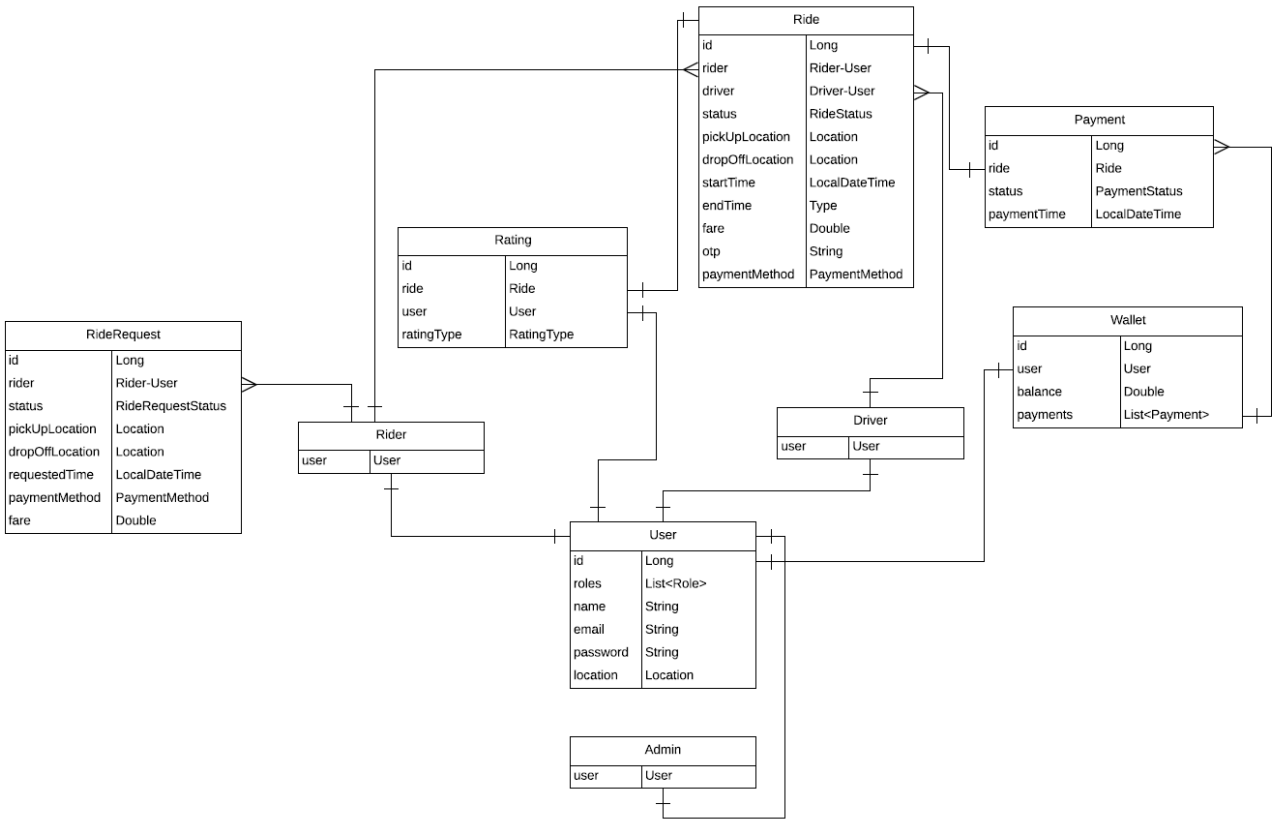
Ride-Hailing System Overview (Based on the UML Diagram)

This system models a ride-hailing service (like Uber), with users, ride requests, trips, payments, wallets, and ratings. Below we explain each part in simple terms.



1. Entities and Attributes

- **User**: Represents a person with an account. Key fields include:
- *id*: a unique number.
- *name, email, password*: the user's login info.
- *location*: the user's current location (e.g. GPS coordinates or address).
- *roles*: a list of roles this user has (for example, **Rider**, **Driver**, or **Admin**). One person can have multiple roles [1].

