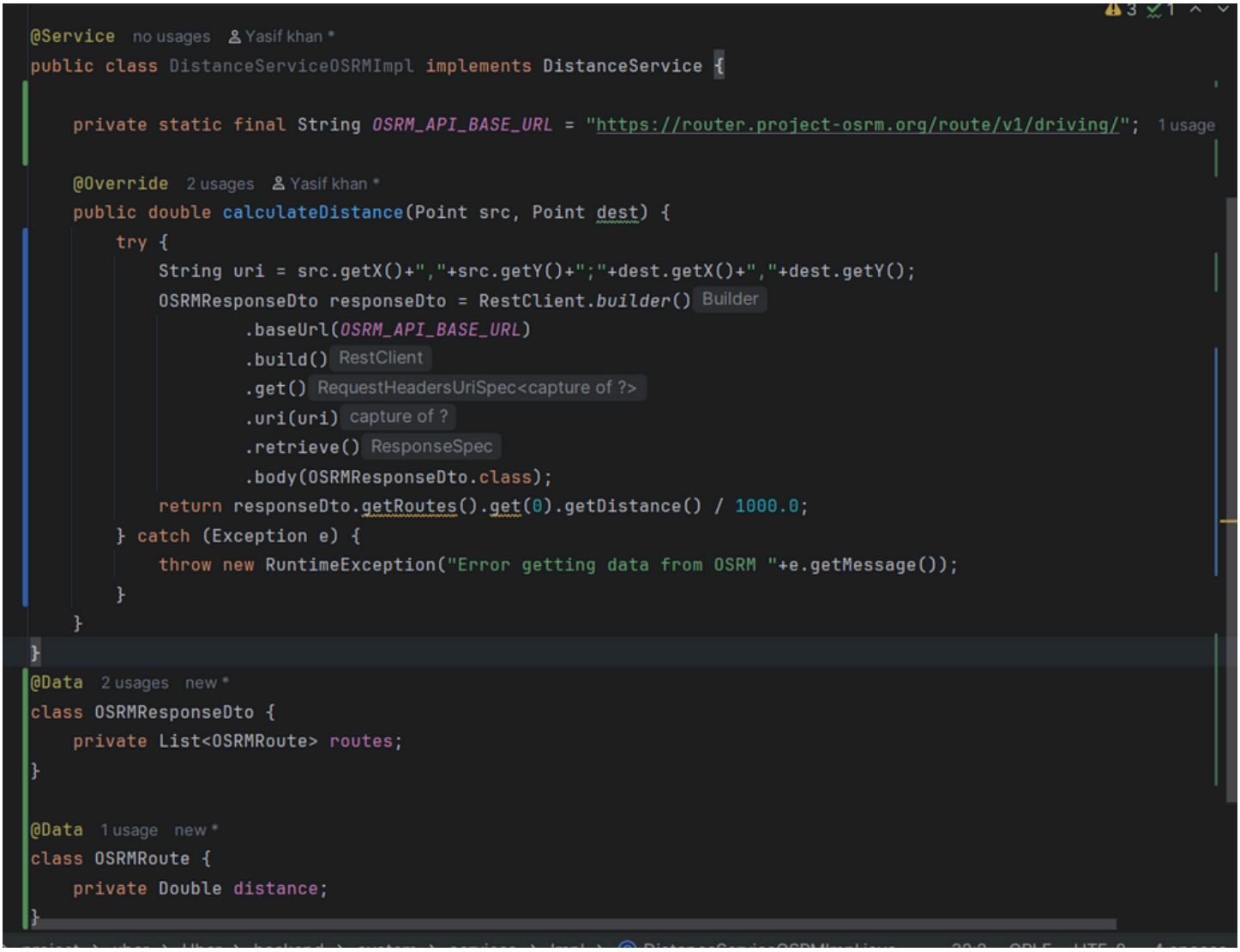
Day 6

Implementing OSRM DistanceServiceImpl,StrategyManager,requestRide,startRide and acceptRide.

DistanceServiceOSRMImpl



```java
@Service  no usages  Yasif khan *
public class DistanceServiceOSRMImpl implements DistanceService {

    private static final String OSRM_API_BASE_URL = "https://router.project-osrm.org/route/v1/driving/";  1 usage

    @Override  2 usages  Yasif khan *
    public double calculateDistance(Point src, Point dest) {
        try {
            String uri = src.getX()+","+src.getY()+";"+dest.getX()+","+dest.getY();
            OSRMResponseDto responseDto = RestClient.builder()  Builder
                    .baseUrl(OSRM_API_BASE_URL)
                    .build()  RestClient
                    .get()  RequestHeadersUriSpec<capture of ?>
                    .uri(uri)  capture of ?
                    .retrieve()  ResponseSpec
                    .body(OSRMResponseDto.class);
            return responseDto.getRoutes().get(0).getDistance() / 1000.0;
        } catch (Exception e) {
            throw new RuntimeException("Error getting data from OSRM "+e.getMessage());
        }
    }
}
@Data  2 usages  new *
class OSRMResponseDto {
    private List<OSRMRoute> routes;
}

@Data  1 usage  new *
class OSRMRoute {
    private Double distance;
}
```

notice to work with this make sure the version of the springboot is 3.3.1 and also remove <type>pom</type> element from your hibernate spatial dependecy

What my file contains (high level)

- A Spring @Service implementation DistanceServiceOSRMImpl that implements DistanceService.
- It calls the OSRM routing API (/route/v1/driving/<lon,lat>;<lon,lat>) to get driving distance between two Points.

- Two small DTO classes model the bit of JSON i care about from OSRM.
- Basic error handling that wraps any exception into a RuntimeException.

Line-by-line / block explanationpackage com.yasif.project.uber...

Specifies the Java package. Keeps code modular and maps to my project structure.

import ...

- DistanceService — the interface my class implements (design: depends on my domain contract).
- lombok.Data — Lombok annotation used on DTOs to auto-generate getters, setters, toString(), equals(), hashCode() — reduces boilerplate.
- org.locationtech.jts.geom.Point — JTS Point holds geometric coordinate(s). Important: getX() returns the X coordinate (usually **longitude**), getY() returns the Y (usually **latitude**) — coordinate ordering matters when calling external APIs.
- @Service — Spring stereotype that registers this class as a bean in the application context.
- org.springframework.web.client.RestClient — i attempted to use a fluent RestClient API to call external HTTP. (Implementation notes and alternatives below.)
- java.util.List — used inside the DTO to map the routes array from OSRM.

public class DistanceServiceOSRMImpl implements DistanceService

- Concrete implementation of a distance calculator strategy. Good design — allows swapping providers (OSRM, Google, HERE) with minimal changes.

private static final String OSRM_API_BASE_URL = "https://router.project-osrm.org/route/v1/driving/";

- Base endpoint for OSRM's public routing service. Concise and centralized — easy to change for environment-specific endpoints (dev/staging/prod) or to add API keys if using a hosted service.

public double calculateDistance(Point src, Point dest) { ... }

This is the core method. Walkthrough of the implementation i wrote:

