day 5

Today's work

```
package com.yasif.project.uber.Uber.backend.system.strategies;

import com.yasif.project.uber.Uber.backend.system.entities.RideRequest;

public interface RideFareCalculationStrategy {  6 usages  2 implementations  Yasif khan *

    double RIDE_FARE_MULTIPLIER = 10;  1 usage

    double calculateFare(RideRequest rideRequest);  1 usage  2 implementations  new *
}
```

✅ What this file represents

You've established a **Strategy Pattern contract** for fare calculation — a clean architectural maneuver to decouple business logic and enable plug-and-play pricing models as your platform scales.

🔍 Line-by-line breakdown

1. public interface RideFareCalculationStrategy

This is a **strategy interface** — a high-level abstraction that enforces a uniform contract for any fare-calculation algorithm.

In corporate terms:
You're creating a **pluggable business capability** that lets you iterate pricing logic without refactoring upstream modules.

2. double RIDE_FARE_MULTIPLIER = 10;

This is a **baseline constant** representing your default per-unit multiplier (could be per kilometer, per minute, or hybrid depending on your domain rules).

Key insight:
It acts as a foundational *pricing coefficient*, giving you centralized control over fare computation.

3. double calculateFare(RideRequest rideRequest);

This is the core method — a **behavioral contract**.

Any class implementing this interface must provide its own fare-calculation formula based on:

- distance
- time
- surge pricing
- driver category
- ride type (mini, sedan, luxury)
- demand/supply heuristics

This unlocks strategic extensibility:

- **StandardFareStrategy**
- **SurgeFareStrategy**
- **PremiumFareStrategy**
- **NightFareStrategy**

Each can implement this method differently while adhering to the same interface.

📌 What this achieves in your system

You're setting up:

- **High modularity**
- **Clean separation of concerns**
- **Algorithm-level agility**
- **Future-proof pricing architecture**

Corporate equivalent:
You're operationalizing a **strategy-driven calculation pipeline** that scales with product maturity.

```java
package com.yasif.project.uber.Uber.backend.system.strategies.Impl;

import com.yasif.project.uber.Uber.backend.system.entities.RideRequest;
import com.yasif.project.uber.Uber.backend.system.services.DistanceService;
import com.yasif.project.uber.Uber.backend.system.strategies.RideFareCalculationStrategy;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@RequiredArgsConstructor  no usages  Yasif khan *
@Service
public class RiderFareDefaultFareCalculationStrategy implements RideFareCalculationStrategy {


    private final DistanceService distanceService;


    @Override  1 usage  new *
    public double calculateFare( @NotNull RideRequest rideRequest) {
        double distance = distanceService.calculateDistance(rideRequest.getPickUpLocation(),
                rideRequest.getDropOffLocation());
        return distance*RIDE_FARE_MULTIPLIER;
    }
}
```

✅ Class Purpose (High-Level)

This class is your **default fare calculation strategy** — a concrete implementation of your Strategy Pattern. It operationalizes your baseline fare computation by leveraging a distance calculation micro-service.

In corporate terms:
You're institutionalizing a **pricing engine component** that converts geospatial inputs into monetary output using a standardized algorithm.

🔍 Line-by-Line / Concept Breakdown

1. @RequiredArgsConstructor

This Lombok annotation:

- Auto-generates a constructor for all final fields
- Enables **dependency injection** without boilerplate
- Ensures the class is immutable where possible

This reinforces clean DI principles and operational robustness.

2. @Service

Spring stereotype annotation marking this class as a **Spring-managed service bean**.

This positions it as a **deployable business capability** within your application's service layer.

3. public class RiderFareDefaultFareCalculationStrategy implements RideFareCalculationStrategy

This class **implements the strategy contract** you defined earlier.

Business takeaway:
You're delivering a concrete pricing model that can be swapped, extended, or overridden without service-level disruption.

4. private final DistanceService distanceService;

This service is responsible for:

- Computing real-time geospatial distance
- Likely leveraging Haversine, PostGIS, or location APIs

This dependency makes fare calculation **data-driven** and flexible.

Because it's final, constructor injection (via Lombok) guarantees reliability and lifecycle stability.

5. public double calculateFare(RideRequest rideRequest)

This is the strategic implementation of your pricing logic.

Workflow inside this method:

Step 1: Calculate distance

```
double distance = distanceService.calculateDistance(

        rideRequest.getPickUpLocation(),

        rideRequest.getDropOffLocation()

);
```

- Input: pickup and dropoff geolocations
- Output: numeric distance in KM (assuming your DistanceService is aligned to that)
- This establishes your **movement value** for billing

Step 2: Apply fare multiplier

```
return distance * RIDE_FARE_MULTIPLIER;
```

- Uses the constant defined in the interface (10)
- Computes **base fare = distance × fare coefficient**
- No surge, no time factor — this is your **MVP pricing baseline**

Corporate-speak:
This method orchestrates a **deterministic fare pipeline**, blending geospatial computation with domain pricing parameters.

📌 Why this design matters

You've set up a **strategic pricing architecture** that delivers:

✓ Scalability

Easily add new strategies (Surge, Night, VIP) without modifying core logic.

✓ Clean Separation

Distance and fare logic are isolated, enabling independent iteration.

✓ Extensibility

Switching strategies becomes a configuration-level decision, not a code change.

✓ Maintainability

Dependencies are injected cleanly, avoiding lifecycle complications.

```java
import com.yasif.project.uber.Uber.backend.system.entities.Driver;
import com.yasif.project.uber.Uber.backend.system.entities.RideRequest;
import com.yasif.project.uber.Uber.backend.system.repositories.DriverRepository;
import com.yasif.project.uber.Uber.backend.system.strategies.DriverMatchingStrategy;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;

import java.util.List;

// ST_Distance(point1,point2)
// ST_DWithin(point1,10000)

@RequiredArgsConstructor  no usages  Yasif khan *
@Service
@Primary
public class DriverMatchingNearestDriverStrategy implements DriverMatchingStrategy {

    private final DriverRepository driverRepository;

    @Override  1 usage  Yasif khan *
    public List<Driver> findMatchingDriver( @NotNull RideRequest rideRequest) {
        return driverRepository.findTenNearbyTopRatedDrivers(rideRequest.getPickUpLocation());
    }
}
```

Here’s a crisp, enterprise-grade breakdown of this strategy component — unpacking exactly what it represents, how it operates, and the architectural value it brings into your ride-matching pipeline.