- **Rider**: A user acting as a passenger. The Rider entity simply holds a reference to a User account. (In other words, every Rider object "belongs" to one User.) It has no extra fields beyond linking to User.

- **Driver**: A user acting as a driver. Like Rider, it just links to one User account. A User who is a driver would have a Driver record pointing back to their User.

- **Admin**: A system administrator account. It links to a User. Admins manage the system (approve drivers, view all rides, etc.).

- **RideRequest**: Represents a ride booking request a Rider makes. Fields include:

- *id*: unique request number.
- *rider*: reference to the Rider (and thus the User) who requested the ride.
- *status*: current state of the request (e.g. "REQUESTED", "CANCELLED").
- *pickUpLocation* and *dropOffLocation*: where the rider wants to be picked up and dropped off (both are of type Location, which would contain coordinates or address).
- *requestedTime*: when the ride was requested.
- *paymentMethod*: how the rider will pay (such as by card, cash, or wallet).
- *fare*: the estimated cost for the ride.

This entity is created when the rider books a trip and holds all the booking details.

- **Ride**: Represents an actual trip after a driver accepts a RideRequest. Fields include:
- *id*: unique trip number.
- *rider* and *driver*: links to the Rider and Driver involved in this trip.
- *status*: current trip state (like "ACCEPTED", "IN_PROGRESS", "COMPLETED").

- *pickUpLocation* and *dropOffLocation*: same as the request (copied over at start).
- *startTime* and *endTime*: timestamps for when the trip began and ended.
- *fare*: final cost for the trip (may include surcharges or changes).
- *otp*: a one-time password/code used at pickup to confirm the correct passenger.

- *paymentMethod*: again records how this ride will be paid.

- **Payment**: Represents a payment transaction for a completed ride. Fields include:

- *id*: unique payment number.
- *ride*: link to the Ride being paid for.

- *status*: e.g. “PENDING”, “COMPLETED”, or “FAILED”.
- *paymentTime*: when the payment was made.

Each completed Ride generates one Payment record once the fare is charged.

- **Wallet**: An in-app account balance for a User. Fields include:
- *id*: unique wallet number.
- *user*: link to the owning User.
- *balance*: current stored money (a dollar amount).
- *payments*: a list of Payment records that have used this wallet.

Each User has one Wallet. Riders can add money to their Wallet and the system deducts from it when they pay for rides [2] [3].

- **Rating**: Captures feedback after a trip. Fields include:
- *id*: unique rating number.
- *ride*: link to the Ride being rated.
- *user*: the User who gave the rating (e.g. the rider rating the driver, or vice versa).
- *ratingType*: the actual rating value or type (for example 1–5 stars).

Each Rating belongs to one ride and one user (the rater). In practice, a user can give many ratings over time (one per ride) [4].

(Other types in the diagram: “Role” would list roles like Rider/Driver/Admin; “PaymentMethod” might be CARD, CASH, or WALLET; “RideStatus”, “PaymentStatus”, “RatingType” are pre-defined categories for those fields.)

2. Relationships Between Entities

The diagram’s arrows and references describe how these entities link:

- **User – Rider/Driver/Admin**: Each Rider/Driver/Admin record is tied to exactly one User (one-to-one). In practice, one User account *can* serve multiple roles (for example, a person can be both a Rider and a Driver) by having different Rider or Driver entries that point to the same User [1].

- **User – Role**: A User has a list of Roles (like “rider” or “driver”). This is many-to-many (a user can have multiple roles, and each role applies to many users). For example, the UML shows `roles List<Role>` for User [1].

- **User – Wallet**: One User has exactly one Wallet (one-to-one), linking them to their in-app balance.

- **Wallet – Payment**: One Wallet can link to many Payment records (one-to-many). Each time the user pays with the wallet, a Payment entry is added to their wallet's payment list.

- **Rider – RideRequest**: One Rider (user) can make many ride requests over time (one-to-many). Each request entry belongs to the rider who made it.

- **RideRequest – Ride**: Typically one RideRequest leads to one Ride when a driver accepts it (one-toone). If a request is never matched or is cancelled, a Ride may not be created.