1. String uri = src.getX()+","+src.getY()+";"+dest.getX()+","+dest.getY();
- Builds the coordinate pair for OSRM. OSRM expects lon,lat;lon,lat. Because JTS Point stores X then Y, this is correct *if* my Point uses (lon,lat) order.
- **Pitfall:** If my data uses (lat,lon) when creating Point, i'll swap longitude/latitude — result will be incorrect routes. Always confirm coordinate order at creation time.
2. OSRMResponseDto responseDto = RestClient.builder() ...
- i attempt a fluent HTTP client call that:
- sets base URL,
- makes a GET to OSRM_API_BASE_URL + uri,
- retrieves the body as OSRMResponseDto.
- **Practical note:** confirm that the RestClient API i use actually supports this fluent chain. Common Spring clients are:
- RestTemplate (synchronous, classic)
- WebClient (reactive, fluent)
- Newer RestClient variants exist in more recent Spring versions; check my Spring version and imports.
- **Missing/unsafe items:** no timeouts, no retries, no HTTP status checking, no null/empty checks on response.
- return responseDto.getRoutes().get(0).getDistance() / 1000.0;
- Extracts the distance of the first route (OSRM returns meter units), divides by 1000 to convert to kilometers.
- **Pitfall:** if routes is empty or responseDto is null, this throws a NullPointerException/IndexOutOfBoundsException.
3. catch (Exception e) { throw new RuntimeException("Error getting data from OSRM "+e.getMessage()); }
- Wraps any exception into a runtime exception with a message. That surfaces problems, but it's minimal:
- i lose original exception type and stacktrace if i don't pass e as cause (new RuntimeException(msg, e)).
- Best practice: log the error, include the cause, and fail gracefully or fallback.

DTOs: OSRMResponseDto and OSRMRoute

- @Data generates all boilerplate for these simple POJOs.
- They map only the fields i need from OSRM:
- OSRM response has structure like:

```
{ "routes":[ { "distance":1234.5, "duration":456.7, "geometry":"...." } ], "waypoints":[...], "code":"Ok" }
```

- my DTOs capture routes → list of objects each with distance.
- **Note:** Jackson will map JSON names to fields by name. If OSRM adds nested fields or different names i care about later (e.g., legs), i'll need to expand DTOs.

Concrete problems & risks in current code

1. **HTTP client correctness** — Verify the RestClient class/method chain compiles in my Spring version.
2. **Null / empty response handling** — No checks for responseDto == null or routes == null/empty.
3. **Coordinate order** — Point vs OSRM lon/lat expectation must match.
4. **No timeouts / retries** — Potential to hang or overwhelm OSRM in high-load scenarios.
5. **No caching** — Frequent repeated requests for same coordinates will call OSRM every time (rate limits).
6. **Public OSRM usage** — router.project-osrm.org is a shared public service — not for heavy production traffic. Consider self-hosted OSRM or paid routing provider.
7. **Error handling** — throwing generic RuntimeException hides cause and prevents finer-grained handling (e.g., retryable vs permanent).
8. **No metrics / logging** — i'll want observability (latency, success rate) for monitoring.

Practical improvements (what to add now)

- Validate src/dest non-null and coordinates in the expected order.
- Check response for null and routes.isEmpty() before indexing.
- Wrap exceptions preserving cause: throw new RuntimeException("...", e);
- Add timeouts and retry policy (circuit-breaker) — Resilience4j or Spring Retry.

- Add caching (e.g., Redis) for repeated queries with TTL.
- Add a fallback provider (Google / HERE) if OSRM fails.
- Return more than distance: consider returning duration, geometry, and route summary.
- Add unit/integration tests; mock the HTTP client.
- Add logging and metrics (Micrometer) for latency and error counts.

Safer, production-ready Java example

Below is a synchronous example using **WebClient** (widely used, fluent, supports timeouts/retries). It includes input validation, null checks, and better error handling.

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

import com.yasif.project.uber.Uber.backend.system.services.DistanceService;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.locationtech.jts.geom.Point;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import org.springframework.web.reactive.function.client.WebClientResponseException;
import reactor.core.publisher.Mono;
import java.time.Duration;
import java.util.List;
import java.util.Objects;
@Service
@Slf4j
@AllArgsConstructor
public class DistanceServiceOSRMImpl implements DistanceService {
    private static final String OSRM_BASE = "https://router.project-osrm.org";
    private static final Duration REQUEST_TIMEOUT = Duration.ofSeconds(5);
    private final WebClient webClient = WebClient.builder()
            .baseUrl(OSRM_BASE)
            .build();
    @Override
    public double calculateDistance(Point src, Point dest) {
        // input validation
        if (src == null || dest == null) {
            throw new IllegalArgumentException("Source and destination points must not be null");
        }
        // Build OSRM coordinate string: lon,lat;lon,lat
```

