their routes on a map, so that I can choose one that matches my travel needs.	Given I am logged in, When I open the home screen, then I should see nearby taxis on a map with route or destination labels. The map displays icons of nearby taxis, including route or destination tags, when available.

Set Pickup and Destination	As a passenger, I want to share my location and set a destination, so that drivers can find me and pick me up efficiently.	Given that I have granted location access, when I enter or select a pickup and destination point, then the app should confirm my trip details and show nearby taxis. Pickup and destination are confirmed and displayed; nearby taxis are suggested based on the selected route.
Book a Seat and Get Confirmation	As a passenger, I want to book a seat on a taxi and receive confirmation, so that I'm guaranteed a spot before the taxi arrives.	Given I've selected a taxi, When I tap "Book Seat" and confirm, Then I should receive a booking confirmation and a ride status update. A booking confirmation message appears with the selected taxi details and current ride status.
Track Assigned Taxi in Real-Time	As a passenger, I want to track my assigned taxi in real time, so that I know when and where to expect pickup.	Given my booking is confirmed, When I open the tracking screen, then I should see the taxi's live location and estimated time of arrival. The assigned taxi is visible on the map with a live location marker and updated ETA.
Receiving Alerts When Taxi is Nearby	As a passenger, I want to receive alerts when the taxi is nearby, so that I can be ready at the pickup location.	Given the assigned taxi is approaching, When it is within 500 meters, then I should receive a push notification that it's nearby. A push alert is triggered and received once the taxi enters the defined proximity radius.
See Available Seats	As a passenger, I want to see how many seats are available, so that I can decide whether to book a seat or wait.	Given I view a taxi on the map or booking screen, When I open its details, then I should see the number of available seats.

		The number of available seats is clearly shown for each listed or selected taxi.
Rate Completed Trip	As a passenger, I want to rate my trip after completion, so that I can provide feedback to help improve the service.	Given my ride has ended, When I open the app, then I should be prompted to leave a 1–5 star rating and optional comments. The rating form appears automatically after the ride ends, and feedback is successfully submitted to the system.
Use App Offline or on Low Bandwidth	As a passenger, I want to use the app offline or on low bandwidth, so that I can still interact with core features in areas with poor connectivity.	Given I have limited internet access, When I open the app, then I should still be able to view saved routes, taxis, and queue a ride request that sends once reconnected. The app functions with cached map data and stores ride requests locally, syncing once connectivity is restored.
Indicate Drop-Off Point Mid-Trip	As a passenger, I should be able to indicate where I want to be dropped off during the trip.	Given that I'm in a running taxi, when I choose a stop or drop-off location mid-trip, then the driver should receive an update of my chosen point. The app allows drop-off selection, sends update to driver, and displays new estimated drop-off.
View All Available Stops	As a passenger, I should be able to see all the available stops that are there during my trip.	Given I'm on a selected route, when I view route details, then I should see a list or map of all possible stops. A stop list or map view is shown, detailing all available stops along the route.
See Estimated Time to Destination	As a passenger, I should be able to see how long it will take to reach my destination or drop-off spot.	Given I've set a destination, when I view trip details, then I should see the estimated time remaining.

		ETA is shown dynamically on screen and is updated with real-time traffic and route changes.

Table 3: Commuter user stories

3.2 Driver User Stories

User Story	Acceptance Criteria	Definition of Done
Account Registration & Login	As a driver, I want to sign up and log in to my account, so that I can securely access and use the Taxi Tap app.	Given that I am on the app's welcome screen, when I choose "Sign Up" or "Log In" and enter valid details, then I should be authenticated and taken to the home screen. Based on my input criteria, I am taken to the home page of Taxi Tap
Announce Route & Destination	As a driver, I want to input the route I will be taking and the destination, so that passengers can see if I'm heading in their direction.	Given that I'm logged in, when I set my starting point and destination, then the route is visible to nearby passengers. The route is stored and displayed to the eligible passenger's interface.
Go Online/Offline	As a driver, I want to go online or offline as needed, so that I can control when I am available to receive ride requests.	Given that I'm on the driver dashboard, when I tap "Go Online" or "Go Offline", then my status is updated accordingly and affects request visibility. The driver's online/offline status is reflected, and the passenger can no longer see the taxi on the map.
Receive Ride Requests	As a driver, I want to receive ride requests from nearby passengers, so that I can choose which pick-ups to accept.	Given that I am online and have an active route, when a passenger requests a ride, then I receive a notification with request details. Ride requests from matching passengers are delivered in real-time to the driver's interface.

Accept or Decline Requests	As a driver, I want to accept or decline a ride request, so that I can manage my route and taxi capacity efficiently.	Given I have received a ride request, when I tap "Accept" or "Decline", then the system updates the request status and notifies the passenger. Accepted rides appear on the active list; declined requests are logged and cleared.
View Passenger Pickup Details	As a driver, I want to see the passenger's pickup point and basic information, so that I know where to stop and who I'm picking up.	Given that I've accepted a booking, when I view the trip summary, then I should see the passenger's location and name or contact info. Pickup details are accurately displayed on the driver's map and trip screen.
View Map & Navigation	As a driver, I want to see a map with passenger pickup and route directions, so that I can navigate efficiently.	Given that I have one or more assigned pickups, When I open the map view, then I should see my location and passenger's location. Live maps with GPS and routing is functional and accurate within the app.
Update Seat Availability	As a driver, I want to update how many seats are available in my taxi, so that passengers can decide whether to book or wait.	Given that I've started a trip or gone online, when I adjust seat count manually, Then, passengers see the updated availability. Seat count updates in real time and is reflected in the passenger's booking screen.
Receive Alerts for New Requests or Updates	As a driver, I want to receive real-time notifications, so that I don't miss ride requests or updates while driving.	Given that I am online, when a new request or important event occurs, then I receive a push notification with the relevant details. Push and in-app alerts trigger correctly and lead to actionable pages.

Work Offline (Partial Functionality)	As a driver, I want to continue using key features even when I'm offline, so that I can operate in areas with poor connectivity.	Given that I am offline or have poor signal, when I open the app, then I should be able to see cached routes and queue ride requests. The app stores critical data locally and syncs changes once reconnected.
Indicate Taxi Association	As a driver, I should be able to indicate which taxi association I am a part of.	Given I am registering or editing my profile, when I select or input my association, then it should be linked to my driver profile. Association name is stored and reflected in the driver's profile and backend database.
Receive Route from Association	As a driver, I should be assigned a route by my taxi association.	Given I belong to a taxi association, when I log in or go online, then the assigned route from the association should appear. System pulls assigned route from the association records and displays it on the app.
Receive Drop-Off Notification	As a driver, I should be notified when a passenger wants to get dropped off.	Given I am currently driving and have active passengers, when a passenger selects a drop-off point, then I receive an alert with the details. Notification appears in real-time and updates route on the driver's screen.

Table 4: Driver user stories

4 Service Contracts

Location

1. **getNearbyTaxis**
   - Query (Read-only)
   - **Request:**

```
{
  passengerLat: number,
```

```
  passengerLng: number
}
```

Example:

```
{
  "passengerLat": -25.746111,
  "passengerLng": 28.188056
}
```

- **Response:**

```
{
  _id: string,
  role: string,
  latitude: number,
  longitude: number
}
```

Example:

```
[
  {
    "_id": "qwerty123",
    "role": "driver",
    "latitude": -25.7471,
    "longitude": 28.2293
  }
]
```

- **Effect:**
  - This function receives the passenger's current location, finds all drivers in the system, calculates how far each driver is from the passenger, and returns only those drivers who are nearby (less than 5 km away).

2. **updateLocation**

- Mutation (Write operation)
- **Request:**

```
{
    userId: string,
    latitude: number,
    longitude: number
}
```

Example:

```
{
    "userId": "user_abc123",
    "latitude": -25.748333,
    "longitude": 28.1875
}
```

- **Response:**

```
null
```

- **Effect:**
  - This function updates the existing location record for a specific user by modifying their latitude, longitude, and updated timestamp. If no existing location record is found for the user, it throws an error.

3. **createLocation**

- Mutation (Write operation)
- **Request:**

```
{
    userId: string,
    latitude: number,
    longitude: number,
    role: "driver" | "passenger" | "both"
}
```

Example:

```
{
    "userId": "user_abc123",
    "latitude": -25.748333,
    "longitude": 28.1875,
    "role": "driver"
}
```

- **Response:**

```
null
```

- **Effect:**
  - Checks whether a location record already exists for the specified user.
  - If no existing location is found, inserts a new location record with the provided latitude, longitude, role, and current timestamp.
  - If a location already exists, no changes are made.

Routes

1. **findBestMatchingRoutes**

   - Query (Read-only)
   - **Request:**

   ```
   {
     startLat: number,
     startLon: number,
     endLat: number,
     endLon: number,
     maxStartDistance?: number,
     maxEndDistance?: number,
     maxResults?: number
   }
   ```

   Example:

   ```
   {
     "startLat": -25.75,
     "startLon": 28.19,
     "endLat": -25.76,
     "endLon": 28.21
   }
   ```

   - **Response:**

   ```
   {
     success: boolean,
     matchingRoutes: Array<{
       routeId: string,
       routeName: string,
       taxiAssociation: string,
       fare: number,
       estimatedDuration: number,
       startProximity: number,
       endProximity: number,
       totalScore: number,
       closestStartStop: {...} | null,
       closestEndStop: {...} | null,
       hasDirectRoute: boolean,
       isRecommended: boolean,
       totalStops: number,
       isActive: boolean
     }>,
     totalRoutesChecked: number,
     routesWithinRange: number,
     searchCriteria: {...},
   ```

```
  message: string
}
```

- **Effect:**
  - Evaluates all active routes.
  - Finds closest stops to start and end locations.
  - Scores routes based on proximity and directionality.
  - Returns sorted and filtered list of best-matching routes.

2. **getRouteWithStopsDetails**

- Query (Read-only)
- **Request:**

```
{
  routeId: string,
  userLat?: number,
  userLon?: number
}
```

Example:

```
{
  "routeId": "route123",
  "userLat": -25.75,
  "userLon": 28.19
}
```

- **Response:**

```
{
  success: boolean,
  route: {
    routeId: string,
    name: string,
    taxiAssociation: string,
    fare: number,
    estimatedDuration: number,
    isActive: boolean,
    totalStops: number,
    stops: Array<{
      id: string,
      name: string,
      coordinates: [number, number],
      order: number,
      distanceFromUser?: number
    }>
  } | null,
```

```
  message: string
}
```

- **Effect:**
  - Returns full route details including all stops.
  - Includes distance to each stop from user location if provided.
  - Uses enriched stop data if available.

3. **findNearbyStops**

- Query (Read-only)
- **Request:**

```
{
  lat: number,
  lon: number,
  radiusKm?: number,
  maxResults?: number
}
```

Example:

```
{
  "lat": -25.75,
  "lon": 28.19,
  "radiusKm": 2.5
}
```

- **Response:**

```
{
  success: boolean,
  nearbyStops: Array<{
    stop: {
      id: string,
      name: string,
      coordinates: [number, number],
      order: number
    },
    route: {
      routeId: string,
      name: string,
      taxiAssociation: string,
      fare: number
    },
    distance: number
  }>,
  searchLocation: {
```

```
    latitude: number,
    longitude: number
  },
  radiusKm: number,
  totalFound: number,
  message: string
}
```

- **Effect:**
  - Finds and returns all stops within a specified radius of the user.
  - Returns details about both the stop and its associated route.
  - Results are sorted by proximity.