✅ High-Level Purpose

This class is your **Nearest-Driver Selection Strategy**, responsible for identifying the best possible drivers around the rider’s pickup location.

Think of it as the **driver allocation engine**, aligned to spatial intelligence and operational prioritization.

🔍 Line-by-Line / Concept Breakdown

1. @RequiredArgsConstructor

This auto-generates a constructor for all final fields (i.e., driverRepository), enabling:

- Immutable dependencies
- Cleaner dependency injection
- A more predictable lifecycle

Corporate lens:
You're ensuring a high-integrity service instantiation pattern.

2. @Service

Marks this class as a Spring service bean — a pluggable, reusable **business capability** within your platform's matching layer.

3. @Primary

This is a strategic annotation.

When multiple DriverMatchingStrategy implementations exist in the system, @Primary ensures **this strategy becomes the default** unless explicitly overridden.

Practical outcome:
This strategy becomes the **baseline matching logic** for your ride allocation flow.

4. public class DriverMatchingNearestDriverStrategy implements DriverMatchingStrategy

You are implementing the DriverMatchingStrategy interface.

This aligns with your use of the **Strategy Pattern**, enabling:

- Swap-in / swap-out matching algorithms
- A future roadmap for surge-aware, category-based, or ML-driven matching

This class is your foundational algorithm.

5. private final DriverRepository driverRepository;

The class depends on data persistence operations for:

- Spatial queries
- Driver availability filtering
- Ranking logic

The repository is where your **PostGIS logic** will live.

6. public List<Driver> findMatchingDriver(RideRequest rideRequest)

This method operationalizes the actual driver-matching logic.

Let's break the responsibility chain:

Step 1: Extract pickup location

rideRequest.getPickUpLocation()

This returns a PostGIS Point representing the rider's geolocation.

Step 2: Query the database

return driverRepository.findTenNearbyTopRatedDrivers(rideRequest.getPickUpLocation());

This repository method is expected to perform:

- **Spatial filtering** using ST_DWithin() to limit results to <10km
- **Distance ordering** using ST_Distance()
- **Rating ordering** so top-rated drivers are prioritized
- **Limiting results** to 10 nearest drivers

In corporate terms:
This method orchestrates a **geospatial + quality-based candidate pipeline**.

🚀 What this strategy accomplishes

✓ Proximity-based matching

Drivers are selected based on actual geographic distance.

✓ Quality optimization

Driver rating becomes a secondary ranking indicator.

✓ Real-time responsiveness

Ideal for Uber-like dynamic allocation.

✓ Extensible architecture

Tomorrow, you can introduce:

- Surge-based matching
- ETA-based matching
- Vehicle-category matching
- Driver-behavior ML scoring

without breaking the system.

📌 Bottom Line

This class operationalizes your **default driver-matching algorithm** by integrating spatial intelligence and repository-driven ranking logic under a clean Strategy Pattern.

```java
import com.yasif.project.uber.Uber.backend.system.dto.DriverDto;
import com.yasif.project.uber.Uber.backend.system.dto.RideDto;
import com.yasif.project.uber.Uber.backend.system.dto.RideRequestDto;
import com.yasif.project.uber.Uber.backend.system.dto.RiderDto;
import com.yasif.project.uber.Uber.backend.system.entities.Rider;
import com.yasif.project.uber.Uber.backend.system.entities.User;

import java.util.List;

public interface RiderService {  6 usages  1 implementation  Yasif khan *

    RideRequestDto requestRide(RideRequestDto rideRequestDto);  1 usage  1 implementation  Yasif khan

    RideDto cancelRide(Long rideId);  no usages  1 implementation  Yasif khan


    DriverDto rateDriver(Long rideId,Integer rating);  no usages  1 implementation  Yasif khan

    DriverDto getMyProfile();  no usages  1 implementation  Yasif khan

    List<RiderDto> getAllMyRides();  no usages  1 implementation  Yasif khan

    Rider createNewRider(User user);  1 usage  1 implementation  new *

}
```

```java
    @Override  1 usage  new *
    public Rider createNewRider(User user) {
        Rider rider = Rider.builder()
                .user(user)
                .rating(0.0)
                .build();
        return riderRepository.save(rider);
    }
}
```

Create creatNewRider in RiderService and Implement in the RiderServiceImpl

1) Rider createNewRider(User user) (interface + implementation)

What it is / what it does

- Purpose: create and persist a Rider domain object when a new user signs up or when you need to create rider-specific metadata.
- Implementation does:
  - Builds a Rider with user and a default rating of 0.0 using Lombok builder.
  - Calls riderRepository.save(rider) to persist and returns the saved entity.

Why this matters

- Separates user identity (User) from rider-specific behavior/metadata (Rider), which is good domain modeling.
- Ensures you have a dedicated place to store rider-only fields (rating, preferences, wallet info, etc.) from day one.

Potential pitfalls & quick fixes

1. **Null-safety / validation**
   - Ensure user is not null (defensive check). If user isn't managed (not saved yet), saving Rider may cascade or fail depending on mapping.
   - Add Objects.requireNonNull(user, "user must not be null") or throw a domain exception.
2. **Transactionality**
   - If signup() saves User then calls createNewRider(savedUser), ensure the caller is in a transactional context. If signup() is not transactional and createNewRider() fails, you may end up with a User without a Rider.
   - Recommendation: make signup() @Transactional (on service) or annotate this method appropriately to participate in the same tx.
3. **Cascade & orphan handling**
   - Verify JPA mapping between Rider and User — if Rider.user is annotated with @OneToOne or @ManyToOne, check cascade type. Usually you don't want CascadeType.REMOVE from Rider to User.
   - Make sure user has an id (persisted) or mapping allows saving both in one tx.
4. **Return type & DTOs**
   - Returning an entity from a service is fine internally, but for controllers prefer returning RiderDto (to avoid leaking internal JPA proxies).
   - If you return the entity in API, beware lazy-loading surprises in serialization.
5. **Default values & audit**
   - Consider populating createdAt/updatedAt, riderStatus, a unique riderId, and default preferences early.
   - Use @PrePersist or an audit helper.
6. **Uniqueness**