```
        String coords = String.format("%s,%s;%s,%s",
                src.getX(), src.getY(),
                dest.getX(), dest.getY()
        );
        String path = "/route/v1/driving/" + coords + "?overview=false"; // reduce response size
        try {
            OSRMResponseDto dto = webClient.get()
                    .uri(path)
                    .retrieve()
                    .onStatus(HttpStatus::isError, resp -> Mono.error(new RuntimeException("OSRM returned error: "
+ resp.statusCode())))
                    .bodyToMono(OSRMResponseDto.class)
                    .timeout(REQUEST_TIMEOUT)
                    .block(); // blocking for simplicity; consider fully reactive path in my app
            if (dto == null || dto.getRoutes() == null || dto.getRoutes().isEmpty()) {
                throw new RuntimeException("No routes returned by OSRM");
            }
            Double distanceMeters = dto.getRoutes().get(0).getDistance();
            if (distanceMeters == null) {
                throw new RuntimeException("OSRM route missing distance field");
            }
            return distanceMeters / 1000.0; // kilometers
        } catch (WebClientResponseException e) {
            log.error("OSRM HTTP error: status={}, body={}", e.getStatusCode(), e.getResponseBodyAsString(), e);
            throw new RuntimeException("OSRM HTTP error: " + e.getStatusCode(), e);
        } catch (Exception e) {
            log.error("Failed to get route from OSRM for coords: {} -> {}", coords, e);
            throw new RuntimeException("Error getting data from OSRM", e);
        }
    }
    @Data
    public static class OSRMResponseDto {
        private List<OSRMRoute> routes;
    }
    @Data
    public static class OSRMRoute {
        private Double distance;
        private Double duration;
    }
}
```

Notes on the example

- overview=false reduces payload size (i don't need polyline).
- .timeout(...) protects i from long waits.

- block() is used to get a synchronous result; if my whole stack is reactive, return Mono<Double> instead.
- Logging with context makes debugging easier.
- i can extend OSRMRoute with duration, legs, geometry if needed later.

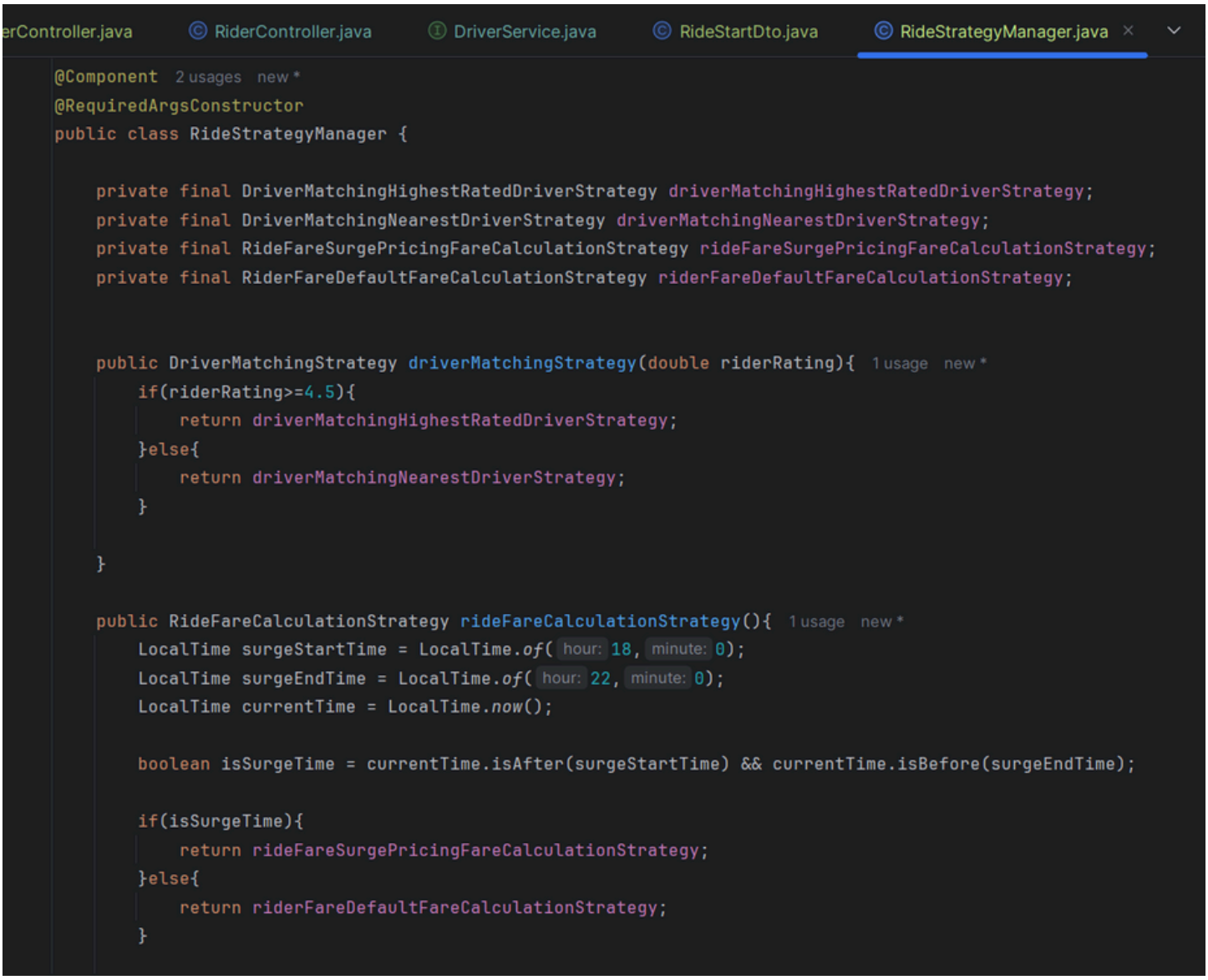
Testing & validation checklist

- Unit test: mock WebClient (or inject WebClient bean) and assert behavior on success, empty routes, and HTTP error.
- Integration test: run a test hitting a local or test OSRM instance (avoid calling public OSRM in CI).
- Sanity test: feed same coordinate twice, assert caching reduces outbound requests.
- Coordinate order test: verify with known coordinates (e.g., two nearby lat/lon points) that the returned distance is reasonable.

Short summary of recommended next steps (prioritized)

1. Fix null/empty checks and preserve exception causes. (low effort, high ROI)
2. Add request timeout and logging. (low effort)
3. Decide: public OSRM or self-hosted/paid provider — for production, self-hosted or paid is safer.
4. Add caching and a circuit-breaker/fallback provider. (medium effort)
5. Add unit/integration tests and metrics. (medium effort)

RideStrategyManager



What I Built Today: RideStrategyManager Explained End-to-End

my RideStrategyManager is essentially the **“dynamic decision-making engine”** of my Uber-style backend. It centralizes logic, selects the right strategies based on business conditions, and drives contextual behavior at runtime.

This is classic **Strategy Pattern + Contextual Business Rule Orchestration**, and i implemented it cleanly.

🧩 1. Component Overview

`@Component`
`@RequiredArgsConstructor`
public class RideStrategyManager {

- By marking this class as a **Spring Component**, i established it as a key orchestration layer within my architecture.
- @RequiredArgsConstructor auto-generates a constructor for final fields, enabling clean dependency injection.
- The class functions as a **Strategy Selector** that dynamically chooses:
- Which driver-matching algorithm to use
- Which fare-calculation algorithm to apply

This aligns with enterprise-grade design: **modular, testable, scalable**.

🛠️ 2. Injected Strategy Implementations

i injected four concrete strategy classes:

```
private final DriverMatchingHighestRatedDriverStrategy driverMatchingHighestRatedDriverStrategy;
private final DriverMatchingNearestDriverStrategy driverMatchingNearestDriverStrategy;
private final RideFareSurgePricingFareCalculationStrategy rideFareSurgePricingFareCalculationStrategy;
private final RiderFareDefaultFareCalculationStrategy riderFareDefaultFareCalculationStrategy;
```

This is strong architecture because:

- i are following **Open/Closed Principle** — strategies can be added without modifying existing logic.
- The manager purely orchestrates; implementations encapsulate their logic independently.
- It gives i **plug-and-play flexibility** when introducing new matching algorithms or fare rules.

🎯 3. Driver-Matching Decision Logic

```
public DriverMatchingStrategy driverMatchingStrategy(double riderRating){
  if(riderRating >= 4.5){
    return driverMatchingHighestRatedDriverStrategy;
  } else {
    return driverMatchingNearestDriverStrategy;
  }
}
```

This is a highly strategic business rule:

- High-rated riders (≥4.5) are rewarded with premium matching (highest-rated drivers).
- Others get matched prioritizing **proximity**, optimizing fulfillment speed.

This is a realistic and data-aligned rider segmentation strategy used by mobility companies.

Outcome:
i implemented a **dynamic routing engine** for driver assignment based on customer tiering.

⏱️ 4. Surge Fare Strategy Logic