4. **findAvailableTaxisForJourney**

- Query (Read-only)
- **Request:**

```
{
  originLat: number,
  originLng: number,
  destinationLat: number,
  destinationLng: number,
  maxOriginDistance?: number,
  maxDestinationDistance?: number,
  maxTaxiDistance?: number,
  maxResults?: number
}
```

Example:

```
{
  "originLat": -25.75,
  "originLng": 28.19,
  "destinationLat": -25.76,
  "destinationLng": 28.21
}
```

- **Response:**

```
{
  success: boolean,
  availableTaxis: Array<{
    driverId: string,
    userId: string,
    name: string,
    phoneNumber: string,
    vehicleRegistration: string,
```

```
  vehicleModel: string,
  vehicleColor: string,
  vehicleYear: number | null,
  isAvailable: boolean,
  numberOfRidesCompleted: number,
  averageRating: number,
  taxiAssociation: string,
  currentLocation: {
    latitude: number,
    longitude: number,
    lastUpdated: string
  },
  distanceToOrigin: number,
  routeInfo: {
    routeId: string,
    routeName: string,
    taxiAssociation: string,
    fare: number,
    estimatedDuration: number,
    startProximity: number,
    endProximity: number,
    totalScore: number,
    closestStartStop: {...} | null,
    closestEndStop: {...} | null
  }
}>,
matchingRoutes: Array<{
  routeId: string,
  routeName: string,
  taxiAssociation: string,
  fare: number,
  availableDrivers: number,
  startProximity: number,
  endProximity: number,
  totalScore: number
}>,
totalTaxisFound: number,
totalRoutesChecked: number,
validRoutesFound: number,
searchCriteria: {
  origin: { latitude: number, longitude: number },
  destination: { latitude: number, longitude: number },
  maxOriginDistance: number,
  maxDestinationDistance: number,
  maxTaxiDistance: number,
  maxResults: number
},
message: string
```

```
}
```

- **Effect:**
  - Identifies valid taxi routes that pass near both origin and destination.
  - Scores routes based on proximity and directness.
  - Gathers nearby drivers assigned to valid routes.
  - Returns sorted list of available taxis and matching routes.

5. **getNearbyTaxisForRouteRequest**

- Query (Read-only)
- **Request:**

```
{
  passengerLat: number,
  passengerLng: number,
  passengerEndLat: number,
  passengerEndLng: number
}
```

Example:

```
{
  "passengerLat": -25.7461,
  "passengerLng": 28.1880,
  "passengerEndLat": -25.7499,
  "passengerEndLng": 28.2091
}
```

- **Response:**

```
Array<{
  userId: string,
  latitude: number,
  longitude: number,
  role: "driver",
  updatedAt: string,
  _id: string,
  name: string,
  phoneNumber: string,
  vehicleRegistration: string,
  vehicleModel: string,
  vehicleColor: string,
  vehicleYear: number | null,
  isAvailable: boolean,
  numberOfRidesCompleted: number,
  averageRating: number,
  taxiAssociation: string,
```

```
  distanceToOrigin: number,
  routeInfo: {
    routeId: string,
    routeName: string,
    taxiAssociation: string,
    fare: number,
    estimatedDuration: number,
    startProximity: number,
    endProximity: number,
    totalScore: number,
    closestStartStop: {...} | null,
    closestEndStop: {...} | null
  }
}>
```

- **Effect:**
  - Wrapper for `findAvailableTaxisForJourney` with fixed proximity thresholds.
  - Transforms results for backward compatibility with older client format.

6. **_findAvailableTaxisForJourney**

- Internal Query (Read-only)
- **Request:**

```
{
  originLat: number,
  originLng: number,
  destinationLat: number,
  destinationLng: number,
  maxOriginDistance?: number,
  maxDestinationDistance?: number,
  maxTaxiDistance?: number,
  maxResults?: number
}
```

Example:

```
{
  "originLat": -25.7501,
  "originLng": 28.1888,
  "destinationLat": -25.7623,
  "destinationLng": 28.2010
}
```

- **Response:**

```
{
  success: boolean,
  availableTaxis: Array<AvailableTaxi>,
  matchingRoutes: Array<{
    routeId: string,
    routeName: string,
    taxiAssociation: string,
    fare: number,
    availableDrivers: number,
    startProximity: number,
    endProximity: number,
    totalScore: number
  }>,
  totalTaxisFound: number,
  totalRoutesChecked: number,
  validRoutesFound: number,
  searchCriteria: {
    origin: { latitude: number, longitude: number },
    destination: { latitude: number, longitude: number },
    maxOriginDistance: number,
    maxDestinationDistance: number,
    maxTaxiDistance: number,
    maxResults: number
  },
  message: string
}
```

- **Effect:**
  - Internally filters and scores routes based on distance and directionality between origin and destination.
  - Finds and filters nearby available drivers assigned to those routes.
  - Aggregates full driver, location, and vehicle metadata.

7. **getEnrichedStopName**
   - Action (Write operation)
   - **Request:**

```
{
  lat: number,
  lon: number
}
```

Example:

```
{
  "lat": -25.7479,
  "lon": 28.2293
```

```
}
```

- **Response:**

  ```
  string
  ```

  Example:

  ```
  "Centurion Mall"
  ```

- **Effect:**
  - Receives latitude and longitude coordinates, performs reverse geocoding using an internal API call, and returns a readable stop name. If reverse geocoding fails, it returns `"Unnamed Stop"`.

8. **getEnrichedStopsForRoute**

- Query (Read-only)
- **Request:**

  ```
  {
    routeId: string
  }
  ```

  Example:

  ```
  {
    "routeId": "route-123"
  }
  ```

- **Response:**

  ```
  [
    {
      id: string,
      name: string,
      order: number
    }
  ]
  ```

  Example:

  ```
  [
    {
      "id": "stop-1",
      "name": "Centurion",
      "order": 1
    }
  ]
  ```

- **Effect:**
  - Retrieves enriched stops for a given route from the database, filters out stops with meaningless names, and returns only valid stop data.

9. **displayRoutes**

- Query (Read-only)
- **Request:**

```
{}
```

- **Response:**

```
[
  {
    routeId: string,
    start: string,
    destination: string,
    startCoords: { latitude: number, longitude: number } | null,
    destinationCoords: { latitude: number, longitude: number } | null,
    stops: [],
    fare: number,
    estimatedDuration: number,
    taxiAssociation: string,
    hasStops: boolean
  }
]
```

Example:

```
[
  {
    "routeId": "route-1",
    "start": "Pretoria",
    "destination": "Johannesburg",
    "startCoords": { "latitude": -25.7479, "longitude": 28.2293 },
    "destinationCoords": { "latitude": -26.2041, "longitude": 28.0473 },
    "stops": [],
    "fare": 45,
    "estimatedDuration": 3600,
    "taxiAssociation": "PUTCO",
    "hasStops": false
  }
]
```

- **Effect:**
  - Reads all route data, processes coordinates, extracts start and destination points, calculates fare based on duration, and formats route information for display.

10. **displayRoutesPaginated**

- Query (Read-only)
- **Request:**

```
{
  page?: number,
  limit?: number
}
```

Example:

```
{
  "page": 1,
  "limit": 10
}
```

- **Response:**

```
{
  routes: [...],
  pagination: {
    currentPage: number,
    totalPages: number,
    totalRoutes: number,
    hasNextPage: boolean,
    hasPrevPage: boolean,
    limit: number
  }
}
```

Example:

```
{
  "routes": [...],
  "pagination": {
    "currentPage": 1,
    "totalPages": 3,
    "totalRoutes": 25,
    "hasNextPage": true,
    "hasPrevPage": false,
    "limit": 10
  }
}
```

- **Effect:**
  - Extends `displayRoutes` with pagination support. Returns paginated list of routes with metadata describing current page, total pages, and availability of next/previous pages.

11. **assignRandomRouteToDriver**

- Mutation (Write operation)
- **Request:**

```
{
  userId: string,  // Convex document ID (taxiTap_users)
  taxiAssociation: string
}
```

Example:

```
{
  "userId": "user-123",
  "taxiAssociation": "PUTCO"
}
```

- **Response:**

```
{
  success: boolean,
  message: string,
  assignedRoute: {
    _id: string,
    routeId: string,
    name: string,
    geometry: object,
    taxiAssociation: string,
    estimatedDuration: number,
    isActive: boolean
  }
}
```

Example:

```
{
  "success": true,
  "message": "Route assigned successfully",
  "assignedRoute": {
    "_id": "route-456",
    "routeId": "route-456",
    "name": "Pretoria - Johannesburg",
    "geometry": { ... },
    "taxiAssociation": "PUTCO",
    "estimatedDuration": 3600,
    "isActive": true
  }
}
```

- **Effect:**
  - This function assigns a random active route (belonging to the driver's taxi association) to a driver. If no active routes are available, or if the driver does not exist, the function throws an error. Upon successful assignment, it updates the driver's assigned route, association, and timestamp.

12. **getRouteStopsWithEnrichment**

- Query (Read-only)
- **Request:**

```
{
  routeId: string
}
```

Example:

```
{
  "routeId": "route-123"
}
```

- **Response:**

```
{
  stops: array,
  isEnriched: boolean,
  updatedAt: string | null
}
```

Example:

```
{
  "stops": [...],
  "isEnriched": true,
  "updatedAt": "2024-06-15T10:00:00Z"
}
```

- **Effect:**
  - Attempts to retrieve enriched route stops from the database. If enrichment is unavailable, it falls back to original route stops.

13. **getAllRoutesWithEnrichmentStatus**

- Query (Read-only)
- **Request:**

```
{}
```

- **Response:**

```
[
  {
    _id: string,
    routeId: string,
    name: string,
    geometry: object,
    taxiAssociation: string,
    estimatedDuration: number,
    isActive: boolean,
    hasEnrichedStops: boolean
  }
]
```

- **Effect:**
  - Returns all routes and indicates if enriched stops exist for each route.

14. **getAllAvailableRoutesForPassenger**

- Query (Read-only)
- **Request:**

```
{}
```

- **Response:**

```
[
  {
    routeId: string,
    routeName: string,
    start: string,
    destination: string,
    taxiAssociation: string,
    fare: number,
    estimatedDuration: number,
    stops: array,
    totalStops: number
  }
]
```

- **Effect:**
  - Returns a sorted list of all active routes available for passengers, including parsed start and destination names.

15. **getRoutesByTaxiAssociationForPassenger**

- Query (Read-only)

- **Request:**

```
{
  taxiAssociation: string
}
```

Example:

```
{
  "taxiAssociation": "PUTCO"
}
```

- **Response:**

```
[
  {
    routeId: string,
    routeName: string,
    start: string,
    destination: string,
    taxiAssociation: string,
    fare: number,
    estimatedDuration: number,
    stops: array,
    totalStops: number
  }
]
```

- **Effect:**
  - Returns all active routes for a specific taxi association, allowing passengers to filter by association.

