ChatGPT

Uber Backend – Rating and Driver Onboarding Implementation

Data Structures (DTOs and Entities)

- **RatingDto:** A simple Lombok-annotated class holding `rideId` and `rating` (both fields with getters/setters from `@Data`). This DTO carries rating info in controller requests and responses.
- **Rating Entity:** Annotated with `@Entity`, it records ratings for each ride. It has an `id` (auto-generated), a one-to-one link to a `Ride` (`@OneToOne`), and many-to-one links to `Rider` and `Driver` (`@ManyToOne`). This means each **Rating** is tied to exactly one ride, but many ratings can reference the same rider or driver [1] [2]. The entity has two fields: `driverRating` (rating given to the driver) and `riderRating` (rating given to the rider). All boilerplate (getters/setters, constructors) is generated by Lombok (`@Setter @Getter @NoArgsConstructor @AllArgsConstructor @Builder`) – for example, `@Data` (from Lombok) automatically creates getters, setters, `toString()`, and other methods [3], and `@Builder` lets us fluently construct new objects without writing verbose code [4].
- **Indexes:** Each entity's `@Table` includes `@Index` annotations on frequently queried columns (e.g. foreign keys like `rider_id`, `driver_id`, and unique fields like `email` or `vehicleId`). In JPA, using `@Index` improves query performance by creating database indexes on those columns [5]. For example, indexing `rider_id` speeds up queries finding ratings or rides by a given rider, which is crucial for performance in a large system.

Repositories

- **RatingRepository:** Extends `JpaRepository<Rating,Long>`, giving CRUD operations. It declares methods like `List<Rating> findByRider(Rider rider)` and `findByDriver(Driver driver)`, and `Optional<Rating> findByRide(Ride ride)`. Spring Data JPA automatically implements these by parsing the method names and generating the corresponding queries [6]. For instance, `findByRider(Rider rider)` will fetch all Rating records where the `rider` field matches. Using derived query methods this way lets us quickly load all ratings for a specific user without writing SQL.

Service Layer

RatingService (Interface and Implementation)

- **Interface:** `RatingService` defines three key methods: `rateDriver(Ride ride, Integer rating)`, `rateRider(Ride ride, Integer rating)`, and `createNewRating(Ride ride)`. These encapsulate the logic for recording a new rating given to a driver or rider, and initializing rating records when a ride is created.

- **Implementation (`RatingServiceImpl`):**

- `rateDriver`:

  1. Retrieves the `Driver` from the given `Ride`.
  2. Fetches the existing `Rating` object for that ride (`ratingRepository.findByRide(ride)`). If none exists, throws a `ResourceNotFoundException`.
  3. Checks if `driverRating` is already set; if so, throws a runtime error to prevent double-rating.
  4. Sets the new `driverRating` on the `Rating` entity and saves it.
  5. Recomputes the driver's overall average rating by retrieving *all* ratings for that driver (`findByDriver(driver)`), mapping each `Rating` to its `driverRating`, and computing the average via Java Streams. It uses `mapToDouble(Rating::getDriverRating)` and `average()` (returning `0.0` if no ratings exist) [7].
  6. Updates the `Driver` entity's `rating` field with this new average and saves the driver.
  7. Returns a DTO (`DriverDto`) of the updated driver using ModelMapper.

  This approach ensures that each time a rider rates a driver, the system updates the driver's average rating based on all past feedback. The use of `mapToDouble(...).average()` encapsulates "add all driver ratings and divide by count" logic in a concise way [7].

- `rateRider`: Symmetrically handles when a driver rates a rider. It fetches the `Rider`, finds the `Rating` by ride, ensures `riderRating` is not already set, assigns the new rating, and saves. Then it recomputes the rider's average rating from all records (`findByRider(rider)`) using the same stream-average pattern and updates the `Rider` entity's `rating`. Finally returns `RiderDto`.

- `createNewRating`: (Stubbed here) would be called when a new ride is started or finished to create an empty `Rating` record linking that ride, rider, and driver. This ensures a `Rating` entry exists before either party can rate.