    ◦ If you don't want multiple riders for the same user, add a DB constraint or check riderRepository.findByUser(user) before saving.

```java
@Service  no usages  Yasif khan *
@RequiredArgsConstructor
@Slf4j
public class RiderServiceImpl implements RiderService {

    private final ModelMapper modelMapper;
    private final RideFareCalculationStrategy rideFareCalculationStrategy;
    private final DriverMatchingStrategy driverMatchingStrategy;
    private final RideRequestRepository rideRequestRepository;
    private final RiderRepository riderRepository;


    @Override  1 usage  Yasif khan *
    public RideRequestDto requestRide(RideRequestDto rideRequestDto) {
        RideRequest rideRequest = modelMapper.map(rideRequestDto,RideRequest.class);
        rideRequest.setRideRequestStatus(RideRequestStatus.PENDING);

        // Calculate fare Strategy
        Double fare = rideFareCalculationStrategy.calculateFare(rideRequest);
        rideRequest.setFare(fare);

        // saving into database
        RideRequest savedRideRequest = rideRequestRepository.save(rideRequest);

        // Driver matching
        List<Driver> drivers = driverMatchingStrategy.findMatchingDriver(rideRequest);

        return modelMapper.map(savedRideRequest,RideRequestDto.class);
    }
```

1) Rider createNewRider(User user) (interface + implementation)

What it is / what it does

- Purpose: create and persist a Rider domain object when a new user signs up or when you need to create rider-specific metadata.
- Implementation does:
    - Builds a Rider with user and a default rating of 0.0 using Lombok builder.
    - Calls riderRepository.save(rider) to persist and returns the saved entity.

Why this matters

- Separates user identity (User) from rider-specific behavior/metadata (Rider), which is good domain modeling.

- Ensures you have a dedicated place to store rider-only fields (rating, preferences, wallet info, etc.) from day one.

Potential pitfalls & quick fixes

1. **Null-safety / validation**
    - Ensure user is not null (defensive check). If user isn't managed (not saved yet), saving Rider may cascade or fail depending on mapping.
    - Add Objects.requireNonNull(user, "user must not be null") or throw a domain exception.
2. **Transactionality**
    - If signup() saves User then calls createNewRider(savedUser), ensure the caller is in a transactional context. If signup() is not transactional and createNewRider() fails, you may end up with a User without a Rider.
    - Recommendation: make signup() @Transactional (on service) or annotate this method appropriately to participate in the same tx.
3. **Cascade & orphan handling**
    - Verify JPA mapping between Rider and User — if Rider.user is annotated with @OneToOne or @ManyToOne, check cascade type. Usually you don't want CascadeType.REMOVE from Rider to User.
    - Make sure user has an id (persisted) or mapping allows saving both in one tx.
4. **Return type & DTOs**
    - Returning an entity from a service is fine internally, but for controllers prefer returning RiderDto (to avoid leaking internal JPA proxies).
    - If you return the entity in API, beware lazy-loading surprises in serialization.
5. **Default values & audit**
    - Consider populating createdAt/updatedAt, riderStatus, a unique riderId, and default preferences early.
    - Use @PrePersist or an audit helper.
6. **Uniqueness**
    - If you don't want multiple riders for the same user, add a DB constraint or check riderRepository.findByUser(user) before saving.

2) RideRequestDto requestRide(RideRequestDto rideRequestDto) (RiderServiceImpl)

What it is / what it does (step-by-step)

1. **Mapping**
    - modelMapper.map(rideRequestDto, RideRequest.class) converts incoming DTO → JPA entity.
    - This creates a RideRequest entity populated with pickup/dropoff locations, requested vehicle type, etc.

2. **Status**
   - rideRequest.setRideRequestStatus(RideRequestStatus.PENDING) sets domain state to "pending" — good explicit lifecycle initialization.
3. **Fare Calculation**
   - Double fare = rideFareCalculationStrategy.calculateFare(rideRequest);
   - Uses pluggable strategy to compute fare (distance * multiplier in current default strategy).
   - Sets rideRequest.setFare(fare).
4. **Persistence**
   - RideRequest savedRideRequest = rideRequestRepository.save(rideRequest);
   - Persists the request. This makes the request durable and queryable by other subsystems (matching, analytics).
5. **Driver Matching**
   - List<Driver> drivers = driverMatchingStrategy.findMatchingDriver(rideRequest);
   - Delegates to strategy (nearest-driver now) to fetch candidate drivers.
6. **Return**
   - Returns the persisted RideRequest mapped back to RideRequestDto.

Why the order matters

- Persisting the ride request before matching gives you an audit trail and a persisted ID to reference while matching/assigning.
- If you matched first and then persisted, you'd need more careful orchestration to avoid losing the request if something fails.

Issues, safety concerns & recommended improvements (prioritized)

A. **Validate DTO inputs**

- **Always validate** pickup/dropoff existence and coordinate validity before mapping:
  - Use @Valid on controller + validation annotations on DTO.
  - If rideRequestDto contains geospatial types, validate SRID and nullness.

B. **Currency precision**

- Using double for fare is risky for money:
  - Recommendation: switch to BigDecimal for fare calculation, storage, and API contracts. Use MathContext/scale for rounding policy.

C. **DistanceService & units consistency**

- Ensure DistanceService.calculateDistance returns distance in expected units (km or meters). Strategy multiplier must match the unit.

D. **ModelMapper caveats**

- Mapping geospatial types (JTS Point) with ModelMapper can be tricky. Ensure there is a converter for Point ↔ DTO representation (lon/lat). Otherwise you might lose data or get exceptions.

E. **Driver matching side effects & assignment**

- Currently you fetch drivers but don't assign or mark a driver as reserved. That leaves the system vulnerable to race conditions (double assigning).
  - Immediate fix: after you pick the final driver from drivers, perform an atomic assignment step (DB-level update) using a transaction and a check (e.g., available = true → set available = false and set current_ride_id), or use optimistic locking (@Version) on Driver.
  - Put assignment logic behind a service method: assignDriverToRide(driverId, rideRequestId) that updates both sides in one transaction.