16. **getRouteDetailsWithDrivers**

- Query (Read-only)
- **Request:**

```
{
  routeId: string
}
```

Example:

```
{
  "routeId": "route-123"
}
```

- **Response:**

```
{
  success: boolean,
  route: {
    routeId: string,
    routeName: string,
    start: string,
    destination: string,
    taxiAssociation: string,
    fare: number,
    estimatedDuration: number,
    stops: array,
    geometry: object,
    totalStops: number,
    isActive: boolean
  } | null,
  activeDrivers: [
    {
      driverId: string,
      driverName: string,
      averageRating: number,
      totalRides: number,
      isActive: boolean
    }
  ],
  message: string
}
```

- **Effect:**
  - Returns complete route details along with all currently active drivers assigned to that route.

17. **getDriverAssignedRoute**

- Query (Read-only)
- **Request:**

```
{
  userId: string
}
```

Example:

```
{
  "userId": "user-123"
}
```

- **Response:**

```
{
  _id: string,
  routeId: string,
  name: string,
  geometry: object,
  taxiAssociation: string,
  estimatedDuration: number,
  isActive: boolean,
  stops: array
} | null
```

- **Effect:**
  - Retrieves the assigned route for a specific driver, or null if the driver has no route assignment.

18. **getAllTaxiAssociations**

- Query (Read-only)
- **Request:**

```
{}
```

- **Response:**

```
[
  string
]
```

Example:

```
[
  "PUTCO",
  "JMPD",
  "Gautrain"
]
```

- **Effect:**
  - Returns a sorted list of all unique taxi associations present in the system.

19. **getCachedStop (InternalQuery)**

- Internal Query (Read-only, internal use)
- **Request:**

```
{
  key: string
}
```

Example:

```
{
  "key": "-25.74790,28.22930"
}
```

- **Response:**

```
{
  id: string,
  name: string,
  lastUsed: number
} | null
```

- **Effect:**
  - Looks up if the reverse geocoded stop name for given coordinates is already cached in the database. If not found, returns null.

20. **cacheStop (InternalMutation)**

- Internal Mutation (Write-only, internal use)
- **Request:**

```
{
  key: string,
  name: string
}
```

Example:

```
{
  "key": "-25.74790,28.22930",
  "name": "Menlyn Mall"
}
```

- **Response:**

```
(void)
```

- **Effect:**
  - Caches the reverse geocoded stop name for future queries to avoid unnecessary API calls.

21. **getReadableStopName (Action)**

- Action (Performs side effects, external API calls)
- **Request:**

```
{
  lat: number,
  lon: number
}
```

Example:

```
{
  "lat": -25.74790,
  "lon": 28.22930
}
```

- **Response:**

```
string
```

Example:

```
"Menlyn Mall"
```

- **Effect:**
  - This action first checks if the location name is already cached.
  - If not cached, it queries the OpenStreetMap Nominatim API to get a human-readable location name.
  - After successful retrieval, it caches the result for future efficiency.
  - If any error occurs during API call, returns `"Unnamed Stop"` as fallback.

Taxis

1. **getAvailableTaxis (Query)**
   - Query (Read-only)
   - **Request:**

```
{}
```

   - **Response:**

```
[
  {
    licensePlate: string,
    image: string | null,
    seats: number,
    model: string,
    driverName: string,
    userId: string
  }
]
```

Example:

```
[
  {
    "licensePlate": "ABC123",
    "image": null,
    "seats": 4,
    "model": "Toyota Avanza",
    "driverName": "John Doe",
    "userId": "user123"
  }
]
```

- **Effect:**
  - Retrieves all taxis that are marked as available in the system.
  - For each available taxi, retrieves the corresponding driver and user information.
  - Returns a list of available taxis enriched with driver names and related user IDs.

2. **getTaxiForDriver (Query)**

- Query (Read-only)
- **Request:**

```
{
  userId: string
}
```

Example:

```
{
  "userId": "user123"
}
```

- **Response:**

```
{
  _id: string,
  driverId: string,
  licensePlate: string,
  model: string,
  capacity: number,
  image: string | null
} | null
```

Example:

```
{
  "_id": "taxi456",
  "driverId": "driver789",
  "licensePlate": "XYZ789",
  "model": "Toyota Quantum",
  "capacity": 14,
  "image": "https://example.com/taxi.jpg"
}
```

- **Effect:**
  - Given a driver's user ID, this query retrieves the taxi assigned to that driver.
  - If no driver profile is found for the given user ID, it returns `null`.
  - If a driver exists but no taxi is assigned, it also returns `null`.

3. **updateTaxiSeatAvailability**

- Mutation (Write operation)
- **Request:**

```
{
  rideId: string,
  action: "decrease" | "increase"
}
```

Example:

```
{
  "rideId": "ride_abc123",
  "action": "decrease"
}
```

- **Response:**

```
{
  success: true,
  updatedSeats: number,
  previousSeats: number
}
```

Example:

```
{
  "success": true,
  "updatedSeats": 3,
  "previousSeats": 4
}
```

- **Effect:**
  - Finds the taxi associated with the driver of the given ride.
  - Increments or decrements the ‘capacity‘ (seats available) based on the action.
  - Ensures the value does not drop below zero when decreasing.
  - Updates the ‘updatedAt‘ timestamp for the taxi record.

4. **updateTaxiInfo (Mutation)**

- Mutation (Modifies data)
- **Request:**

```
{
  userId: string,
  licensePlate?: string,
  model?: string,
  color?: string,
  year?: number,
  image?: string,
  capacity?: number,
  isAvailable?: boolean
}
```

Example:

```
{
  "userId": "user123",
  "licensePlate": "XYZ789",
  "model": "Toyota Quantum",
  "color": "White",
  "year": 2019,
  "capacity": 14,
  "isAvailable": true
}
```

- **Response:**

```
{
  success: boolean,
  taxiId: string
}
```

Example:

```
{
  "success": true,
  "taxiId": "taxi456"
}
```

- **Effect:**
  - Updates the taxi information for the driver identified by the given `userId`.
  - Only fields provided in the request will be updated; all others remain unchanged.
  - Throws an error if no driver profile or taxi is found for the user.

5. **viewTaxiInfo (Query)**

- Query (Read-only)
- **Request:**

```
{
  passengerId: string
}
```

Example:

```
{
  "passengerId": "passenger123"
}
```

- **Response:**

```
{
  taxi: {
    _id: string,
    licensePlate: string,
    model: string,
    color?: string,
    year?: number,
    image?: string,
    capacity: number,
    isAvailable: boolean,
    updatedAt?: number
  },
  driver: {
    name?: string,
    phoneNumber?: string,
    rating?: number
  },
  rideId: string,
  status: string
}
```

Example:

```
{
  "taxi": {
```

```
    "_id": "taxi789",
    "licensePlate": "XYZ123",
    "model": "Toyota Quantum",
    "capacity": 14,
    "isAvailable": true
  },
  "driver": {
    "name": "John Doe",
    "phoneNumber": "+27123456789",
    "rating": 4.8
  },
  "rideId": "ride456",
  "status": "in_progress"
}
```

- **Effect:**
  - Retrieves the taxi and driver information related to the passenger's most recent or active ride reservation.
  - Throws an error if no active reservation or assigned driver is found.
  - Useful for passengers to view taxi details after reserving a seat.

User

1. **getUserById (Query)**

   - Query (Read-only)
   - **Request:**

     ```
     {
       userId: string
     }
     ```

     Example:

     ```
     {
       "userId": "user123"
     }
     ```

   - **Response:**

     ```
     {
       _id: string,
       name: string,
       email: string,
       age?: number,
       phoneNumber: string,
       isVerified: boolean,
       isActive: boolean,
     ```

```
  accountType: string,
  currentActiveRole?: string,
  lastRoleSwitchAt?: string,
  profilePicture?: string,
  dateOfBirth?: string,
  gender?: string,
  emergencyContact?: string,
  createdAt: string,
  updatedAt: string,
  lastLoginAt?: string
}
```

- **Effect:**
  - Retrieves a user by their Convex document ID.
  - Throws an error if the user is not found.
  - Excludes sensitive fields like passwords.

2. **getUserByPhone (Query)**
   - Query (Read-only)
   - **Request:**

```
{
  phoneNumber: string
}
```

Example:

```
{
  "phoneNumber": "+27123456789"
}
```

   - **Response:**

```
{
  _id: string,
  name: string,
  email: string,
  age?: number,
  phoneNumber: string,
  isVerified: boolean,
  isActive: boolean,
  accountType: string,
  currentActiveRole?: string,
  lastRoleSwitchAt?: string,
  profilePicture?: string,
  dateOfBirth?: string,
  gender?: string,
```

```
  emergencyContact?: string,
  createdAt: string,
  updatedAt: string,
  lastLoginAt?: string
}
```

- **Effect:**
  - Retrieves a user by their phone number.
  - Throws an error if no user with the given phone number exists.
  - Useful for login or phone-based lookup.

3. **getUserWithProfiles (Query)**

- Query (Read-only)
- **Request:**

```
{
  userId: string
}
```

Example:

```
{
  "userId": "user123"
}
```

- **Response:**

```
{
  user: {
    _id: string,
    name: string,
    email: string,
    age?: number,
    phoneNumber: string,
    isVerified: boolean,
    isActive: boolean,
    accountType: string,
    currentActiveRole?: string,
    lastRoleSwitchAt?: string,
    profilePicture?: string,
    dateOfBirth?: string,
    gender?: string,
    emergencyContact?: string,
    createdAt: string,
    updatedAt: string,
    lastLoginAt?: string
  },
```

```
  driverProfile: object | null,
  passengerProfile: object | null
}
```

- **Effect:**
  - Retrieves user data along with associated driver and passenger profiles (if any).
  - Returns ‘null‘ for profiles if they do not exist.
  - Throws an error if the user is not found.

4. **loginSMS (Query)**

- Query (Read-only)
- **Request:**

```
{
  phoneNumber: string,
  password: string
}
```

Example:

```
{
  "phoneNumber": "+27123456789",
  "password": "user_password_here"
}
```

- **Response:**

```
{
  id: string,
  phoneNumber: string,
  name: string,
  accountType: string,
  currentActiveRole: string,
  isVerified: boolean
}
```

- **Effect:**
  - Authenticates user by verifying the provided password against a securely stored salted hash using PBKDF2 with SHA-256.
  - Checks if the user account is active.
  - Confirms that the user's current active role matches their account type or is "both".
  - Throws errors for:
    - User not found,

        * Invalid password,
        * Deactivated account,
        * Missing active role,
        * Role mismatch between active role and account permissions.
    - Returns user identification and status information for a successful login.

5. **signUpSMS (Mutation)**

    - Mutation (Modifies data)
    - **Request:**

```
{
  phoneNumber: string,
  name: string,
  password: string,
  accountType: "passenger" | "driver" | "both",
  email?: string,
  age?: number
}
```

      Example:

```
{
  "phoneNumber": "+27123456789",
  "name": "John Doe",
  "password": "securePassword123",
  "accountType": "driver",
  "email": "john@example.com",
  "age": 30
}
```

    - **Response:**

```
{
  success: boolean,
  userId: string
}
```

    - **Effect:**
        - Creates a new user with the specified account type and details.
        - Hashes the password securely using PBKDF2 with a random salt and SHA-256.
        - Prevents duplicate phone numbers and throws an error if the phone number already exists.
        - Initializes location record with default coordinates (latitude: 0, longitude: 0) and role based on accountType.