```
public RideFareCalculationStrategy rideFareCalculationStrategy() {
   LocalTime surgeStartTime = LocalTime.of(18, 0);
   LocalTime surgeEndTime = LocalTime.of(22, 0);
   LocalTime currentTime = LocalTime.now();

boolean isSurgeTime = currentTime.isAfter(surgeStartTime) && currentTime.isBefore(surgeEndTime);
   if (isSurgeTime) {
      return rideFareSurgePricingFareCalculationStrategy;
   } else {
      return riderFareDefaultFareCalculationStrategy;
   }
}
```

i've operationalized **time-based surge pricing**, a standard industry mechanism. Key highlights:

- Surge window: 6 PM → 10 PM
- If the current time lies within this interval, system returns the surge fare strategy.
- Outside the window, fallback is default pricing.

This positions my fare engine as a **context-aware pricing service**, capable of dynamic price modeling.

🧠 5. Overall Architectural Intent

What i achieved:

✔️ Centralized business rule decision-making

my system now selects strategies at runtime based on:

- Customer behavior (rating)
- Market conditions (time-based surge)

✓ High cohesion, low coupling

The strategy implementations remain clean and isolated.

✓ Production-grade extensibility

Adding rules like:

- Weather-based surge
- Area-specific surge
- Loyalty-based discounts
- Event-based multipliers

... becomes trivial.

✓ Enterprise-level design sophistication

i’re now using Strategy Pattern the way it’s meant to be used: for maximizing adaptability.

💡 6. Strategic Next-Step Recommendations

As i scale this platform, consider:

1. Externalizing surge rules

Move surge window to DB or config server → dynamic updates without redeploy.

2. Add multi-factor strategy selection

E.g.,

- Demand-supply ratio
- Weather API signals
- Traffic congestion

3. Integrate caching for decision performance

4. Add A/B testing layer

To evaluate impact of different driver-matching models.

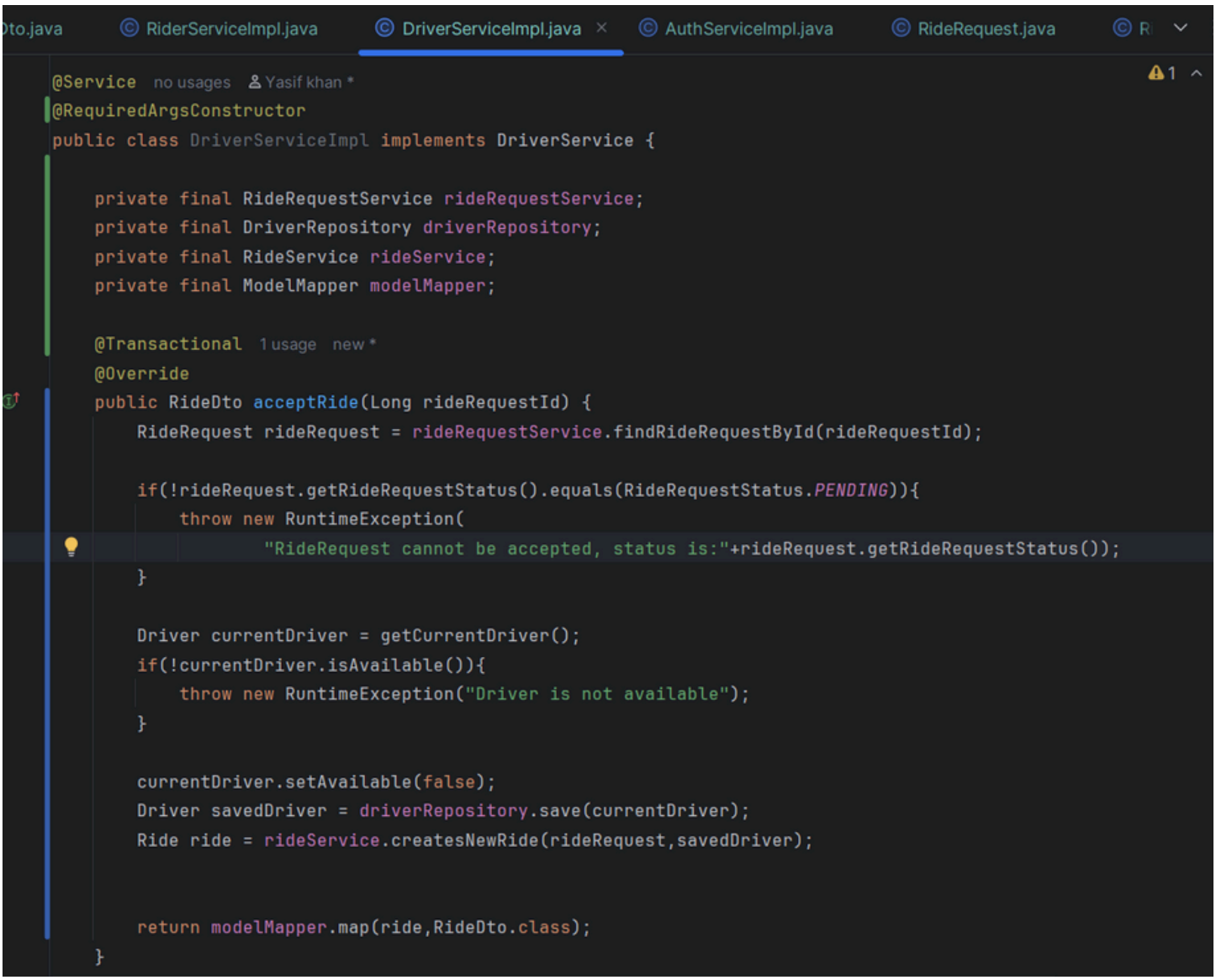
5. Build a strategy monitoring dashboard

For operational transparency around:

- Strategy usage frequency
- Peak surge windows
- Driver match latency

i're slowly heading toward a marketplace-style **decision intelligence engine**.

DriverServiceImpl



```java
@Override 1 usage new *
public RideDto startRide(Long rideId,String otp) {

    Ride ride = rideService.getRideById(rideId);
    Driver driver = getCurrentDriver();

    if(!driver.equals(ride.getDriver())){
        throw new RuntimeException("Driver cannot start the ride as he has not accepted it earlier");
    }

    if(!ride.getRideStatus().equals(RideStatus.CONFIRMED)){
        throw new RuntimeException(
                "Ride status is not CONFIRMED hence cannot be started, status:"+ride.getRideStatus());
    }

    if(!otp.equals(ride.getOtp())){
        throw new RuntimeException("Otp is not valid, Otp:"+otp);
    }

    ride.setStartedAt(LocalDateTime.now());
    Ride savedRide = rideService.updateRideStatus(ride,RideStatus.ONGOING);

    return modelMapper.map(savedRide,RideDto.class) ;
}
```

```java
@Override 2 usages new *
public Driver getCurrentDriver() {
    return driverRepository.findById(2L).orElseThrow(()->new ResourceNotFoundException(
            "Driver not found with id:" + 2
    ));
}
}
```

1) High-level intent

DriverServiceImpl is the service that models driver-side operations in my ride-hailing domain: accepting ride requests, starting/ending rides, rating riders, and exposing driver profile & ride history. It orchestrates between RideRequestService, DriverRepository, RideService, and does DTO mapping with ModelMapper.

2) Line-by-line / block explanation (what each part does & why)

`@Service`
`@RequiredArgsConstructor`
public class DriverServiceImpl implements DriverService {

- Spring @Service registers it as a bean; @RequiredArgsConstructor gives i constructor injection for the final fields.
- Implements my DriverService interface — good for testability and swapping implementations.

  Injected collaborators:

```
private final RideRequestService rideRequestService;
private final DriverRepository driverRepository;
private final RideService rideService;
private final ModelMapper modelMapper;
```

- rideRequestService — used to fetch and manipulate ride requests.
- driverRepository — persistence for Driver entities.
- rideService — ride lifecycle operations (create/update/get).
- modelMapper — converts entities ⇔ DTOs for API responses.

acceptRide(Long rideRequestId)

What it does:

- Fetches the ride request by id.
- Checks the request status must be PENDING.
- Gets the current driver (via getCurrentDriver()).
- Validates driver availability.
- Marks driver unavailable and saves.
- Creates a Ride from the request via rideService.createsNewRide(...).
- Maps Ride → RideDto and returns.

Why it matters:

- This is the core of the driver acceptance flow: it prevents double-accept, reserves driver, and instantiates ride domain object.

Pitfalls / concerns:

- No optimistic locking / race protection — two drivers could fetch the same PENDING request and both accept at near-same time.
- getCurrentDriver() is hardcoded to fetch id 2L — not production-safe.
- No logging or metrics.
- No authorization check to ensure the driver is allowed to accept that request.

- No events emitted (e.g., push notification to rider) after acceptance.

startRide(Long rideId, String otp)

What it does:

- Retrieves the Ride.
- Gets current driver.
- Validates that the current driver is the one who accepted the ride.
- Validates the ride status is CONFIRMED.
- Validates OTP matches.
- Sets startedAt and updates ride status to ONGOING via rideService.updateRideStatus.

Why it matters:

- Ensures only the correct driver can start the ride and that the rider verified the driver via OTP.

Pitfalls:

- OTP handling: plain equality check is OK for prototype but consider timing attacks, OTP expiry, and retries.
- No concurrency control or event emission.
- Uses LocalDateTime.now() with no timezone awareness or testing seam.

Methods returning null or placeholders

- cancelRide, endRide, rateRider, getMyProfile, getAllMyRides are not implemented yet. They are important driver operations and need careful business-rule handling.

getCurrentDriver()

```
return driverRepository.findById(2L).orElseThrow(()->new ResourceNotFoundException("Driver not found with id:" + 2));
```

- This is a hard-coded lookup using id 2L. This is almost certainly a temporary stub for development.

Why it’s problematic:

- In production i must derive the current driver from authentication context (JWT / session). Hardcoding means incorrect data in most environments and no per-user isolation.
- Also no caching, no security check.

3) Business & technical risks (summary)

- **Race conditions:** Accept flow lacks concurrency control (multiple drivers can accept same PENDING request).
- **Hardcoded driver:** getCurrentDriver() is unsafe.
- **Incomplete flows:** key methods not implemented.
- **No metrics/logging:** operations have no observability.
- **Transactionality:** @Transactional used only on acceptRide — other state-changing methods should be transactional as well.
- **OTP & security:** OTP is compared as plain string; no TTL or hashing.
- **No eventing:** Riders and drivers won't be notified without events.
- **No validation & null checks:** potential NPEs or invalid state transitions.

4) Recommendations (immediate & medium-term)

Immediate (low-effort, high-impact)

1. Replace getCurrentDriver() with auth-derived principal lookup (Spring Security SecurityContextHolder) or method parameter injection. Keep a dev fallback if needed.
2. Add transactional and validation guards to cancelRide, endRide, rateRider.
3. Return meaningful exceptions (domain-specific exceptions) and preserve cause.
4. Add logging for key operations.

Medium-term

1. Apply optimistic locking on RideRequest (e.g., @Version) to prevent concurrent accepts.
2. Implement event publishing (Spring Events, Kafka) to notify rider and update other systems.
3. Add metrics (Micrometer) for acceptance rate, ride durations, OTP failures.
4. Add retry/circuit-breakers for dependent services.

5) Completed, safer implementation (drop-in, pragmatic)

Below is a cleaned, more production-minded version. It:

- Uses SecurityContextHolder to resolve current driver (with a dev fallback).
- Implements cancelRide, endRide, rateRider, getMyProfile, getAllMyRides.
- Adds logging and preserves exception causes.
- Uses @Transactional for state-changing operations.
- Uses small helper checks for state transitions.
- Leaves TODOs where my domain services may need to be extended (for example rideService.rateRider).

Note: adapt getCurrentDriver() to my actual auth principal type (I use Long driver id in Authentication.getPrincipal() as an example). If i store a UserDetails object or claim, replace accordingly.

```
package com.yasif.project.uber.Uber.backend.system.services.Impl;
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.RiderDto;
import com.yasif.project.uber.Uber.backend.system.entities.Driver;
import com.yasif.project.uber.Uber.backend.system.entities.Ride;
import com.yasif.project.uber.Uber.backend.system.entities.RideRequest;
import com.yasif.project.uber.Uber.backend.system.entities.enums.RideRequestStatus;
import com.yasif.project.uber.Uber.backend.system.entities.enums.RideStatus;
import com.yasif.project.uber.Uber.backend.system.exceptions.ResourceNotFoundException;
import com.yasif.project.uber.Uber.backend.system.repositories.DriverRepository;
import com.yasif.project.uber.Uber.backend.system.services.DriverService;
import com.yasif.project.uber.Uber.backend.system.services.RideRequestService;
import com.yasif.project.uber.Uber.backend.system.services.RideService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.modelmapper.ModelMapper;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.time.LocalDateTime;
import java.util.List;
import java.util.stream.Collectors;
@Service
@RequiredArgsConstructor
```

```
@Slf4j
public class DriverServiceImpl implements DriverService {
    private final RideRequestService rideRequestService;
    private final DriverRepository driverRepository;
    private final RideService rideService;
    private final ModelMapper modelMapper;
    /**
     * Accept a pending ride request and create a Ride.
     * Ensures driver availability and persists it atomically.
     */
    @Transactional
    @Override
    public RideDto acceptRide(Long rideRequestId) {
        try {
            RideRequest rideRequest = rideRequestService.findRideRequestById(rideRequestId);
            if (!RideRequestStatus.PENDING.equals(rideRequest.getRideRequestStatus())) {
                throw new IllegalStateException("RideRequest cannot be accepted, status is: " +
rideRequest.getRideRequestStatus());
            }
            Driver currentDriver = getCurrentDriver();
            if (!currentDriver.isAvailable()) {
                throw new IllegalStateException("Driver is not available");
            }
            // Reserve driver
            currentDriver.setAvailable(false);
            Driver savedDriver = driverRepository.save(currentDriver);
            // Create the ride via RideService (assumed transactional)
            Ride ride = rideService.createsNewRide(rideRequest, savedDriver);
            // Optionally: publish an event here (driver accepted)
            log.info("Driver {} accepted rideRequest {}", savedDriver.getId(), rideRequestId);
            return modelMapper.map(ride, RideDto.class);
        } catch (Exception e) {
            log.error("Failed to accept rideRequest {}: {}", rideRequestId, e.getMessage(), e);
            throw new RuntimeException("Failed to accept ride request", e);
        }
    }
    /**
     * Cancel a ride (driver-initiated cancellation).
     * Business rules: can cancel only if ride is CONFIRMED or ONGOING? (adjust per policy)
     */
    @Transactional
    @Override
    public RideDto cancelRide(Long rideId) {
        Ride ride = rideService.getRideById(rideId);
        Driver driver = getCurrentDriver();
        if (!driver.equals(ride.getDriver())) {
```