F. **Transactions & consistency**

- Wrap the whole requestRide in a transaction if you intend to perform assignment in the same call. If you just persist the request and return a candidate list, ensure eventual consistency expectations are documented.

G. **Observability & logging**

- Add structured logging:
  - Log request id, rider id, pickup coordinates, calculated fare, number of candidate drivers. Avoid PII.
  - Example: log.info("RideRequest created: id={}, riderId={}, fare={}, candidates={}", savedRideRequest.getId(), savedRideRequest.getRiderId(), fare, drivers.size());

H. **Error handling**

- If fare calculation throws (e.g., distance missing), propagate a domain-specific exception (e.g., InvalidRideRequestException) and map it to 400 in GlobalExceptionHandler.
- You already have RuntimeConflictException and ResourceNotFoundException — consider adding ValidationException or reuse IllegalArgumentException -> 400 mapping.

I. **Return contract**

- You map savedRideRequest back to DTO — ensure the DTO contains the fields you need (id, fare, status). Avoid returning entity directly in controllers.

J. **Async matching (optional)**

- If matching is heavy or requires 3rd-party calls, consider pushing matching to an async pipeline:
  - Persist the ride request synchronously (fast), then enqueue a matching job (Kafka/Redis) to perform matching and assignment.
  - Return immediate 202 Accepted or the persisted request with status PENDING_MATCHING. This avoids high latency in the API call.

Extra micro-improvements you can apply in <15 minutes

Fix money type:

```
// change in RideRequest entity + strategy
private BigDecimal fare;
// strategy returns BigDecimal
```

Add null checks at method entry:

```
Objects.requireNonNull(rideRequestDto, "rideRequestDto is required");
```

Add logging after save:

log.info("RideRequest saved id={} riderId={} fare={}", savedRideRequest.getId(), savedRideRequest.getRider().getId(), fare);

Add @Transactional to requestRide **only if** you plan to assign driver within same call.

Convert Double fare to BigDecimal and round to 2 decimals with RoundingMode.HALF_UP.

Example: safer createNewRider (small refactor)

```
@Transactional
public Rider createNewRider(User user) {
    Objects.requireNonNull(user, "user must not be null");
    // prevent duplicates if necessary
    if (riderRepository.existsByUserId(user.getId())) {
        throw new RuntimeConflictException("Rider already exists for user id: " + user.getId())
    }
    Rider rider = Rider.builder()
            .user(user)
            .rating(0.0)
            .build();
    return riderRepository.save(rider);
}
```

```java
import java.util.Set;

@RequiredArgsConstructor  no usages  Yasif khan *
@Service
public class AuthServiceImpl implements AuthService {

    private final UserRepository userRepository;
    private final ModelMapper modelMapper;
    private final RiderService riderService;

    @Override  no usages  Yasif khan
    public void login(String email, String password) {

    }

    @Override  1 usage  Yasif khan *
    public UserDto signup( @NotNull SignupDto signupDto) {
        User user = userRepository.findByEmail(signupDto.getEmail()).orElse( other: null);
        if(user!=null){
            throw new RuntimeConflictException("User already exist with email "+signupDto.getEmail());
        }

        User mapUser = modelMapper.map(signupDto,User.class);
        mapUser.setRoles(Set.of(Role.RIDER));
        User savedUser = userRepository.save(mapUser);

        // creating user related entities
        riderService.createNewRider(savedUser);

        return modelMapper.map(savedUser,UserDto.class);
    }
```

High-level intent

I implemented the core of the authentication service responsible for **user signup** and left a placeholder for **login**. The signup flow enforces a duplicate-email guard, persists the User, assigns the RIDER role, and provisions the associated Rider domain entity so the system can immediately treat the account as a rider.

File responsibilities (quick summary)

- AuthServiceImpl is a Spring @Service that implements AuthService.
- It coordinates between UserRepository, ModelMapper, and RiderService to perform user onboarding.

Method walkthrough

signup(SignupDto signupDto)

What I do:

1. Check for existing user by email:
   - userRepository.findByEmail(signupDto.getEmail()).orElse(null)
   - If a record exists, I throw RuntimeConflictException (mapped to HTTP 409 in the app).
2. Map incoming DTO to entity:
   - Use ModelMapper to convert SignupDto → User.
3. Assign role:
   - Set the user roles to Set.of(Role.RIDER) so the account has rider privileges out of the box.
4. Persist user:
   - Save the user via userRepository.save(mapUser).
5. Provision rider entity:
   - Call riderService.createNewRider(savedUser) to create the domain Rider record linked to the new User.
6. Return:
   - Map saved User → UserDto and return it as the signup response.

Why this order:

- I persist the User first to obtain a stable identifier and then create the Rider so downstream services and joins are consistent. This yields a clear onboarding lifecycle: create identity → create domain profile.

login(String email, String password)

What I do:

- I left a placeholder method for login to implement authentication (password verification, token issuance) later.

Design strengths

- **Separation of concerns:** The service delegates mapping (ModelMapper) and rider provisioning (RiderService), keeping the auth flow focused on orchestration.
- **Explicit conflict handling:** Duplicate emails surface as RuntimeConflictException (HTTP 409), which is a clear, client-actionable contract.
- **Role assignment on signup:** Ensures downstream authorization logic treats new users correctly without extra steps.

Practical risks I noticed (and one-line remediation proposals)

1. **Password handling (security)**
   - Risk: Password hashing is not shown — storing raw passwords is unsafe.
   - I recommend: hash passwords (e.g., BCryptPasswordEncoder) before saving.
2. **Race condition on duplicate email**
   - Risk: findByEmail + save is vulnerable under concurrent signups.
   - I recommend: add a unique DB constraint on users.email and catch DataIntegrityViolationException to map to a 409.
3. **Transactionality**
   - Risk: If riderService.createNewRider(savedUser) fails after userRepository.save, the system may end up with a User without a Rider.
   - I recommend: mark signup() @Transactional so both operations commit or rollback together.
4. **Validation**
   - Risk: DTO inputs aren't validated here.
   - I recommend: perform controller-level @Valid checks and enforce password/email constraints (min length, format).
5. **Returning entity-derived DTO**
   - Risk: Mapping savedUser directly to UserDto without sanitizing sensitive fields could leak data.
   - I recommend: ensure UserDto excludes sensitive fields (password, secrets).