- Creates associated driver and/or passenger profiles depending on the account type.
- Sets default values for user status flags (‘isVerified‘, ‘isActive‘) and timestamps.
- Handles potential race conditions on phone number uniqueness.

6. **switchActiveRole (Mutation)**

- Mutation (Modifies data)
- **Request:**

```
{
  userId: string,
  newRole: "passenger" | "driver"
}
```

Example:

```
{
  "userId": "user123",
  "newRole": "driver"
}
```

- **Response:**

```
{
  success: boolean,
  message: string,
  newRole: "passenger" | "driver"
}
```

- **Effect:**
  - Allows a user with both passenger and driver roles to switch their active role.
  - Throws an error if the user does not exist or does not have both account types.
  - Prevents switching to the same active role the user already has.
  - Checks for active rides (accepted or in progress) in the current role before allowing a switch.
    - If switching to ”passenger”, ensures no active driver rides exist.
    - If switching to ”driver”, ensures no active passenger rides exist.
  - Updates the user’s current active role and timestamps on successful switch.

7. **switchBothToDriver (Mutation)**

- Mutation (Modifies data)
- **Request:**

```
{
  userId: string
}
```

Example:

```
{
  "userId": "user123"
}
```

- **Response:**

```
{
  success: boolean,
  message: string
}
```

- **Effect:**
  - Switches a user with account type `both` to `driver` only.
  - Throws an error if the user does not exist or is not currently `both`.
  - Prevents switching if the user has any active rides as a passenger (statuses: `requested`, `accepted`, `in_progress`).
  - Updates the user's account type to `driver` and sets the active role to `driver`.
  - Updates `roleSwitchAt` and updated timestamps.

8. **switchBothToPassenger (Mutation)**

- Mutation (Modifies data)
- **Request:**

```
{
  userId: string
}
```

Example:

```
{
  "userId": "user123"
}
```

- **Response:**

```
{
  success: boolean,
  message: string
}
```

- **Effect:**
  - Switches a user with account type `both` to `passenger` only.
  - Throws an error if the user does not exist or is not currently `both`.
  - Prevents switching if the user has any active rides as a driver (statuses: `accepted`, `in_progress`).
  - Updates the user's account type to `passenger` and sets the active role to `passenger`.
  - Updates `roleSwitchAt` and updated timestamps.

9. **switchDriverToBoth (Mutation)**

- Mutation (Modifies data)
- **Request:**

```
{
  userId: string
}
```

Example:

```
{
  "userId": "user123"
}
```

- **Response:**

```
{
  success: boolean,
  message: string
}
```

- **Effect:**
  - Upgrades a user from "driver" to "both" (driver and passenger).
  - Throws an error if the user does not exist or is not currently a driver.
  - Sets the account type to "both" and the current active role to "driver".
  - Updates role switch and updated timestamps.
  - Creates a passenger profile with default values if one does not already exist.

10. **switchPassengerToBoth (Mutation)**

- Mutation (Modifies data)
- **Request:**

```
{
  userId: string
}
```

Example:

```
{
  "userId": "user123"
}
```

- **Response:**

```
{
  success: boolean,
  message: string
}
```

- **Effect:**
  - Upgrades a user from "passenger" to "both" (passenger and driver).
  - Throws an error if the user does not exist or is not currently a passenger.
  - Sets the account type to "both" and the current active role to "passenger".
  - Updates role switch and updated timestamps.
  - Creates a driver profile with default values if one does not already exist.

Notifications

deactivatePushToken

- Mutation (Write operation)
- **Request:**

```
{
  token: string
}
```

Example:

```
{
  "token": "abc123def456"
}
```

- **Response:**

```
{
  _id: string,
  isActive: boolean,
  updatedAt: number
} \text{ or } null
```

Example:

```
{
  "_id": "pushToken_xyz789",
  "isActive": false,
  "updatedAt": 1687804800000
}
```

- **Effect:**
  - Searches the `pushTokens` collection for a document matching the provided token.
  - If found, updates the document to set `isActive` to `false` and `updatedAt` to the current timestamp.
  - If no matching token is found, returns `null`.

getNotifications

- Query (Read operation)
- **Request:**

```
{
  userId: Id<"taxiTap_users">,
  limit?: number,
  unreadOnly?: boolean
}
```

Example:

```
{
  "userId": "user_abc123",
  "limit": 10,
  "unreadOnly": true
}
```

- **Response:**

```
[
  {
    _id: string,
    userId: string,
    message: string,
    isRead: boolean,
    createdAt: number,
    // ...other notification fields
```

```
  },
  ...
]
```

Example:

```
[
  {
    "_id": "notif_001",
    "userId": "user_abc123",
    "message": "Your ride is arriving soon",
    "isRead": false,
    "createdAt": 1687804800000
  },
  {
    "_id": "notif_002",
    "userId": "user_abc123",
    "message": "New promotional offer available",
    "isRead": true,
    "createdAt": 1687804700000
  }
]
```

- **Effect:**
  - Retrieves notifications for the specified user from the database.
  - Can optionally filter to only unread notifications if `unreadOnly` is true.
  - Can optionally limit the number of notifications returned.
  - Notifications are ordered descending by creation time (most recent first).

getUnreadCount

- Query (Read operation)
- **Request:**

```
{
  userId: Id<"taxiTap_users">
}
```

Example:

```
{
  "userId": "user_abc123"
}
```

- **Response:**

```
number
```

Example:

```
5
```

- **Effect:**
  - Counts and returns the number of unread notifications for the specified user.

getNotificationSettings

- Query (Read operation)
- **Request:**

```
{
  userId: Id<"taxiTap_users">
}
```

Example:

```
{
  "userId": "user_abc123"
}
```

- **Response:**

```
{
  rideUpdates: boolean,
  promotionalOffers: boolean,
  systemAlerts: boolean,
  emergencyNotifications: boolean,
  routeUpdates: boolean,
  paymentNotifications: boolean,
  ratingReminders: boolean,
  soundEnabled: boolean,
  vibrationEnabled: boolean,
  quietHoursStart: string,
  quietHoursEnd: string
}
```

Example:

```
{
  "rideUpdates": true,
  "promotionalOffers": true,
  "systemAlerts": true,
  "emergencyNotifications": true,
  "routeUpdates": true,
  "paymentNotifications": true,
  "ratingReminders": true,
  "soundEnabled": true,
  "vibrationEnabled": true,
  "quietHoursStart": "22:00",
  "quietHoursEnd": "07:00"
}
```

- **Effect:**
  - Retrieves the user's notification settings from the database.
  - If no custom settings exist, returns a default notification preference object.

markAllAsRead

- Mutation (Write operation)
- **Request:**

```
{
  userId: Id<"taxiTap_users">
}
```

Example:

```
{
  "userId": "user_abc123"
}
```

- **Response:**

```
<number of notifications marked as read>
```

Example:

```
5
```

- **Effect:**

- Queries all unread notifications for the given user.
- Updates each to mark them as read by setting:
  - `isRead` to `true`
  - `readAt` to the current timestamp
- Returns the count of notifications that were marked as read.

markAsRead

- Mutation (Write operation)
- **Request:**

```
{
  notificationId: Id<"notifications">
}
```

Example:

```
{
  "notificationId": "notif_456xyz"
}
```

- **Response:**

```
{
  _id: "notif_456xyz",
  isRead: true,
  readAt: <timestamp>,
  ... // other fields unchanged
}
```

- **Effect:**
  - Locates the notification with the given ID.
  - Updates the following fields:
    - `isRead` → `true`
    - `readAt` → current timestamp
  - Returns the updated notification document.

registerPushToken

- Mutation (Write operation)
- **Request:**

```
{
  userId: Id<"taxiTap_users">,
  token: string,
  platform: "ios" | "android"
}
```

Example:

```
{
  userId: "user_abc123",
  token: "fcmToken1234567890",
  platform: "android"
}
```

- **Response:**

```
{
  _id: "pushToken_789xyz",
  userId: "user_abc123",
  token: "fcmToken1234567890",
  platform: "android",
  isActive: true,
  createdAt: <timestamp>,
  updatedAt: <timestamp>,
  lastUsedAt: <timestamp>
}
```

- **Effect:**
  - If the token already exists:
    - Updates the ‘userId‘, ‘isActive‘, ‘updatedAt‘, and ‘lastUsedAt‘ fields.
  - If the token does not exist:
    - Inserts a new push token document.

sendRideNotification

- Internal Mutation (Write operation)
- **Request:**

```
{
  rideId: string,
  type: string,
  passengerId?: Id<"taxiTap_users">,
  driverId?: Id<"taxiTap_users">
}
```

Example:

```
{
  rideId: "ride_12345",
  type: "ride_completed",
  driverId: "user_driver123"
}
```

- **Response:**

```
undefined
```

- **Effect:**
  - Depending on the notification type, constructs and sends one or more ride-related notifications to passengers and/or drivers using:

    `internal.functions.notifications.sendNotifications.sendNotificationInternal`

  - Notification types supported:
  * "ride$_r$*equested*" : *Senttodriver*"ride$_a$*ccepted*" : *Senttopassenger*"driver$_a$*rrived*" : *Sentto*

* **sendRideNotification**

- Internal Mutation (Write operation)
- **Request:**

```
{
  rideId: string,
  type: string,
  passengerId?: Id<"taxiTap_users">,
  driverId?: Id<"taxiTap_users">
}
```

Example:

```
{
  "rideId": "ride_abc123",
  "type": "ride_requested",
  "passengerId": "user_xyz789",
  "driverId": "user_def456"
}
```

- **Response:**

`void (no direct response)`

- **Effect:**
  - Looks up the ride by the given `rideId`.
  - Depending on the `type` of notification, prepares one or more notification objects targeting the passenger and/or driver.
  - Notifications include type, title, message, priority, and metadata with relevant ride and user information.
  - Sends notifications internally by calling `sendNotificationInternal` mutation for each prepared notification.
  - If the ride is not found, no notifications are sent.

sendNotification

- Mutation (Write operation)
- **Request:**

```
{
  userId: Id<"taxiTap_users">,
  type: string,
  title: string,
  message: string,
  priority: string,
  metadata?: any,
  scheduledFor?: number,
  expiresAt?: number
}
```

Example:

```
{
  "userId": "user_abc123",
  "type": "ride_request",
  "title": "New Ride Request",
  "message": "You have a new ride request.",
  "priority": "high",
  "metadata": { "rideId": "ride_xyz789" },
  "scheduledFor": 1687804800000,
  "expiresAt": 1687891200000
}
```

- **Response:**

```
{
  _id: string,
  notificationId: string,
  userId: string,
  type: string,
  title: string,
  message: string,
  isRead: boolean,
  isPush: boolean,
  priority: string,
  metadata?: any,
  scheduledFor?: number,
  expiresAt?: number,
  createdAt: number
}
```

Example:

```
{
  "_id": "notif_doc123",
  "notificationId": "notif_1687804800000_ab12cd34e",
  "userId": "user_abc123",
  "type": "ride_request",
  "title": "New Ride Request",
  "message": "You have a new ride request.",
  "isRead": false,
  "isPush": true,
  "priority": "high",
  "metadata": { "rideId": "ride_xyz789" },
  "scheduledFor": 1687804800000,
  "expiresAt": 1687891200000,
  "createdAt": 1687804800000
}
```