- **Ride – Driver**: Each Ride has one assigned Driver (one-to-one).

- **Ride – Rider**: Each Ride has one Rider (the passenger) (one-to-one).

- **Ride – Payment**: Each Ride generates one Payment (one-to-one) after completion.

- **Ride – Rating**: A Ride can have ratings given by participants. In this model, each completed Ride usually gets one rating from each party. We assume at least the rider rates the driver; so practically a ride can have one or two ratings.

- **User – Rating (corrected)**: Although the diagram shows Rating User as 1-to-1, it should be **manyto-one**: one User can give many Ratings (one per ride) over time. That is, each rating has one user (the rater), but each user can appear in many Rating records [4]. (For example, Alice may rate 100 different drivers over many rides – 100 ratings all link back to Alice's user account.) [4]

3. System Flow (Ride Lifecycle)

Here is how a typical ride progresses through the system:

1. **Requesting a Ride:** A user (as a Rider) opens the app and books a trip by specifying pickup and drop-off locations and maybe a payment method. The system creates a new **RideRequest** record with status *REQUESTED*, storing the rider's ID, pickup/dropoff locations, requested time, chosen payment method, etc.

2. **Matching a Driver:** The system looks for an available **Driver** nearby and notifies them. When a driver accepts, a **Ride** record is created, linking that driver and the rider. The ride status goes to *ACCEPTED*. (In some designs, there is a separate match/dispatch service – but overall the RideRequest transitions into an active Ride.) *As one reference notes, "when a rider books a ride, the RideRequestService logs the request and triggers driver-matching logic; accepted rides move into the Ride service"* [5] .

3. **Starting the Trip:** Before pickup, the system may generate a one-time passcode (OTP). When the driver arrives, the rider shows this OTP to confirm their identity and ensure it's the correct person. Then the driver marks the ride as *IN_PROGRESS*, and the system records the startTime in the Ride record.

4. **Completing the Trip:** Once the rider is dropped off at the destination, the driver marks the ride as completed. The system sets the Ride's status to *COMPLETED* and records the endTime. Now the trip details (distance, duration, any extra fees) are finalized.

5. **Payment:** The fare is calculated (typically based on distance, time, surge multipliers, etc.) and a **Payment** is processed.

6. If the rider chose to use their in-app **Wallet**, the system *automatically deducts* the fare from their wallet balance [3] [2]. (For example: "the fare is automatically deducted from your wallet balance… the ride ends, and you're on your way without any payment delays" [3].)

7. Otherwise, the system charges the rider via the selected method (credit card, PayPal, etc.). In either case, a Payment record is created with status *COMPLETED* once the charge succeeds. (As one guide explains, "Fare is calculated based on time, distance, surge, etc. *Payment is processed automatically at the end*" [6] . And the Payment Service "processes completed-ride payments (via credit card, Stripe, etc.)" [7] .)

8. **Rating:** After payment, the rider and driver have the opportunity to rate each other. For example, the rider rates the driver (and/or vice versa) on a scale. When the rider submits a rating, the system creates a **Rating** record linking that rating to the ride and the rating user. These ratings are stored for each trip. *"After a ride, riders can rateDriver and drivers can rateRider"* [4] . Each rating points to the trip and the user who gave it. Over time, one user will appear in many Rating records (one per ride they gave feedback for).

Throughout this flow, the system updates each entity's status fields (like RideRequest.status or Ride.status) and timestamps (start/end time). The Admin role may oversee these operations (for instance, an admin can view all rides or suspend a driver).

In summary, the UML entities work together as follows: users (with roles) create ride requests, drivers are matched to requests creating rides, rides are tracked from start to finish, payments are handled (often with an internal wallet) [6]2 , and finally rides are rated by users [4]. This covers the complete trip lifecycle in clear, step-by-step terms.

Sources: The explanation above is guided by the given UML diagram and common ride-hailing system designs (e.g. Uber) [1] [5] [6] [2] [4] [3]. Each cited reference provides background on user roles, payment processing, and rating flows in ride-sharing apps.