Lightweight checklist I would follow next

- Add BCryptPasswordEncoder bean and encode password before saving.
- Apply @Transactional to signup() so user + rider persist atomically.
- Add DB unique constraint on email and convert constraint violations into RuntimeConflictException.
- Implement login() to verify password and return a token (or user DTO) — plan for JWT later.
- Add @Valid on controller DTOs and field-level validation annotations on SignupDto.

One-line summary

I built the signup orchestration: conflict detection → user creation → role assignment → rider provisioning, and left login() to implement later; I also identified the key security and consistency hardening steps to apply next.

```java
> import ...

@Repository  2 usages  Yasif khan *
public interface UserRepository extends JpaRepository<User,Long> {
    Optional<User> findByEmail(String email);  1 usage  new *
}
```

What I implemented

I introduced the UserRepository as the data-access layer for the User entity. This repository becomes the single source of truth for all persistence operations related to user accounts.

Class-by-class explanation

✅ 1. public interface UserRepository extends JpaRepository<User, Long>

By extending JpaRepository, I instantly unlocked a full suite of CRUD operations without writing any boilerplate.
This gives me:

- save()
- findById()
- findAll()
- deleteById()
- and more out-of-the-box capabilities.

This is a strategic step — it reduces operational overhead and accelerates development velocity.

✅ 2. Optional<User> findByEmail(String email)

This is the custom query method I added to support the signup flow.

What it does:

- Spring Data JPA auto-generates the SQL based on the method name.
- It searches the users table where email = ?.
- It wraps the result in Optional to safely handle cases where the user may not exist.

Why it's important:

This is the backbone of the **duplicate-email validation** I implemented in AuthServiceImpl.

How this repository integrates into the signup process

In AuthServiceImpl.signup():

Step 1: Check if user exists

User user = userRepository.findByEmail(signupDto.getEmail()).orElse(null);

if(user!=null){

    throw new RuntimeConflictException("User already exist...");

}

This ensures that email uniqueness is enforced at the service level.

Step 2: Save new user

User savedUser = userRepository.save(mapUser);

The repository handles the persistence cleanly, leveraging the underlying JPA/Hibernate lifecycle.

Strategic benefits of this addition

- Centralized data-access abstraction
- Better alignment with domain-driven design
- Reduced boilerplate and enhanced maintainability
- Safe and expressive Optional-based operations
- Clean integration with validation and exception-handling layers

```java
package com.yasif.project.uber.Uber.backend.system.repositories;

import com.yasif.project.uber.Uber.backend.system.entities.Driver;
import org.locationtech.jts.geom.Point;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository  2 usages  Yasif khan *
public interface DriverRepository extends JpaRepository<Driver,Long> {

    @Query(value = "SELECT d.*, ST_Distance(d.current_location, :pickUpLocation) AS distance " +  1 usage  ne
            "FROM driver  d " +
            "WHERE d.available = true AND ST_DWithin(d.current_location, :pickUpLocation, 10000) " +
            "ORDER BY distance " +
            "LIMIT 10", nativeQuery = true)
    List<Driver> findTenNearbyTopRatedDrivers(Point pickUpLocation);
}
```

Here's a crisp, executive-grade breakdown of what **I delivered** when creating the DriverRepository and the custom PostGIS query — articulated with clarity and forward-moving intent.

1. Repository Definition

public interface DriverRepository extends JpaRepository<Driver, Long>

By extending JpaRepository, I unlocked the complete persistence lifecycle for Driver entities without any manual SQL. This becomes the centralized data-access layer for all driver-related operations.

This aligns the architecture with best-practice domain patterns and ensures clean separation of responsibilities.

2. Custom PostGIS Query for Nearest Drivers

This is the core of today's geospatial capability.

@Query(

  value = "SELECT d.*, ST_Distance(d.current_location, :pickUpLocation) AS distance " +

        "FROM driver d " +

```
        "WHERE d.available = true AND ST_DWithin(d.current_location, :pickUpLocation, 10000)
" +
        "ORDER BY distance " +
        "LIMIT 10",
  nativeQuery = true
)
List<Driver> findTenNearbyTopRatedDrivers(Point pickUpLocation);
```

Let’s break it down step-by-step:

3. ST_DWithin — Filtering drivers within 10km radius

ST_DWithin(d.current_location, :pickUpLocation, 10000)

What it does:

- Checks whether the driver’s current location is within **10,000 meters (10 km)** of the pickup point.
- This immediately narrows down to only relevant candidates, improving query efficiency.

Why it matters:

This acts as the **first-layer proximity filter**, ensuring the search space is optimized for performance and accuracy.

4. ST_Distance — Calculating precise distance

ST_Distance(d.current_location, :pickUpLocation) AS distance

What it delivers:

- Computes exact geospatial distance between driver and rider.
- Returns it as an additional calculated column named distance.

Why this is valuable:

This enables **proper sorting** and future enhancements like ETA-based ranking or dynamic surge logic.

5. ORDER BY distance — Ranking drivers by closeness

After narrowing down the list, I sort the drivers in ascending order of distance.

Outcome:

The system now identifies **the closest available drivers** with deterministic precision.

6. LIMIT 10 — Picking the top nearest candidates

This guarantees:

- Performance optimization
- Predictable response size
- Alignment with real-world ride-hailing matching strategies (e.g., fetching top 10 nearest candidates)

7. Integrated with Driver Matching Strategy

This repository method directly powers:

driverRepository.findTenNearbyTopRatedDrivers(rideRequest.getPickUpLocation());

That ensures the strategy layer remains clean, modular, and highly replaceable — a key advantage of the strategy pattern I implemented earlier.