- **Effect:**
  - Inserts a new notification document into the `notifications` collection.
  - Automatically generates a unique `notificationId`.
  - Sets `isRead` to false by default and `isPush` to false initially.
  - Queries active push tokens for the user; if any exist, updates the notification to mark `isPush` as true.
  - Returns the created notification document.

sendNotificationInternal

- Internal Mutation (Write operation)

- **Request:** Same as `sendNotification`.
- **Response:** Same as `sendNotification`.
- **Effect:** Same as `sendNotification`, but intended for system-generated notifications.

updateNotificationSettings

- Mutation (Write operation)
- **Request:**

```
{
  userId: Id<"taxiTap_users">,
  settings: {
    rideUpdates?: boolean,
    promotionalOffers?: boolean,
    systemAlerts?: boolean,
    emergencyNotifications?: boolean,
    routeUpdates?: boolean,
    paymentNotifications?: boolean,
    ratingReminders?: boolean,
    soundEnabled?: boolean,
    vibrationEnabled?: boolean,
    quietHoursStart?: string,
    quietHoursEnd?: string
  }
}
```

Example:

```
{
  "userId": "user_abc123",
  "settings": {
    "rideUpdates": false,
    "promotionalOffers": true,
    "quietHoursStart": "23:00",
    "quietHoursEnd": "06:00"
  }
}
```

- **Response:**

```
{
  _id: string,
  userId: string,
  rideUpdates: boolean,
```

```
  promotionalOffers: boolean,
  systemAlerts: boolean,
  emergencyNotifications: boolean,
  routeUpdates: boolean,
  paymentNotifications: boolean,
  ratingReminders: boolean,
  soundEnabled: boolean,
  vibrationEnabled: boolean,
  quietHoursStart: string,
  quietHoursEnd: string,
  createdAt: number,
  updatedAt: number
}
```

Example:

```
{
  "_id": "notifSettings_123xyz",
  "userId": "user_abc123",
  "rideUpdates": false,
  "promotionalOffers": true,
  "systemAlerts": true,
  "emergencyNotifications": true,
  "routeUpdates": true,
  "paymentNotifications": true,
  "ratingReminders": true,
  "soundEnabled": true,
  "vibrationEnabled": true,
  "quietHoursStart": "23:00",
  "quietHoursEnd": "06:00",
  "createdAt": 1687804800000,
  "updatedAt": 1687891200000
}
```

- **Effect:**
  - Checks if notification settings for the user already exist.
  - If existing, updates only the provided settings fields and the `updatedAt` timestamp.
  - If not, creates a new settings document with provided values or defaults.
  - Defaults are mostly `true` for booleans, and `"22:00"` / `"07:00"` for quiet hours.

Rides

1. **acceptRide**

- Mutation (Write operation)
- **Request:**

```
{
    rideId: string,
    driverId: string
}
```

Example:

```
{
    "rideId": "ride_xyz123",
    "driverId": "driver_abc789"
}
```

- **Response:**

```
{
    _id: string,
    message: string
}
```

- **Effect:**
  - Finds the ride by rideId.
  - Throws an error if ride not found or if the ride status is not "requested".
  - Updates the ride's status to "accepted" and assigns the driverId.
  - Sets the acceptedAt timestamp.
  - Sends a notification to the passenger that the ride has been accepted.

2. **cancelRide**

- Mutation (Write operation — external access)
- **Request:**

```
{
    rideId: string,
    userId: string
}
```

Example:

```
{
    "rideId": "ride_abc123",
    "userId": "user_xyz789"
}
```

- **Response:**

```
{
    _id: string,    // Updated ride document ID
    message: string // Success confirmation message
}
```

- **Effect:**
  - Finds the ride by `rideId` and verifies that `userId` matches either the passenger or driver of the ride.
  - Updates the ride status to `"cancelled"` and records the cancellation time and user.
  - Sends a notification to the other party (driver or passenger) about the cancellation, specifying who cancelled.
  - Throws an error if the ride does not exist or the user is unauthorized to cancel.

3. **completeRide**

- Mutation (Write operation — external access)
- **Request:**

```
{
    rideId: string,
    driverId: string
}
```

Example:

```
{
    "rideId": "ride_abc123",
    "driverId": "user_driver456"
}
```

- **Response:**

```
{
    _id: string,    // Updated ride document ID
    message: string // Confirmation message
}
```

- **Effect:**
  - Validates that the ride exists and the requesting driver is the assigned driver.
  - Ensures the ride status is currently "accepted" before completing.
  - Updates the ride status to "completed" and sets the completion timestamp.
  - Sends notifications to both passenger and driver via internal notification system.

- Throws an error if the ride is not found, the driver is unauthorized, or the ride is not in the correct status.

4. **endRide**

   - Mutation (Write operation)
   - **Request:**

     ```
     {
       rideId: string,
       userId: Id<"taxiTap_users">
     }
     ```

     Example:

     ```
     {
       "rideId": "ride_abc123",
       "userId": "user_xyz789"
     }
     ```

   - **Response:**

     ```
     {
       // Response depends on the implementation in endRideHandler,
       // typically a status object or updated ride info,
       // or void if no response is returned.
     }
     ```

     Example:

     ```
     {
       "status": "success",
       "rideId": "ride_abc123",
       "endedAt": 1687804800000
     }
     ```

   - **Effect:**
     - Invokes the `endRideHandler` to mark the ride as ended.
     - Typically updates ride status and records the end timestamp.
     - May trigger related business logic like fare calculation, notifications, etc.

5. **getRideById**

   - Query (Read operation)
   - **Request:**

     ```
     {
         rideId: string
     }
     ```

Example:

```
{
    "rideId": "ride_abc123"
}
```

- **Response:**

```
{
    _id: string,
    rideId: string,
    passengerId: string,
    driverId?: string,
    status: string,
    startLocation: object,
    endLocation: object,
    requestedAt: number,
    acceptedAt?: number,
    completedAt?: number,
    ... // other ride fields
}
// or null if rideId is not provided or ride not found
```

- **Effect:**
  - Retrieves a ride document by its rideId.
  - Returns null if no rideId is provided or the ride is not found.

6. **requestRide**

- Mutation (Write operation)
- **Request:**

```
{
    passengerId: string,
    driverId: string,
    startLocation: {
        coordinates: { latitude: number, longitude: number },
        address: string
    },
    endLocation: {
        coordinates: { latitude: number, longitude: number },
        address: string
    },
    estimatedFare?: number,
    estimatedDistance?: number,
    estimatedDuration?: number
}
```

Example:

```
{
    "passengerId": "user_passenger123",
    "driverId": "user_driver456",
    "startLocation": {
        "coordinates": { "latitude": -25.748333, "longitude": 28.1875 },
        "address": "123 Main St, Pretoria"
    },
    "endLocation": {
        "coordinates": { "latitude": -25.757, "longitude": 28.229 },
        "address": "456 Elm St, Pretoria"
    },
    "estimatedFare": 150.50,
    "estimatedDistance": 12.5,
    "estimatedDuration": 1800
}
```

- **Response:**

```
{
    _id: string,      // Database document ID
    rideId: string,   // Generated ride identifier
    message: string   // Confirmation message
}
```

- **Effect:**
  - Creates a new ride request record with the status set to "requested".
  - Generates a unique ride ID.
  - Notifies the assigned driver of the new ride request via the internal notification system.

7. **startRide**

- Mutation (Write operation)
- **Request:**

```
{
  rideId: string,
  userId: Id<"taxiTap_users">
}
```

Example:

```
{
  "rideId": "ride_xyz123",
  "userId": "user_abc789"
}
```

- **Response:**

```
{
  _id: string,
  message: string
}
```

Example:

```
{
  "_id": "ride_doc456",
  "message": "Ride marked as started."
}
```

- **Effect:**
  - Finds the ride by `rideId`.
  - Throws an error if the ride is not found.
  - Throws an error if the user is not the passenger of the ride.
  - Throws an error if the ride status is not "accepted".
- Updates the ride status to "in$_progress$" *andsetsthe*`startedAt`*timestamp.Sendsnotifications*

8. **declineRide**

- Mutation (Write operation)
- **Request:**

```
{
  rideId: string,
  driverId: Id<"taxiTap_users">
}
```

Example:

```
{
  "rideId": "ride_xyz123",
  "driverId": "user_driver456"
}
```

- **Response:**

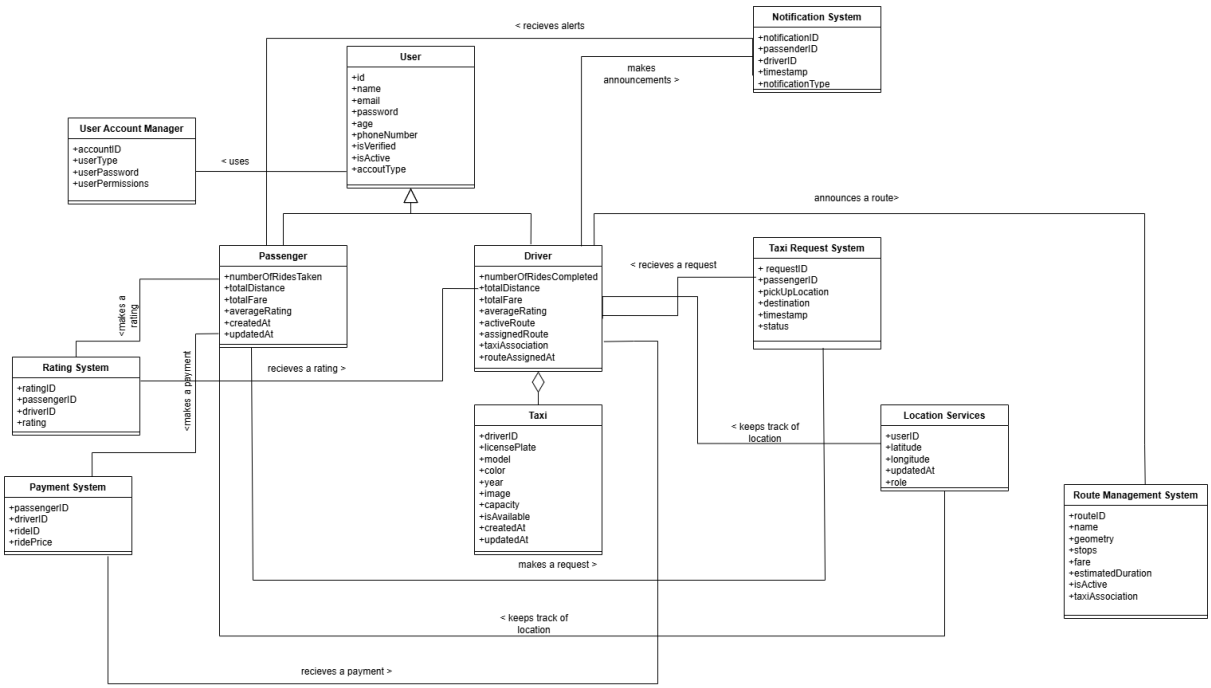
```
{
  message: string
}
```

Example:

```
{
  "message": "Ride declined by driver."
}
```

- **Effect:**
  - Finds the ride by `rideId`.
  - Validates that the driverId belongs to the driver assigned or available for the ride.
  - Marks the ride as declined or removes the driver from the list of available candidates.
  - May trigger reassignment of the ride to another driver or notify the passenger.