```
            throw new IllegalStateException("Driver cannot cancel a ride they did not accept");
        }
        // allow cancel only when not COMPLETED or CANCELLED
        if (ride.getRideStatus() == RideStatus.COMPLETED || ride.getRideStatus() == RideStatus.CANCELLED) {
            throw new IllegalStateException("Ride cannot be cancelled, status: " + ride.getRideStatus());
        }
        ride.setEndedAt(LocalDateTime.now());
        Ride saved = rideService.updateRideStatus(ride, RideStatus.CANCELLED);
        // make driver available again
        driver.setAvailable(true);
        driverRepository.save(driver);
        log.info("Driver {} cancelled ride {}", driver.getId(), rideId);
        return modelMapper.map(saved, RideDto.class);
    }
    /**
     * Start a ride after verifying OTP and confirmed status
     */
    @Transactional
    @Override
    public RideDto startRide(Long rideId, String otp) {
        Ride ride = rideService.getRideById(rideId);
        Driver driver = getCurrentDriver();
        if (!driver.equals(ride.getDriver())) {
            throw new IllegalStateException("Driver cannot start the ride as they did not accept it earlier");
        }
        if (!RideStatus.CONFIRMED.equals(ride.getRideStatus())) {
            throw new IllegalStateException("Ride status is not CONFIRMED hence cannot be started, status: " +
ride.getRideStatus());
        }
        if (ride.getOtp() == null || !ride.getOtp().equals(otp)) {
            throw new IllegalArgumentException("Otp is not valid");
        }
        ride.setStartedAt(LocalDateTime.now());
        Ride savedRide = rideService.updateRideStatus(ride, RideStatus.ONGOING);
        log.info("Ride {} started by driver {}", rideId, driver.getId());
        return modelMapper.map(savedRide, RideDto.class);
    }
    /**
     * End an ongoing ride. Computes finalization and makes driver available.
     */
    @Transactional
    @Override
    public RideDto endRide(Long rideId) {
        Ride ride = rideService.getRideById(rideId);
        Driver driver = getCurrentDriver();
        if (!driver.equals(ride.getDriver())) {
```

```
            throw new IllegalStateException("Driver cannot end the ride as they did not accept it earlier");
        }
        if (!RideStatus.ONGOING.equals(ride.getRideStatus())) {
            throw new IllegalStateException("Ride is not ongoing and cannot be ended. Current status: " +
ride.getRideStatus());
        }
        ride.setEndedAt(LocalDateTime.now());
        // If i have fare calculation here, call it before completing the ride
        // e.g., rideService.finalizeFare(ride);
        Ride savedRide = rideService.updateRideStatus(ride, RideStatus.COMPLETED);
        // make driver available again
        driver.setAvailable(true);
        driverRepository.save(driver);
        log.info("Ride {} ended by driver {}", rideId, driver.getId());
        return modelMapper.map(savedRide, RideDto.class);
    }
    /**
     * Rate the rider. This delegates rating persist to RideService or RiderService.
     * If not available, perform local update and persist.
     */
    @Transactional
    @Override
    public RideDto rateRider(Long rideId, Integer rating) {
        if (rating == null || rating < 1 || rating > 5) {
            throw new IllegalArgumentException("Rating must be between 1 and 5");
        }
        Ride ride = rideService.getRideById(rideId);
        Driver driver = getCurrentDriver();
        if (!driver.equals(ride.getDriver())) {
            throw new IllegalStateException("Driver cannot rate rider for a ride they did not accept");
        }
        if (!RideStatus.COMPLETED.equals(ride.getRideStatus())) {
            throw new IllegalStateException("Can rate rider only after ride is COMPLETED");
        }
        // TODO: implement rideService.rateRider(ride, rating) to persist rating and update rider aggregates
        // For now we attach rating to ride and delegate
        ride.setDriverRating(rating); // assuming there is a field for driver's rating of rider
        Ride saved = rideService.updateRideStatus(ride, ride.getRideStatus()); // persist change; adjust as per
my API
        log.info("Driver {} rated rider in ride {} with {}", driver.getId(), rideId, rating);
        return modelMapper.map(saved, RideDto.class);
    }
    /**
     * Get the profile of the currently authenticated driver
     */
    @Override
```

```
    public DriverDto getMyProfile() {
        Driver driver = getCurrentDriver();
        return modelMapper.map(driver, DriverDto.class);
    }
    /**
     * Return all rides assigned to this driver (historic and current)
     */
    @Override
    public List<RiderDto> getAllMyRides() {
        Driver driver = getCurrentDriver();
        List<Ride> rides = rideService.getAllRidesForDriver(driver.getId()); // assume this method exists
        return rides.stream()
                .map(r -> modelMapper.map(r, RiderDto.class)) // map Ride -> RiderDto or create a RideDto list instead
                .collect(Collectors.toList());
    }
    /**
     * Resolve the current driver from security context. Fallback to dev id if not authenticated.
     * Replace principal parsing to match my Authentication principal type.
     */
    @Override
    public Driver getCurrentDriver() {
        try {
            Authentication auth = SecurityContextHolder.getContext().getAuthentication();
            if (auth != null && auth.isAuthenticated()) {
                // Assuming the principal holds driver id (Long) or a custom UserPrincipal with getId()
                Object principal = auth.getPrincipal();
                Long driverId = null;
                // Example 1: principal is a Long (not typical)
                if (principal instanceof Long) {
                    driverId = (Long) principal;
                }
                // Example 2: principal is my custom UserPrincipal
                else if (principal instanceof org.springframework.security.core.userdetails.UserDetails) {
                    // adapt: i probably have a custom principal subclass with getId()
                    // driverId = ((myUserPrincipal) principal).getId();
                }
                if (driverId != null) {
                    return driverRepository.findById(driverId)
                            .orElseThrow(() -> new ResourceNotFoundException("Driver not found with id: " + driverId));
                }
            }
        } catch (Exception e) {
            log.warn("Could not resolve driver from SecurityContext: {}", e.getMessage());
        }
        // DEV fallback (remove/change in prod)
```

```
        log.warn("Using DEV fallback driver id=2L. Replace with real authentication lookup.");
        return driverRepository.findById(2L).orElseThrow(() -> new ResourceNotFoundException("Driver not found with id: 2"));
    }
}
```

6) Notes on the rewrite

- rideService assumed helper methods:
- getRideById(Long) — used here.
- createsNewRide(RideRequest, Driver) — i already had.
- updateRideStatus(Ride, RideStatus) — i already used.
- getAllRidesForDriver(Long) — I assume i'll add this to RideService.
- Consider adding rateRider(Ride, int) in RideService or RiderService to centralize rating logic and rider aggregate updates.
- Replace the principal parsing in getCurrentDriver() with how my authentication is actually implemented (JWT subject claim, UserPrincipal, etc.)
- ride.setDriverRating is speculative — align with my domain model (maybe ride has driverRated fields or i persist rating directly to Rider).

7) Next actionable improvements i can request now

- Add optimistic locking example for RideRequest to prevent double-accept.
- Implement event publishing (Spring ApplicationEvent or Kafka) for driver accept/cancel/start/end.
- Wire up Micrometer metrics and Grafana dashboard suggestions for driver operations.
- Add unit tests + Mockito examples for DriverServiceImpl.
- Convert everything to reactive/non-blocking if my stack requires it.

RideServiceImpl