Overall Impact

With this repository and query in place, I now have:

- A **fully geospatial-aware driver matching pipeline**
- PostGIS-optimized nearest-neighbor search
- Clean, scalable integration with service and strategy layers
- A production-aligned foundation for ETA, surge, and ranking algorithms

```java
package com.yasif.project.uber.Uber.backend.system.exceptions;

public class RuntimeConflictException extends RuntimeException{  5 usages  new *
    public RuntimeConflictException() {  no usages  new *
        super();
    }

    public RuntimeConflictException(String message) {  1 usage  new *
        super(message);
    }
}
```

```java
package com.yasif.project.uber.Uber.backend.system.exceptions;

public class ResourceNotFoundException extends RuntimeException{  3 usages  new *
    public ResourceNotFoundException() {  no usages  new *
    }

    public ResourceNotFoundException(String message) {  no usages  new *
        super(message);
    }
}
```

1. ResourceNotFoundException

```
public class ResourceNotFoundException extends RuntimeException {

  public ResourceNotFoundException() {}

  public ResourceNotFoundException(String message) {

    super(message);

  }

}
```

What I did

I introduced a dedicated exception to represent scenarios where the system fails to locate a required domain resource — such as a user, driver, rider profile, ride request, or any entity expected to exist during workflow execution.

Why this is important

- Establishes a **clean, semantic signal** for 404-type failures.
- Decouples business logic from generic Java exceptions.
- Enables the global exception handler (which I'll integrate later) to map this exception to a structured HTTP response.
- Creates a more predictable and resilient error-handling pipeline across the platform.

Strategic value

This provides a **controlled fault boundary**, ensuring downstream layers (controller → service → repository) respond consistently when data isn't available.
This is foundational for delivering a stable, production-grade user experience.

2. RuntimeConflictException

```
public class RuntimeConflictException extends RuntimeException {

    public RuntimeConflictException() { super(); }

    public RuntimeConflictException(String message) { super(message); }

}
```

What I implemented

This custom exception represents **business rule violations**, specifically conflict scenarios — for example, when a new signup request uses an already registered email.

Where it's already used

In the signup workflow:

```
if(user != null) {

    throw new RuntimeConflictException("User already exist with email...");

}
```

Why this matters

- Aligns with **409 Conflict** semantics at the HTTP layer.
- Supports clean separation of business validations from infrastructural concerns.
- Ensures the system immediately surfaces constraint breaches without ambiguity.

Strategic value

This exception type signals operational conflicts clearly, allowing the platform to maintain data integrity and reinforce domain rules without degrading the user journey.

Overall Architectural Impact

By introducing both exceptions, I formalized a structured **error taxonomy** for the Uber backend:

ResourceNotFoundException → Retrieval failures

RuntimeConflictException → Business conflicts or duplication issues

This paves the way for:

- A unified global exception handling layer
- Standardized REST responses
- Cleaner controller logic
- More maintainable service code

```
@RestController  no usages  new *
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthService authService;

    @PostMapping("/signup")  no usages  new *
    UserDto singUp(@RequestBody SignupDto signupDto){
        return authService.signup(signupDto);
    }

}
```

AuthController — What I Delivered Today

I introduced the AuthController as the API-facing entry point for all authentication-related workflows. This is the orchestration layer that bridges client requests with core business services.

1. Class-Level Setup

@RestController

@RequestMapping("/auth")

@RequiredArgsConstructor

public class AuthController {

What this achieves

- **@RestController**: Establishes the class as a REST endpoint provider, returning serialized JSON responses by default.
- **@RequestMapping("/auth")**: Consolidates all auth operations under a clean, predictable URL namespace (/auth/...).
- **@RequiredArgsConstructor**: Enforces immutability and ensures dependency injection without boilerplate.

Together, this creates a lean, production-grade controller pattern aligned with best practices.

2. Dependency Injection

private final AuthService authService;

What I'm doing here

I wired the AuthService into the controller, delegating all logic-heavy workflows to the service layer to maintain controller cleanliness and separation of concerns.

Strategic value

- Prevents business logic leakage into the API layer.
- Supports vertical scalability by keeping controllers as lightweight coordinators.

3. Signup Endpoint

@PostMapping("/signup")

UserDto singUp(@RequestBody SignupDto signupDto) {

    return authService.signup(signupDto);

}

What this endpoint does

- Receives user signup data via SignupDto.
- Hands off the entire workflow to authService.signup().
- Returns a standardized UserDto response back to the client.

Why this design is strong

- Ensures the controller remains declarative and expressive.
- Fully embraces DTO-based communication, strengthening data governance.
- Keeps the API contract clean, predictable, and easily testable.

Operational flow

1. Client sends signup payload
2. Controller receives it and triggers the signup orchestration
3. Service checks email conflicts
4. Service creates User → Rider
5. Response is mapped and returned

This forms a **fully aligned onboarding pipeline** for new riders.

Overall Architectural Impact

The AuthController formalizes the authentication gateway, delivering:

- A streamlined API entry point
- Strong segregation between transport and business layers
- A scalable foundation for future features (login, password reset, OAuth, MFA, etc.)

This completes the first functional slice of the auth module.

```java
import lombok.Builder;
import lombok.Data;
import org.springframework.http.HttpStatus;

import java.util.List;

@Data  7 usages  new *
@Builder
public class ApiError {

    private HttpStatus status;
    private String message;
    private List<String> subError;

}
```

```java
import lombok.Data;

import java.time.LocalDateTime;

@Data  4 usages  new *
public class ApiResponse<T> {

    private LocalDateTime timeStamp;
    private T data;
    private ApiError apiError;

    public ApiResponse(){  2 usages  new *
        this.timeStamp = LocalDateTime.now();
    }

    public ApiResponse(T data){  no usages  new *
        this();
        this.data = data;
    }

    public ApiResponse(ApiError error){  1 usage  new *
        this();
        this.apiError = error;
    }

}
```

```java
@RestControllerAdvice  no usages  new *
public class GlobalExceptionHandler {

    @ExceptionHandler(RuntimeConflictException.class)  no usages  new *
    public ResponseEntity<ApiResponse<?>> handlerRuntimeConflictException( @NotNull RuntimeConflictException ru
        ApiError apiError = ApiError.builder()
                .status(HttpStatus.CONFLICT)
                .message(runtimeConflictException.getMessage())
                .build();
        return builderErrorResponseEntity(apiError);
    }

    @ExceptionHandler(ResourceNotFoundException.class)  no usages  new *
    public ResponseEntity<ApiResponse<?>> handleResourceNotFoundException( @NotNull ResourceNotFoundException e
        ApiError apiError = ApiError.builder()
                .status(HttpStatus.NOT_FOUND)
                .message(exception.getMessage())
                .build();
        return builderErrorResponseEntity(apiError);
    }

    @Contract("_ -> new")
    private @NotNull ResponseEntity<ApiResponse<?>> builderErrorResponseEntity(ApiError apiError){  2 usages  new
        return new ResponseEntity<>(new ApiResponse<>(apiError),apiError.getStatus());
    }

}
```

Here's a structured, enterprise-ready breakdown of the **three advices (ApiError, ApiResponse, and GlobalExceptionHandler)** you implemented today — continuing the same forward-thinking, delivery-focused style.