5 Domain Model



6 Functional Requirements

R1: User Account Management

- R1.1: Users should be able to register as either a driver or a passenger.
- R1.2: Users should be able to update their Profile information.
- R1.3: The system should support role-based access control for passenger and driver interfaces.
- R1.4: Users should be able to reset or change their passwords.

R2: Location Services

- R2.1: The system should track driver locations in real-time using GPS.
- R2.2: The system should determine passenger locations for pickup requests.
- R2.3: The system should calculate proximity between taxis and passengers.
- R2.4: The system should send proximity alerts to notify passengers when their requested taxi is approaching.
- R2.5: The system should display estimated time of arrival for approaching taxis.

R3: Taxi Request System

- R3.1: Passengers should be able to request taxi pickups based on their location.
- R3.2: Passengers should be able to see nearby available taxis.
- R3.3: Drivers should be notified of nearby passenger pickup requests.
- R3.4: Drivers should be able to accept or decline pickup requests.
- R3.5: Passengers should be able to specify their destinations.

R4: Route Management

- R4.1: The system should allow drivers to announce their routes.
- R4.2: The system should display taxi routes to passengers.
- R4.3: The system should allow drivers to indicate their destinations.
- R4.4: The system should support flexible drop-off points along routes.
- R4.5: The system should display route information in a visual format suitable for quick comprehension.

R5: Taxi Status Information

- R5.1: The system should display real-time taxi tracking showing vehicle location.
- R5.2: The system should show available seats in approaching taxis.
- R5.3: The system should allow drivers to update their seat availability status.
- R5.4: The system should indicate taxi status (en route, picking up, full, etc.).
- R5.5: The system should notify waiting passengers when taxis reach capacity.

R6: Notifications

- R6.1: The system should send push notifications for taxi proximity alerts.
- R6.2: The system should notify passengers when their requested taxi accepts or declines the pickup.
- R6.3: The system should notify drivers of new nearby passenger requests.
- R6.4: The system should provide ETA updates to waiting passengers.
- R6.5: The system should send notifications even with limited connectivity.
- R6.6: The system should allow users to customize notification preferences.

R7: Passenger Destination Management

- R7.1: The system should allow passengers to specify their drop-off locations.
- R7.2: The system should suggest optimal drop-off order to drivers.

R8: User Interface

- R8.1: The system should provide separate interfaces for passengers and drivers.
- R8.2: The system should offer a clean, easy-to-use interface with visual elements.
- R8.3: The system should support multiple South African languages.

R9: Rating and Feedback

- R9.1: Passengers should be able to rate drivers/taxis.
- R9.2: The system should collect feedback on routes and service.

R10: Fare Management

- R10.1: The system should calculate fare estimates based on route and distance.
- R10.2: The system should support both digital and cash payment options.
- R10.3: The system should provide payment confirmation receipts.
- R10.4: The system should track payment status for trips.

R11: Taxi Identification

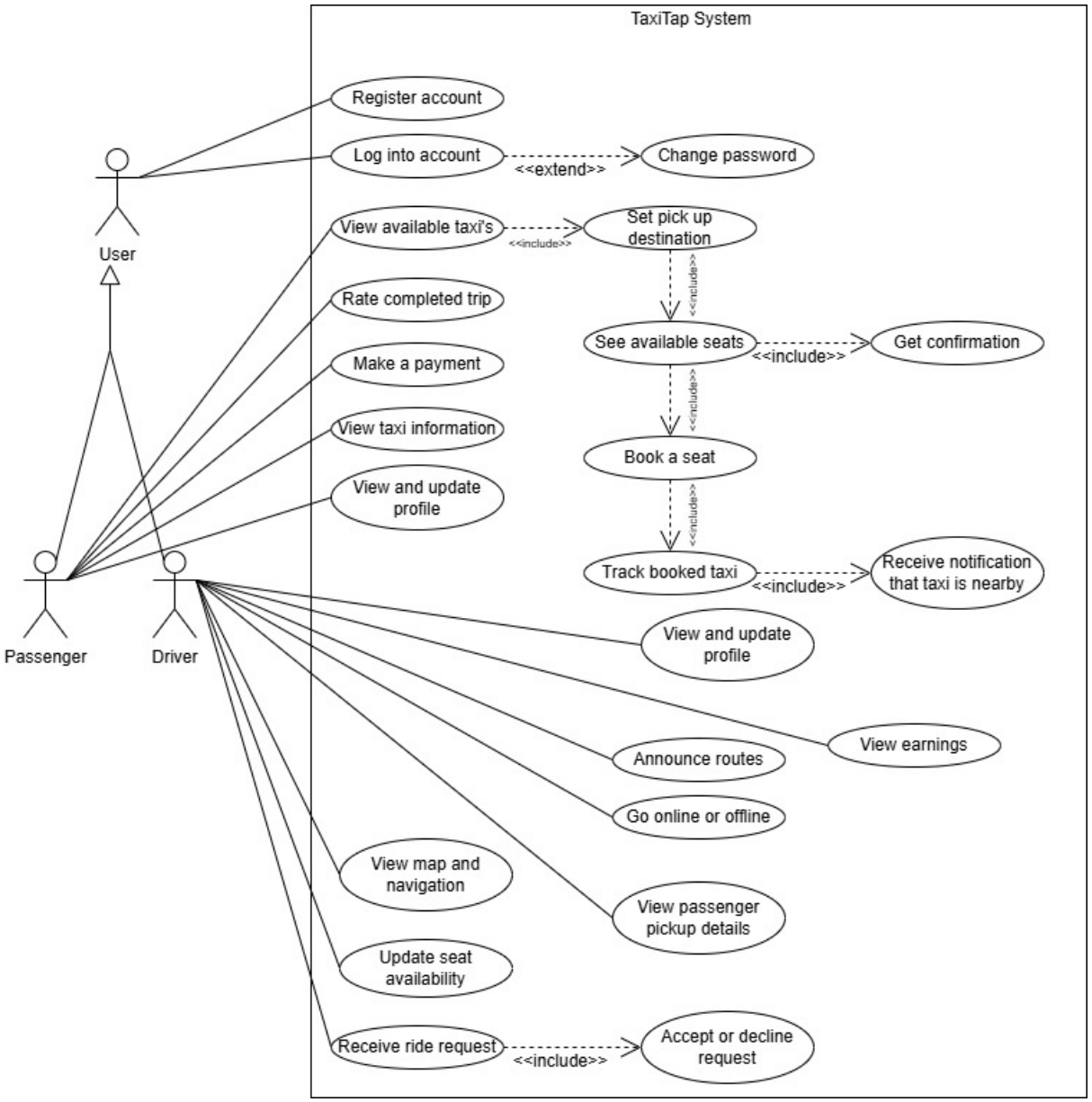
- R11.1: The system should provide unique identifiers for each taxi.
- R11.2: The system should support QR code-based taxi identification and verification.
- R11.3: The system should display taxi information (registration, operator) to passengers.

R12: Safety Features

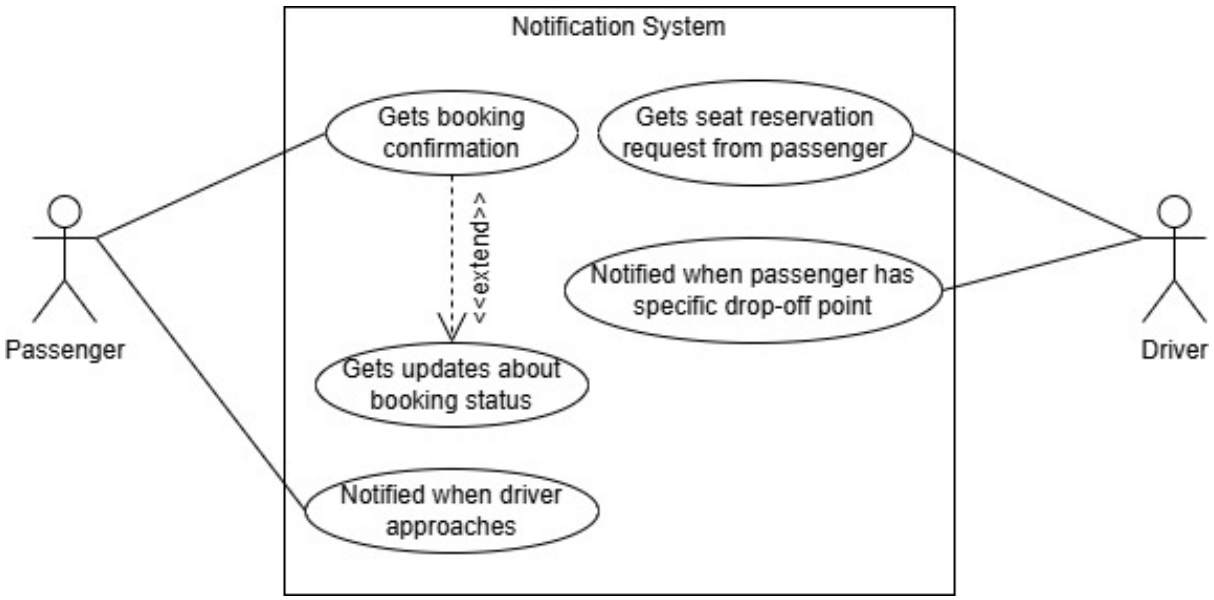
- R12.1: The system should provide an anonymous crime reporting tool.
- R12.2: The system should include emergency contact features.