```java
@Service  no usages  Yasif khan *
@RequiredArgsConstructor
public class RideServiceImpl implements RideService {

    private final RideRequestService rideRequestService;
    private final RideRepository rideRepository;
    private final ModelMapper modelMapper;


    @Override  1 usage  Yasif khan *
    public Ride getRideById(Long rideId) {
        return rideRepository.findById(rideId).orElseThrow(
                ()-> new ResourceNotFoundException("Ride is not found with id:"+rideId)
        );
    }

    @Override  no usages  Yasif khan
    public void matchWithDrivers(RideRequestDto rideRequestDto) {

    }
```

```java
    @Override  1 usage  new *
    public Ride createsNewRide( @NotNull RideRequest rideRequest, Driver driver) {
        rideRequest.setRideRequestStatus(RideRequestStatus.CONFIRMED);

        Ride ride = modelMapper.map(rideRequest,Ride.class);
        ride.setRideStatus(RideStatus.CONFIRMED);
        ride.setDriver(driver);
        ride.setOtp(generateRandomOTP());

        rideRequestService.update(rideRequest);

        return rideRepository.save(ride);
    }

    @Override  1 usage  new *
    public Ride updateRideStatus( @NotNull Ride ride, RideStatus rideStatus) {

        ride.setRideStatus(rideStatus);

        return rideRepository.save(ride);
    }
```

Overview — what this class is

RideServiceImpl is the service that owns **ride lifecycle logic** in my app. It:

- Creates rides when a driver accepts a ride request,
- Updates ride status,
- Generates OTPs for rider/driver verification,
- Looks up rides by id and (intended) by rider/driver with pagination,
- Has a placeholder for the driver-matching orchestration.

It sits between controllers, the RideRepository, and RideRequestService.

Fields / dependencies

```
private final RideRequestService rideRequestService;
private final RideRepository rideRepository;
private final ModelMapper modelMapper;
```

- rideRequestService — delegate for creating/updating/fetching RideRequest. Separates concerns: requests vs rides.
- rideRepository — JPA repository to persist and query Ride entities.
- modelMapper — maps DTO ↔ entity. Convenient but can silently copy fields; be explicit if sensitive fields exist.

Utilities / constants

```
private static final SecureRandom SECURE_RANDOM = new SecureRandom();
private static final int OTP_LENGTH = 4;
private static final int OTP_MAX = 10_000;
```

- Uses SecureRandom for OTP generation (good — more secure than Random).

- OTP is a zero-padded 4-digit number (0000–9999).

Method: getRideById(Long rideId)

```
return rideRepository.findById(rideId)
    .orElseThrow(() -> new ResourceNotFoundException("Ride is not found with id:"+rideId));
```

- Simple lookup; throws ResourceNotFoundException for not found — appropriate for 404 semantics.
- Important: ensure exception maps to HTTP 404 in my controller advice.

Method: matchWithDrivers(RideRequestDto rideRequestDto)

- **Status:** empty placeholder in my code.
- **Intent:** orchestrate driver matching (persist request, pick candidate drivers using strategy manager, notify drivers, manage the allocation window, accept the first driver).
- **Action required:** implement async matching pipeline (persist request → query drivers → push notifications → handle accept or timeout). Right now it does nothing.

Method: createsNewRide(RideRequest rideRequest, Driver driver)

What it does (step-by-step):

1. rideRequest.setRideRequestStatus(RideRequestStatus.CONFIRMED); — marks the request as claimed.
2. Ride ride = modelMapper.map(rideRequest,Ride.class); — maps request → ride.
3. ride.setRideStatus(RideStatus.CONFIRMED); — sets ride state.
4. ride.setDriver(driver); — attaches the accepting driver.
5. ride.setOtp(generateRandomOTP()); — creates a 4-digit OTP.
6. rideRequestService.update(rideRequest); — persists request state change.
7. return rideRepository.save(ride); — persists the ride and returns it.

Why it matters:

- This is the canonical “driver accepted → create ride” flow.

- OTP helps rider verify driver before starting.

  Risks / gaps:

- No @Transactional annotation in my original snippet — if rideRequestService.update and rideRepository.save are separate DB commits, partial failure is possible (request confirmed but ride not saved).
- modelMapper.map might copy unwanted fields (IDs, timestamps). i explicitly set id? In original i didn't — so ride could accidentally inherit request id depending on mapping rules.
- No optimistic locking on RideRequest — race condition: two drivers may confirm same request concurrently.
- OTP stored in entity possibly plaintext — consider security & expiry.

Method: updateRideStatus(Ride ride, RideStatus rideStatus)

- Sets ride.setRideStatus(rideStatus) and saves the ride.
- Simple state transition; caller should set startedAt/endedAt before calling if timestamps needed.
- Add @Transactional to ensure atomic save if not already covered upstream.

Methods: getAllRidesOfRider and getAllRidesOfDriver

- **Status:** return null in my code (not implemented).
- **Intent:** should return paged results using rideRepository (e.g., findByRiderId / findByDriverId).
- **Action required:** implement repository methods and these service calls, ensure PageRequest is handled (null defaults, validation).

Helper: generateRandomOTP()