Throughout, ModelMapper is used to convert entities to DTOs (e.g. `DriverDto`, `RiderDto`). ModelMapper is a library that automatically maps one object to another (entity to DTO) based on matching field names [8], simplifying our code so we don't write boilerplate copy logic.

DriverService / RiderService Integration

- **DriverService.rateRider (in `DriverServiceImpl`):** Called when a driver submits a rating for the rider. It checks security (via a hypothetical `getCurrentDriver()` to ensure the logged-in user is indeed the ride's driver), fetches the `Ride`, ensures the ride status is `ONGOING` (meaning completed – if not, it rejects rating), then delegates to `ratingService.rateRider(ride, rating)`.
- **RiderService.rateDriver (in `RiderServiceImpl`):** Similarly, when a rider rates a driver. It gets the current rider, validates they match the ride's rider, checks ride status, then calls `ratingService.rateDriver(ride, rating)`.

These checks enforce that only the correct participant can give the rating and only once the ride has ended. Errors like "not owner of ride" or "already rated" are thrown as runtime exceptions.

REST Controllers

DriverController Endpoints

- `POST /driver/cancelRide/{rideId}` (note: the code snippet shows a missing `{}`, but the intention is to use `@PathVariable`): Cancels an ongoing ride by ID. Returns updated `RideDto`.
- `POST /driver/rateRider`: Expects a JSON `RatingDto` with `rideId` and `rating`. Calls `driverService.rateRider(rideId, rating)` and returns the updated `RiderDto`.
- `GET /driver/getMyProfile`: Returns the logged-in driver's profile as `DriverDto`.
- `GET /driver/getMyRides`: Returns a paginated list of the driver's rides (`Page<RideDto>`). It reads `pageOffset` and `pageSize` from query parameters (using defaults if absent), constructs a `PageRequest`, and calls `driverService.getAllMyRides(pageRequest)`. This follows the common Spring pattern of injecting pagination parameters with `@RequestParam` and using `PageRequest` [9].

In all controller methods, responses are wrapped with `ResponseEntity.ok(...)` or `ResponseEntity.status(...)`. Using `ResponseEntity.ok(body)` is a standard way to return HTTP 200 with a JSON body [10].

RiderController Endpoints

- `POST /rider/cancelRide/{rideId}`: Similar to driver's cancel, calls `riderService.cancelRide`.
- `POST /rider/rateDriver`: Takes `RatingDto` from body, calls `riderService.rateDriver`, and returns `DriverDto`.
- `GET /rider/getMyProfile`: Returns `RiderDto` of current user.
- `GET /rider/getMyRides`: Paginated list of rides for the rider, similar to the driver's version.

These controllers handle incoming HTTP requests, delegate to the service layer, and return appropriate DTOs wrapped in HTTP responses. The annotations like `@PostMapping` and `@GetMapping`, along with `@RequestBody` and `@PathVariable/@RequestParam`, are standard Spring MVC for REST endpoints.

AuthController – Onboarding a New Driver

- `POST /auth/signup`: Registers a new user (not shown in detail).
- `POST /auth/onBoardNewDriver/{userId}`: Takes a path variable `userId` and a JSON body `OnboardDriverDto` (with `vehicleId`). In the `AuthController`, this calls `authService.onBoardNewDriver(userId, vehicleId)` and returns the created `DriverDto` with HTTP 201 Created.

AuthService.onBoardNewDriver (Implementation)

1. **Lookup User:** Finds the `User` entity by ID. If not found, throws `ResourceNotFoundException`.
2. **Role Check:** Ensures the user is not already a driver (e.g., doesn't already have a `DRIVER` role); otherwise throws conflict.
3. **Create Driver Entity:** Builds a new `Driver` with Lombok's `@Builder`, setting fields: link to the user, `rating` = 0.0, provided `vehicleId`, and `available = true`.

4. **Update User Role:** Adds the `DRIVER` role to the user's roles set, then saves the user.
5. **Save Driver:** Calls `driverService.createNewDriver(createDriver)`, which saves the driver entity to the database.
6. **Return DTO:** Uses ModelMapper to convert the saved `Driver` to `DriverDto`.

This allows an existing user to become a driver by providing their vehicle details. We use the builder pattern and Lombok to keep code concise [4], and ensure transactional consistency by updating both user and driver records.