7 Use Case Diagrams

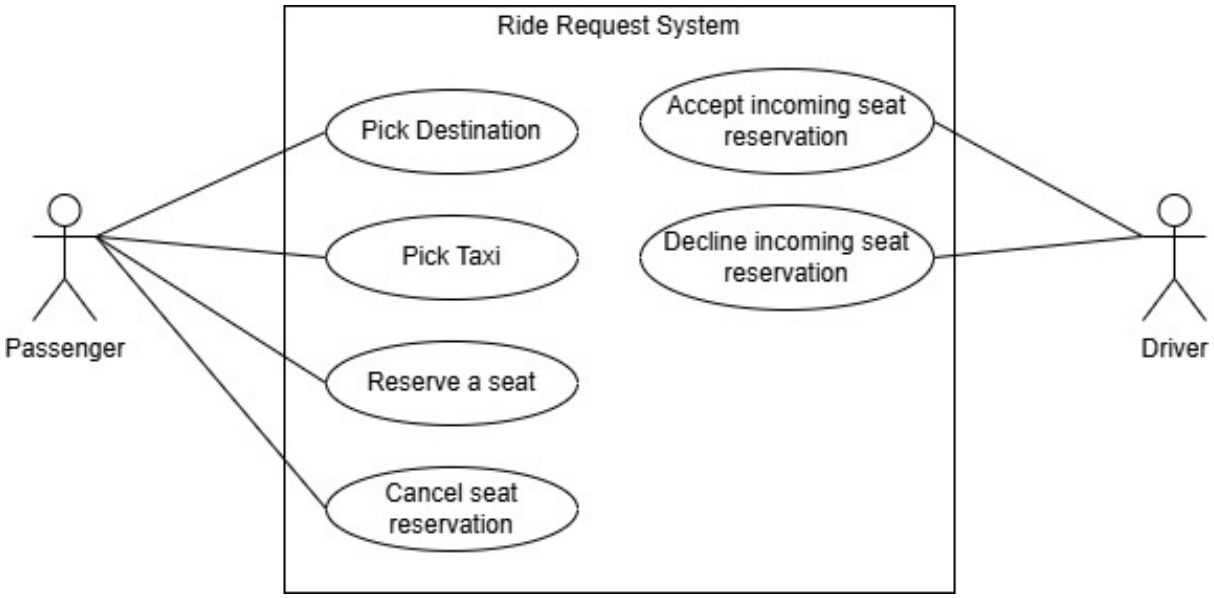
Overall System



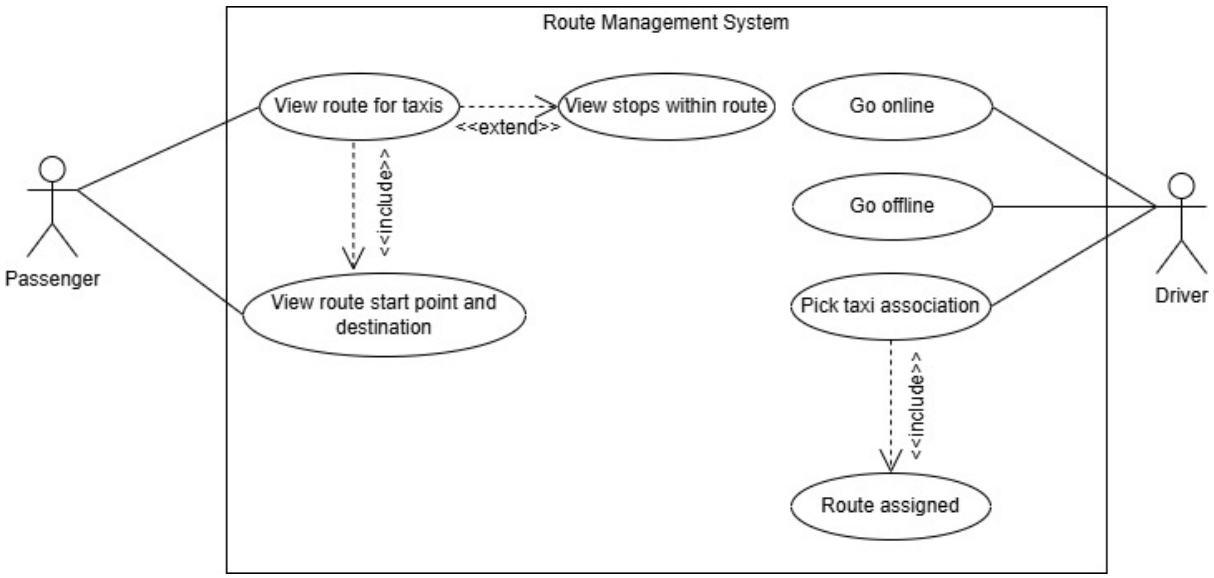
Notification System



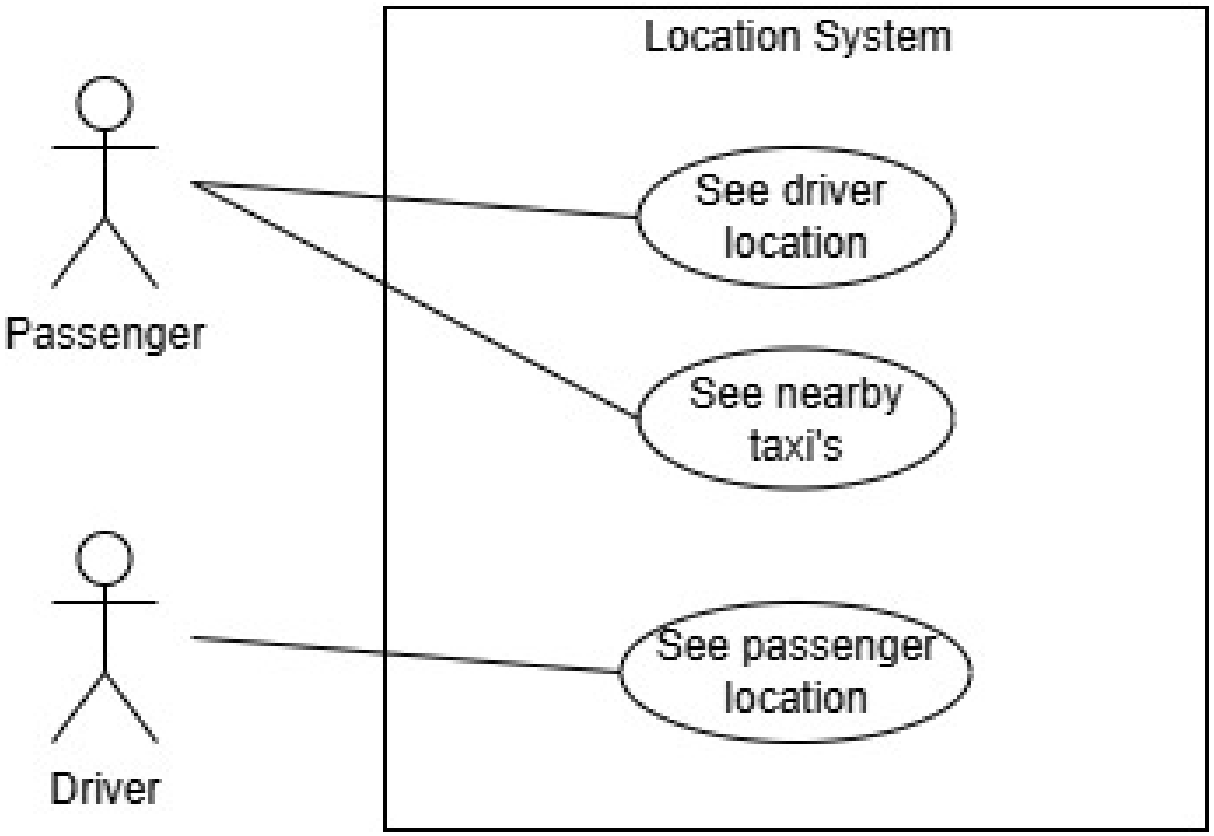
Ride Request



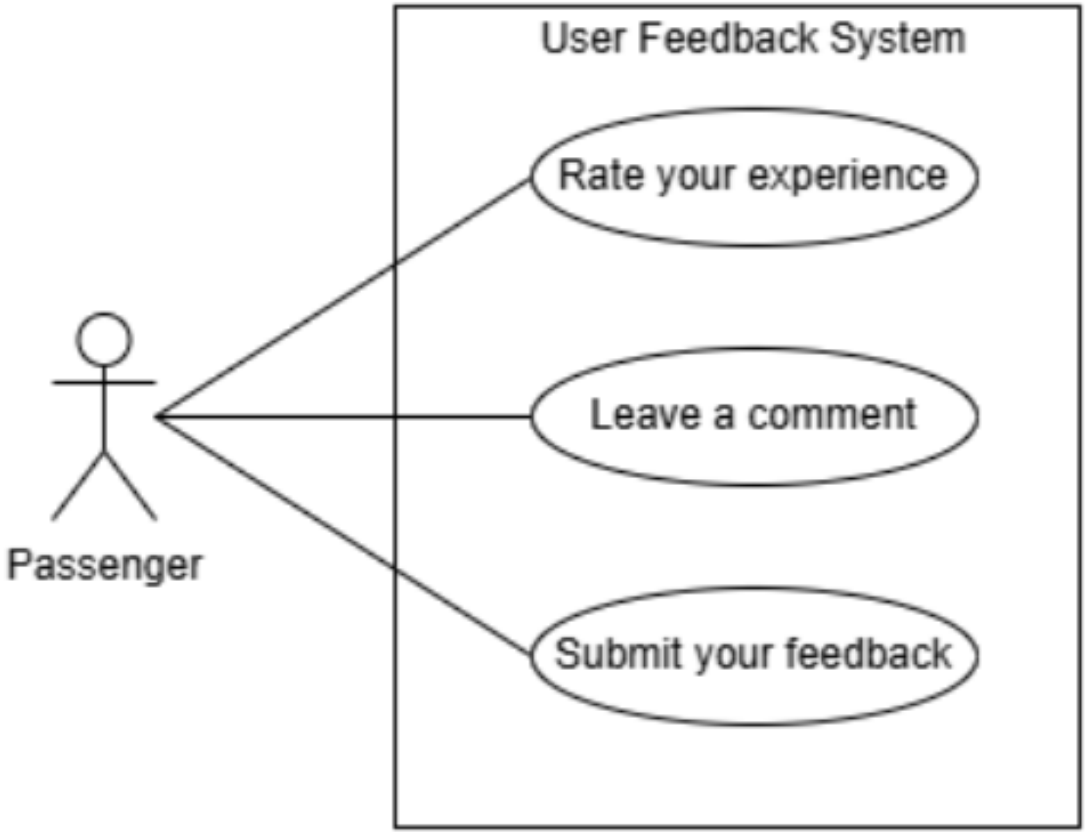
Route Management System



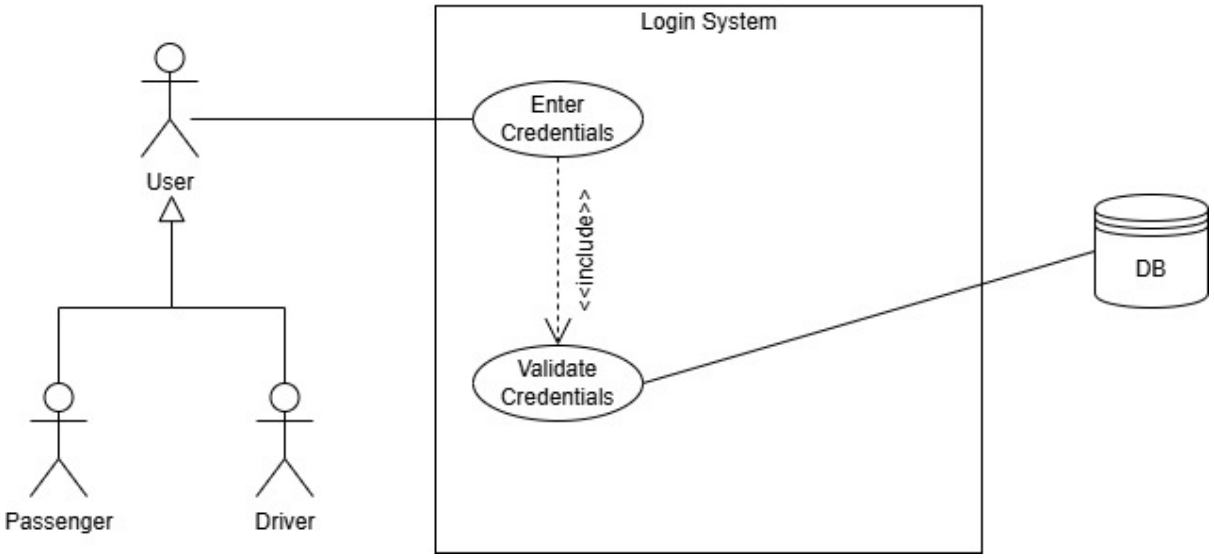
Location System



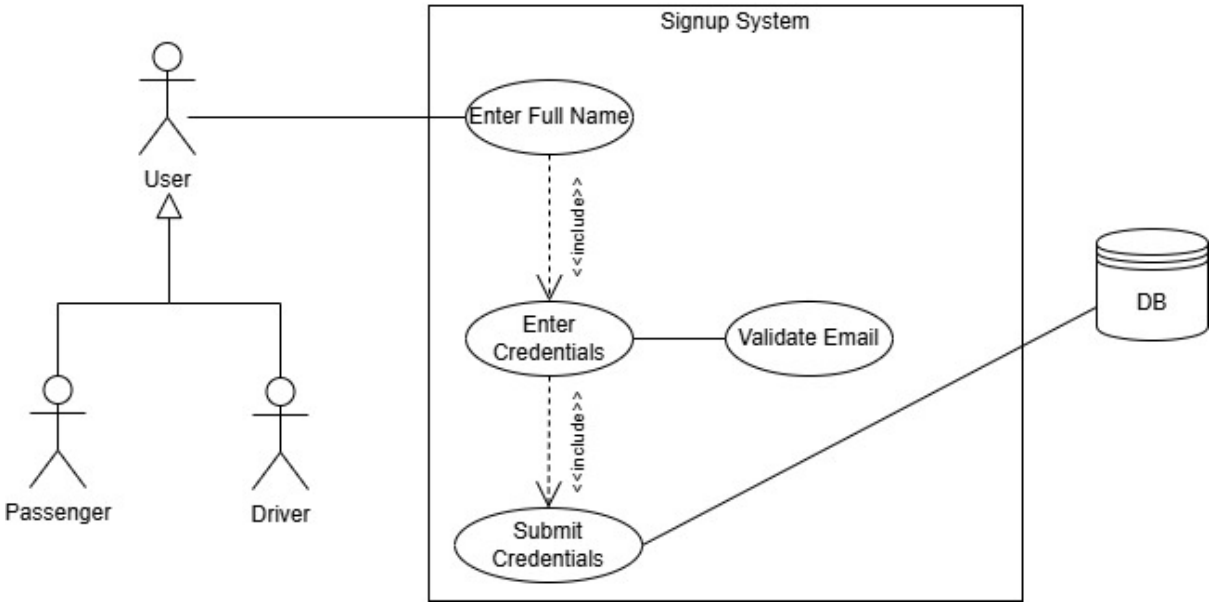
Feedback System



Login System



Signup System



8 Technology Requirements

8.1 Frontend

Expo (React Native with TypeScript)
Why we chose to use Expo?

- **Cross-platform Compatibility:** Code once, deploy to both Android and iOS.
- **Native Features:** Access to GPS, accelerometer, push notifications, offline storage, camera, QR scanning, etc.
- **Web Support:** Leverages Expo Web for rendering web-based dashboards and admin panels.
- **Live Reloading & Fast Iteration:** Expo Go provides hot reloading and rapid prototyping with a unified development experience.
- **Battery & Data Optimization:** React Native ecosystem provides fine-grained control over performance, reducing overhead.

8.2 Backend

Convex (TypeScript)
Why we chose to use Convex?

- **Truly Serverless:** No provisioning, no scaling headaches. Functions, database, and auth all run in one integrated environment.
- **Built-in Database:** Convex provides a powerful document-oriented database that supports relations, IDs, indexes, and real-time reactivity.

- **Type Safety:** Schema definition is in TypeScript, ensuring end-to-end type safety from backend to frontend.
- **Zero DevOps:** No need to manage infrastructure or containers. Deploy directly from your project.
- **Realtime Sync:** Built-in support for reactive queries allows passengers to see live taxi updates, seat availability, and ETA.

Convex Database Architecture

- **Document Store:** Convex uses collections of JSON-like documents, like MongoDB, but with built-in schema validation.
- **Indexes:** Automatic indexing on IDs and custom indexing for optimized query performance.
- **Relationships:** You can use Convex `v.id()` to reference documents between tables, ensuring referential integrity.
- **Realtime Subscriptions:** Query results update automatically when the underlying data changes.

Convex Free Tier (as of 2025)

- Compute: Up to 1 million function calls/month.
- Storage: 1 GB document data storage.
- Bandwidth: 5 GB of egress.
- Authentication: Integrated with third-party auth providers (Firebase Auth, Clerk, etc.).
- Deployment: 1 Production Deployment and 1 Dev Deployment per project.

Perfect for COS 301: Within budget, no surprise bills, and production-grade scalability.

8.3 Key Functional Modules & Implementation Plan

User Management Subsystem

- Authentication: Convex Auth with Clerk or Firebase integration.
- Registration/Login: Role-based registration (passenger or driver) with schema enforcement.
- Profile Updates: Mutation to update user document with profile fields.
- Security: JWT-based session validation, encryption at rest and in transit.

Location Services Subsystem

- Driver Location: Periodic GPS updates using Expo Location API.
- Passenger Location: One-time or continuous tracking during trip.
- Proximity Alerts: Triggered from Convex using background function.
- ETA Calculation: Naive approach using Haversine distance + average speed (no Google Maps API due to cost).

Taxi Request Subsystem

- Request Workflow:
  - Passenger sends request with coordinates and optional destination.
  - Nearby drivers notified (push notification via Expo).
  - Driver accepts or rejects request.
  - Status changes handled in real time.

Route Management Subsystem

- Driver Route Declaration: Input form for common route + destination.
- Passenger View: Map view of taxis on route + destinations.
- Optimized Routing (Optional): Historical route optimization using stored patterns (stretch goal).

Notification System

- Technology: Expo Notifications API.
- Use Cases:
  - Taxi is approaching.
  - Ride accepted or declined.
  - Route changes or delays.
- Offline Support: Caching notifications locally using AsyncStorage.

Safety and Fare Management Subsystem

- QR Identification: QR codes linked to taxi documents in Convex.
- Reporting: Anonymous incident reports saved to a secure Convex table.
- Fare Estimate: Static fare matrix per route (e.g., km-based fare slabs).
- Payment: Optional - integrate with SnapScan/Yoco for digital payments.

8.4 Testing Frameworks

- Backend: Jest (unit and integration tests for Convex functions).
- Frontend: React Native Testing Library.
- Manual Testing: Device tests using Expo Go and emulators.

8.5 CI/CD

- Convex Deployment: Triggered via GitHub Action or manual `npx convex dev` / `convex deploy`.
- Expo Deployment: Use `eas build` + `eas submit` for App Store/Play Store releases.
- Linting & Tests: Pre-commit lint checks with ESLint + Jest unit tests.

8.6 Version Control

- GitHub repo with main and dev branches.
- Feature branches for each core module.

9 Architectural Requirements

Architectural Requirements Document Here

9.1 Quality Requirements

Quality requirements determine the overall quality of Taxi Tap by specifying criteria that define how well the system performs and behaves.

1. **Security**
    - **Encryption:** All data must be encrypted in transit and at rest using the best security practices.