1. ApiError — My Standardized Error Envelope

```java
@Data

@Builder

public class ApiError {

   private HttpStatus status;

   private String message;

   private List<String> subError;

}
```

What I achieved

I formalized a **uniform error payload** that will be shared across the entire platform. This ensures clients always receive predictable, structured, machine-readable error metadata.

Key Components

- **status** → The HTTP status code of the failure
- **message** → Human-readable explanation of the error
- **subError** → Optional list for granular validation/field-specific errors

Strategic Value

This establishes an enterprise-grade error contract, enabling seamless integration with mobile apps, dashboards, and internal tooling.

2. ApiResponse — My Universal Response Wrapper

```
@Data

public class ApiResponse<T> {

    private LocalDateTime timeStamp;

    private T data;

    private ApiError apiError;


    public ApiResponse() {

        this.timeStamp = LocalDateTime.now();

    }


    public ApiResponse(T data) {

        this();

        this.data = data;

    }


    public ApiResponse(ApiError error) {

        this();
```

```
        this.apiError = error;
    }
}
```

What I delivered

I introduced a standardized response envelope that wraps **both success and failure outcomes** in a consistent contract.

Capabilities

- Auto-timestamps every response
- Holds data when operations succeed
- Holds apiError when exceptions occur

Why this matters

This aligns all API responses to a single format, dramatically improving client-side parsing, debugging, and observability across the board.

3. GlobalExceptionHandler — My Centralized Exception Governance Layer

```
@RestControllerAdvice

public class GlobalExceptionHandler {
```

What I implemented

A top-level governance layer that intercepts all exceptions across the application and transforms them into **standardized ApiResponse objects**.

Handled Exceptions

a) RuntimeConflictException → HTTP 409

```
@ExceptionHandler(RuntimeConflictException.class)

public ResponseEntity<ApiResponse<?>> handlerRuntimeConflictException(...) {
```

Used for business rule conflicts like duplicate email signup.

b) ResourceNotFoundException → HTTP 404

```
@ExceptionHandler(ResourceNotFoundException.class)
```

public ResponseEntity<ApiResponse<?>> handleResourceNotFoundException(...) {

Used when requested resources don't exist (e.g., missing driver, rider, or ride request).

Central Builder Method

private ResponseEntity<ApiResponse<?>> builderErrorResponseEntity(ApiError apiError)

This ensures:

- Consistency in response format
- Accurate status codes
- Clean separation of concerns

Enterprise Value

This layer creates a **fully unified error-handling pipeline**, eliminating scattered try-catch blocks and improving API hygiene.

Overall Architecture Impact of Today's Advices

By introducing these three components, I operationalized:

✅ A unified API response model
✅ A consistent error-handling strategy
✅ A production-grade exception governance layer
✅ Better client integration and debugging capability
✅ Scalable foundation for future frameworks (logging, tracing, observability)

This effectively closes the loop on cross-cutting concerns for the service.

Summary

Perfect — I'll walk through each file/component you shared **one by one**, explain what it does, call out bugs or pitfalls, and give short, actionable suggestions you can apply immediately. I'll keep it pragmatic and deployment-minded so you can iterate safely.

1) RiderServiceImpl

What it does

- Implements RiderService.

- requestRide(RideRequestDto):
  - Maps DTO → RideRequest entity using ModelMapper.
  - Sets RideRequestStatus.PENDING.
  - Computes fare via RideFareCalculationStrategy.
  - Persists the RideRequest using rideRequestRepository.save.
  - Calls DriverMatchingStrategy to get matching drivers (returns a List<Driver>).
  - Returns the saved RideRequest mapped back to RideRequestDto.
- createNewRider(User):
  - Builds a Rider entity with default rating = 0.0 and the provided User.
  - Persists via riderRepository.save.

Issues & suggestions

- **Transactionality:** signup() in AuthServiceImpl calls riderService.createNewRider(savedUser). Make sure signup() is @Transactional so both user and rider persist/rollback together on error.
- **Null checks / validation:** validate rideRequestDto fields (pickup/dropoff coords) before mapping.
- **Driver assignment:** currently you only *fetch* matching drivers — you still need a safe assignment step (transaction + optimistic lock or DB-level marker) to avoid double-booking.
- **Return value:** you map and return savedRideRequest — make sure stored associations (e.g., foreign keys) are correctly initialized for DTO mapping.

2) RideFareCalculationStrategy (interface)

What it does

- Strategy abstraction for fare calculation.
- Exposes double calculateFare(RideRequest rideRequest).
- Exposes RIDE_FARE_MULTIPLIER = 10 as a default constant (you can use or override in implementations).

Why it's good

- Decouples pricing concerns from business flow — swap in surge, ML model, or promotions without changing service code.

Suggestion

- Consider returning BigDecimal for currency calculations to avoid floating point precision issues in money computations.

3) RiderFareDefaultFareCalculationStrategy (implementation)

What it does

- Uses DistanceService to compute distance and multiplies by RIDE_FARE_MULTIPLIER.

Bug

```
distanceService.calculateDistance(rideRequest.getPickUpLocation(),
                                   rideRequest.getPickUpLocation());
```

It passes pickup as both origin and destination, producing zero distance.

Fix

- Change the second parameter to drop-off location:

```
distanceService.calculateDistance(rideRequest.getPickUpLocation(),
                                  rideRequest.getDropOffLocation());
```

Other suggestions

- Validate existence of dropOffLocation.
- Use BigDecimal for final fare; round to cents using a consistent rounding mode.
- Add metrics for distance and fare distribution.

4) DriverMatchingNearestDriverStrategy

What it does

- Primary implementation of DriverMatchingStrategy.
- Delegates to DriverRepository.findTenNearbyTopRatedDrivers(pickupLocation) to fetch up to 10 nearby drivers (nearest first).