Database Indexing (Performance Optimization)

At the end, indexes were added to many tables via JPA's `@Index` annotation inside `@Table` on each entity: - **Driver:** Index on `vehicleId`. - **Rating:** Indexes on `rider_id` and `driver_id`. - **Ride:** Indexes on `rider_id` and `driver_id`. - **RideRequest:** Index on `rider_id`. - **User:** Table `uber_user` has index on `email`. - **WalletTransaction:** Indexes on `wallet_id` and `ride_id`.

These indexes mean the database will keep sorted copies of those columns, greatly speeding up queries that filter by these fields. For example, finding all rides by a given driver, or looking up a user by email (common during login), now uses an index scan instead of a full table scan. As Baeldung explains, indexes "improve the speed of data retrieval operations on a table at the cost of additional writes and storage space" and should be added on columns that are frequently queried [5]. In a ride-sharing app, foreign-key fields (like `rider_id` on a ride) and unique identifiers (like user email or vehicle ID) are prime candidates for indexing.

Summary of Changes

- **Rating Feature:** Introduced a new `RatingDto` and `Rating` entity to allow mutual rating between riders and drivers. Implemented service methods to save ratings, prevent duplicate ratings, and automatically update the average rating of each user.
- **Service Checks:** Added validation in services to ensure only the correct party (driver vs. rider) can rate, and only after a ride has completed.
- **Controllers:** Exposed new endpoints (`/rateRider` for drivers and `/rateDriver` for riders) accepting `RatingDto`. Updated Driver and Rider controllers to wire these endpoints to the corresponding service calls.
- **Driver Onboarding:** Added a new `OnboardDriverDto` and endpoint (`/onBoardNewDriver`) in the AuthController. The AuthService logic now creates a `Driver` record, assigns the DRIVER role to the user, and returns the new driver profile.
- **Model Mapping:** Used ModelMapper to convert between entity and DTO objects, reducing boilerplate mapping code [8].
- **Database Optimization:** Added JPA-managed indexes on key columns across entities (drivers, rides, users, ratings, etc.) to boost query performance in the backend [5].
- **Design Patterns:** Leveraged Lombok annotations (`@Data`, `@Builder`) to minimize boilerplate code [3] [4], and Spring Data JPA's derived query methods (`findByX`) to simplify data access [6].

Overall, these changes implement the core rating flow in the Uber-style backend and prepare the system for efficient data access and future features. Each code section works together: Controllers handle HTTP details, services enforce business rules and calculations, repositories fetch/persist data, and entities/DTOs

define the data structures. The indexing ensures the system scales by optimizing common lookups, and Lombok/ModelMapper keep the code clean and maintainable.

Sources: Official Spring and JPA documentation and tutorials were referenced for concepts like JPA mappings, repository methods, indexing, and Lombok usage [3] [6] [5] [7] [8] [10]. These provide background on the annotations and patterns used above.

---

[1] Hibernate - @ManyToOne Annotation - GeeksforGeeks
https://www.geeksforgeeks.org/java/hibernate-manytoone-annotation/

[2] One-to-One Relationship in JPA | Baeldung
https://www.baeldung.com/jpa-one-to-one

[3] A Complete Guide to Lombok
https://auth0.com/blog/a-complete-guide-to-lombok/

[4] Using Lombok's @Builder Annotation | Baeldung
https://www.baeldung.com/lombok-builder

[5] Defining Indexes in JPA | Baeldung
https://www.baeldung.com/jpa-indexes

[6] JPA Query Methods :: Spring Data JPA
https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

[7] Java Program: Calculate Average of Integers using Streams
https://www.w3resource.com/java-exercises/stream/java-stream-exercise-1.php

[8] Spring Boot - Map Entity to DTO using ModelMapper - GeeksforGeeks
https://www.geeksforgeeks.org/java/spring-boot-map-entity-to-dto-using-modelmapper/

[9] REST Pagination in Spring | Baeldung
https://www.baeldung.com/rest-api-pagination-in-spring

[10] Using Spring ResponseEntity to Manipulate the HTTP Response | Baeldung
https://www.baeldung.com/spring-response-entity