    - **Compliance:**
        - Data capturing and storing must adhere to the POPI act.
        - Ensure data privacy and consent handling.
    - **Secure authentication:** Users must authenticate securely, and sessions must be protected.
2. **Usability**
    - **Simplicity:** The interface should be easy to use for people with varying levels of tech literacy.
    - **Accessibility:** The use of clear labels, large tap targets and minimal steps to complete key tasks.

- **Feedback and error handling:** Provide real-time feedback for user actions, loading states and clear error messages when issues occur.

3. **Scalability**
   - The backend must scale to handle fluctuations in user or data load without performance degradation. This is automatically done by our chosen backend.
4. **Performance**
   - **Low bandwidth optimization:** The system must perform reliably under low-bandwidth or intermittent connectivity.
   - **Battery efficiency:** The app must minimize CPU, GPS and network usage to extend battery life.
5. **Reliability and Availability**
   - **Offline Support:** The app must function even without a constant internet connection, using local caching or data queuing mechanisms.
   - **High uptime:** The system should be available with minimal downtime to support driver operations throughout the day.
   - **Data integrity:** Ensure that data is not lost or duplicated during sync offline and online states.
6. **Maintainability and Extensibility**
   - **Clean architecture:** Backend and frontend systems should be modular and loosely coupled to allow easier updates, fixes, or feature additions in the future.
   - **Logging and monitoring:** Implement centralized logging and monitoring to quickly identify and resolve issues.
   - **Configurability:** Support code configurations without needing code changes.
7. **Affordability**
   - **Low data consumption:** The app must use data sparingly to remain cost-effective for users in regions with expensive or limited mobile data.
   - **Resource efficiency:** The system should minimize server and client-side consumption to reduce infrastructure and battery costs.

9.2 Design Patterns

Observer Pattern

- Pattern Type: Behavioural
- Participants:
  - Subject: Notification System
  - Observer: User
  - Concrete Observer: Passenger, Driver

- Explanation: The Observer pattern allows an object (User) to be notified automatically of state changes in another object (Notification System). This is ideal for handling events like route updates or ride status.
- Example:
  - User receives alerts from the Notification System.
  - Notification System initiates a notification when a route is announced.

Mediator Pattern

- Pattern Type: Behavioural
- Participants:
  - Mediator: Taxi Request System
  - Colleague: Passenger, Driver
- Explanation: The Mediator pattern centralizes complex communication between objects. Instead of Passenger directly interacting with Driver, requests are handled through the Taxi Request System.
- Example:
  - Taxi Request System acts as an intermediary between Passenger and Driver.
  - Passenger makes a request for pickup to a driver, but the Taxi request system acts as the middleman for this request.

9.3 Constraints

The client laid out the following constraints, by which Taxi Tap must abide, in their specification.

1. **All data must be encrypted at transit and at rest**
   All data exchanged between the mobile application and backend services will be encrypted using HTTPS with TLS (Transport Layer Security). Role-based access policies and authentication mechanisms (e.g., JWTs) ensure only authorised users can access specific system resources.
2. **POPI act**
   To ensure we abide by this, we will not collect any user data that is not necessary for the functionality of the app. With that, we will have permission set up to ensure that users are comfortable with collecting info, such as the user's location. Furthermore, we will consider providing a Terms and Conditions for the app that lays out how user data will be used.
3. **The app must function with low bandwidth, low data usage and be battery-efficient**
   We will accomplish this by having a UI that does not use too many resources and lightweight calls to the API.

4. **Budget**
   We must use AWS Free Tier platforms or any platforms that are open source or within free tier allowance.

10 Deployment Model

The system will be deployed following a cloud-based deployment model to ensure scalability, availability, and ease of access. The deployment environment will include:

- **Frontend:** Deployed via Expo Go.
- **Backend:** Hosted on Convex cloud (Convex backend as a service).
- **Database:** Managed by Convex (integrated serverless database).
- **CI/CD:** Managed via GitHub Actions to automate linting, testing and deployment.

11 Live Deployed System

A fully functional live version of the system will be accessible for demonstration purposes. This deployed system will allow real-time interaction by both drivers and passengers via mobile devices. The system will include:

- User registration and authentication.
- Route selection and reservation.
- Real-time updates.
- Payment simulation (optional).
- Feedback and rating system.
- Switch between passenger and driver.