Notes

- Good single-responsibility implementation — keeps matching logic pluggable.
- You may later implement other strategies (rating-aware, ETA-aware, surge-prioritized).

Suggestion

- After fetching candidates, you may want to filter/sort in Java by last seen, vehicle type, rating, etc., before selecting the final driver(s).
- Consider an interface method that returns an ordered list of candidates with computed ETA/distance, not just entity list.

5) RiderFareDefaultFareCalculationStrategy & DistanceService interaction

What to check

- Ensure DistanceService#calculateDistance expects the same types as RideRequest's location fields (e.g., Point from JTS).
- If using lat/lon, ensure coordinate order (lon, lat vs lat, lon) is consistent.

6) AuthServiceImpl (signup + login)

What it does

- signup(SignupDto):
  - Checks if user exists by email via userRepository.findByEmail.
  - If exists → throws RuntimeConflictException.
  - Maps SignupDto → User and sets role Role.RIDER.
  - Persists User.
  - Calls riderService.createNewRider(savedUser).
  - Returns mapped UserDto.

Issues & suggestions

- **Transactionality:** make signup() transactional to ensure atomicity between user and rider record creation.
- **Password handling:** not shown — ensure password is hashed & salted before saving.
- **Duplicate-check race condition:** findByEmail -> save pattern has a race; consider unique DB constraint on email column and handle constraint violation gracefully (map to 409).
- **Logging & monitoring:** log signup events (avoid logging raw passwords).

7) UserRepository

What it does

- Simple JPA repository with findByEmail.

Suggestion

- Ensure email has a unique constraint at the DB schema level to prevent duplicates under concurrent writes.

8) DriverRepository (PostGIS native query)

What it does

```sql
SELECT d.*, ST_Distance(d.current_location, :pickUpLocation) AS distance
FROM driver d
WHERE d.available = true
  AND ST_DWithin(d.current_location, :pickUpLocation, 10000)
ORDER BY distance
LIMIT 10
```

Potential problems

1. **Parameter binding for Point:** Passing a org.locationtech.jts.geom.Point directly works only if your JPA provider supports binding JTS geometries — otherwise use WKT/WKB or ST_SetSRID(ST_MakePoint(:lon,:lat),4326).
2. **Coordinate units:** ST_DWithin units depend on the SRID: if current_location is geometry(Point,4326), distances are in degrees → inaccurate. Use geography type or cast to geography: current_location::geography.
3. **Indexing:** Ensure GIST index on current_location::geography or current_location for performance.
4. **Return mapping:** You add extra column distance. Native query returning d.* may still map to Driver but the extra column is ignored; if you want the distance value in Java, map to a projection (e.g., List<Object[]> or a DTO projection).
5. **SRID consistency:** Ensure driver points and incoming pickUpLocation share same SRID (4326 usually).

Recommendations

- Use geography for meters:

```sql
SELECT d.*, ST_Distance(d.current_location::geography, ST_SetSRID(ST_MakePoint(:lon, :lat)
FROM driver d
WHERE d.available = true
  AND ST_DWithin(d.current_location::geography, ST_SetSRID(ST_MakePoint(:lon, :lat), 4326)
ORDER BY distance_m
LIMIT 10;
```

```
tSRID(ST_MakePoint(:lon, :lat), 4326)::geography) AS distance_m

T_MakePoint(:lon, :lat), 4326)::geography, 10000)
```

Or accept lon/lat params in repository and construct the point inside SQL.

If you need distance in Java, map via projection:

```
@Query(value = " native sql ", nativeQuery = true)
List<Object[]> findTenNearbyTopRatedDrivers(double lon, double lat);
// each Object[] => {Driver fields..., distance}
```

9) RuntimeConflictException & ResourceNotFoundException

What they do

- Simple RuntimeException subclasses used to signal domain error conditions.

Notes

- Good practice to keep domain exceptions distinct.
- You already map RuntimeConflictException → HTTP 409 in GlobalExceptionHandler.

Suggestion

- Optionally annotate @ResponseStatus(HttpStatus.CONFLICT) on RuntimeConflictException for a declarative fallback (redundant with the advice but useful for controller-level exceptions).
- Consider adding error codes or structured fields for clients to programmatically react.

10) AuthController

What it does

- Exposes /auth/signup as POST that accepts SignupDto and returns UserDto via AuthService#signup.

Suggestion

- Add @Valid on @RequestBody DTO and DTO validation annotations for input validation.
- Return appropriate HTTP statuses: 201 Created for new resource (optional: ResponseEntity.created(...)).

11) ApiError and ApiResponse

What they do

- ApiError is a structured error payload with HttpStatus, message, subError.
- ApiResponse<T> is the wrapper returned by controllers including timestamp, data, and optional ApiError.

Why this is good

- Standardizes error payloads across the API — good for frontend/mobile UX and easier debugging.

Suggestion

- Consider including an error code (string or int) so clients can switch on machine-friendly error types.
- Avoid serializing HttpStatus directly; prefer int status and String error for client-friendliness.

12) GlobalExceptionHandler

What it does

- Catches RuntimeConflictException → builds ApiError with HttpStatus.CONFLICT and returns ApiResponse<ApiError> with that status.
- Catches ResourceNotFoundException → returns 404 similarly.
- Uses builderErrorResponseEntity to wrap the response.

Suggestions

- Add handlers for:
  - MethodArgumentNotValidException → return 400 with field errors in subError.
  - DataIntegrityViolationException (DB unique constraint) → map to 409.
  - Exception fallback → map to 500 with safe message and log the stack trace.
- Add structured logging for exceptions (capture correlationId if you have one).

Quick prioritized action list (apply in < 30–60 mins)

1. **Fix the distance bug** in RiderFareDefaultFareCalculationStrategy (use drop-off location).
2. **Add @Transactional** to AuthServiceImpl.signup() to ensure atomic persistence of User + Rider.
3. **Adjust DriverRepository query** to use geography or SRID-correct point creation with lon/lat; decide how you'll pass coordinates (Point vs lon/lat).
4. **Add DB index & unique constraint**:
   - GIST index on driver location as geography.
   - Unique index on users.email.
5. **Add input validation** to controllers (@Valid) and handle validation failures in GlobalExceptionHandler.