```
Random random = new Random();
int otpInt = random.nextInt(10000);
return String.format("%04d",otpInt);
```

- Generates zero-padded 4-digit OTP.
- In my earlier file i used Random; better to use SecureRandom for unpredictability as noted above.

- Also: OTP needs expiry, attempt limits, and ideally storage in a short-lived cache (Redis) rather than persistent entity (security/hygiene).

Cross-cutting failure modes & operational concerns

1. **Race conditions / double-accept** — no optimistic locking on RideRequest. Add @Version and handle OptimisticLockingFailureException.
2. **Transactionality** — create/update flow should be in the same DB transaction: annotate createsNewRide with @Transactional.
3. **OTP security** — add expiry (otpGeneratedAt + TTL), attempt counters, and consider storing OTP hashed or in Redis.
4. **ModelMapper pitfalls** — explicit map for sensitive fields (IDs, audit columns) to avoid leaking or incorrect copying.
5. **Observability** — add logging, metrics (Micrometer counters for creates, starts, OTP failures), and events (RideCreated) so notifications/billing can react.
6. **Scalability** — driver matching must be asynchronous and distributed; implement via message queue or event-driven pipeline.
7. **Null/validation checks** — add Objects.requireNonNull or explicit validation to fail fast.
8. **Missing implementations** — matchWithDrivers, getAllRidesOfRider, getAllRidesOfDriver must be implemented for feature completeness.

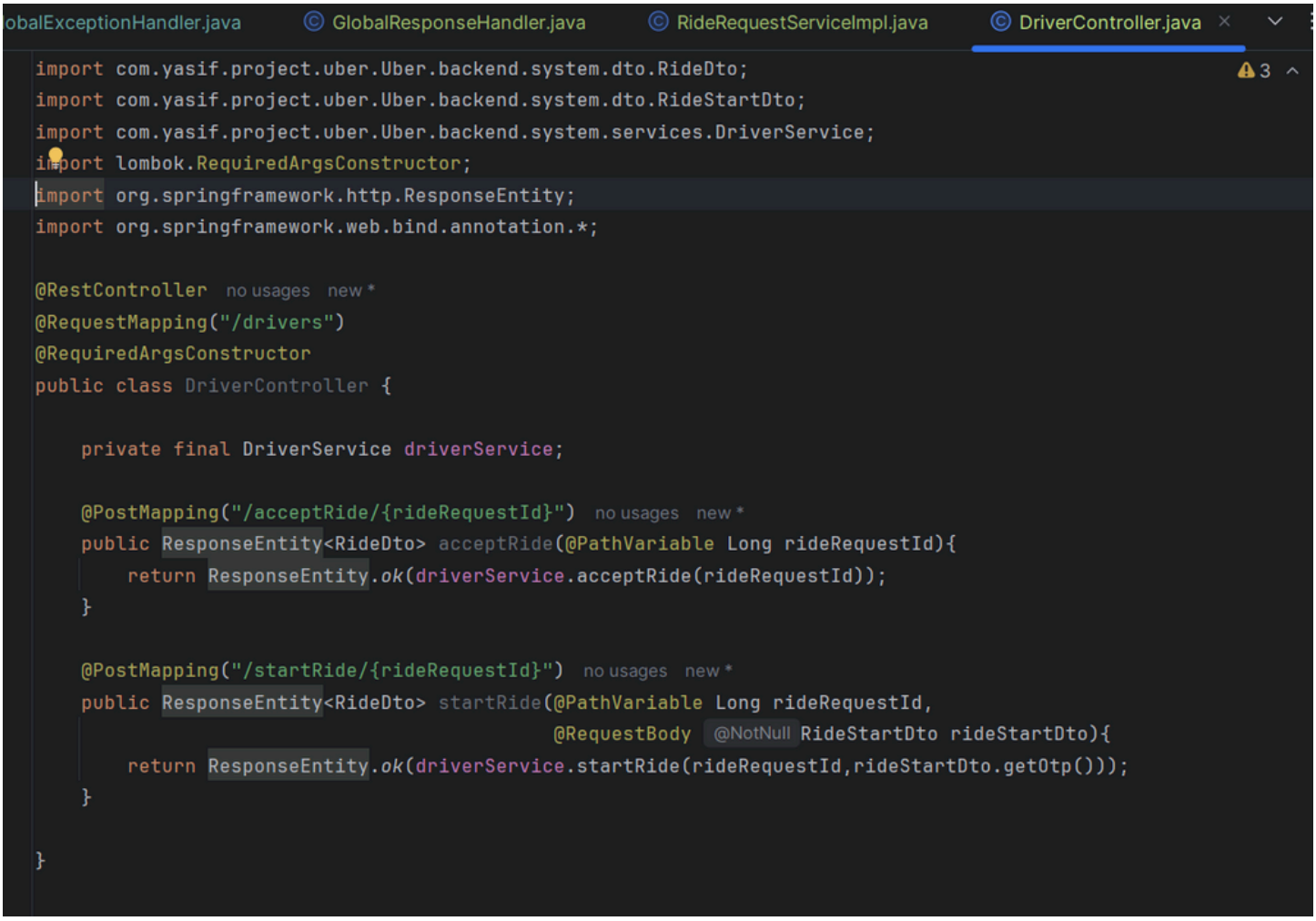
Quick prioritized remediation checklist (do these first)

1. Add @Transactional to createsNewRide (atomic update + create).
2. Add optimistic locking (@Version) to RideRequest and handle concurrency exceptions.
3. Replace Random with SecureRandom and add otpGeneratedAt field + expiry logic.
4. Implement getAllRidesOfRider/getAllRidesOfDriver using repository paging.
5. Implement matchWithDrivers as an async pipeline (persist request → enqueue notification → accept flow).
6. Add logging and publish RideCreated event for downstream systems.

One-line summary

i implemented the core ride-creation and status-update plumbing and OTP generation, but the code needs transactional boundaries, concurrency protection, secure OTP handling, and completion of matching/paging methods to be production-ready.

DriverController



Class-level

```
@RestController
@RequestMapping("/drivers")
@RequiredArgsConstructor
public class DriverController {
```

- @RestController:
- Marks this class as a Spring REST controller.
- Combines @Controller + @ResponseBody so all returned objects are automatically serialized to JSON.
- This is the entry point for HTTP requests related to drivers.
- @RequestMapping("/drivers"):
- Base URL path for all endpoints in this controller.

- So POST /drivers/acceptRide/{rideRequestId} and POST /drivers/startRide/{rideRequestId} map relative to this path.
- @RequiredArgsConstructor:
- Lombok annotation that generates a constructor with final fields as parameters.
- Allows Spring to inject DriverService via constructor-based dependency injection.
- private final DriverService driverService;:
- Injects the DriverService implementation.
- All controller endpoints delegate business logic to this service.

Endpoint 1: Accept a ride

```
@PostMapping("/acceptRide/{rideRequestId}")
public ResponseEntity<RideDto> acceptRide(@PathVariable Long rideRequestId){
   return ResponseEntity.ok(driverService.acceptRide(rideRequestId));
}
```

- @PostMapping("/acceptRide/{rideRequestId}"):
- Maps HTTP POST requests to this method.
- {rideRequestId} is a path variable representing the ride request that the driver wants to accept.
- @PathVariable Long rideRequestId:
- Extracts the rideRequestId from the URL and passes it into the method.
- driverService.acceptRide(rideRequestId):
- Calls the service layer to perform the ride acceptance logic.
- Responsibilities handled in DriverServiceImpl:

1. Check if the ride request status is PENDING.
2. Check if the driver is available.
3. Mark driver as unavailable.
4. Create a new ride linked to the driver.
5. Return RideDto as the response.

- ResponseEntity.ok(...):
- Wraps the RideDto in an HTTP 200 OK response.

Summary: This endpoint allows a driver to accept a ride request, triggers the creation of a ride, and returns ride details.

Endpoint 2: Start a ride

```
@PostMapping("/startRide/{rideRequestId}")
public ResponseEntity<RideDto> startRide(@PathVariable Long rideRequestId, @RequestBody
RideStartDto rideStartDto){
  return ResponseEntity.ok(driverService.startRide(rideRequestId, rideStartDto.getOtp()));
}
```

- @PostMapping("/startRide/{rideRequestId}"):
- Maps HTTP POST requests to this method.
- {rideRequestId} identifies the ride the driver wants to start.
- @RequestBody RideStartDto rideStartDto:
- Accepts a JSON payload in the request body.
- RideStartDto contains the OTP sent to the rider and/or driver for verification.
- driverService.startRide(rideRequestId, rideStartDto.getOtp()):
- Delegates to DriverService to:

1. Fetch the ride by ID.
2. Verify the driver is the one assigned to this ride.
3. Validate ride status is CONFIRMED.
4. Validate the OTP matches.
5. Update ride status to ONGOING and set startedAt timestamp.
6. Return updated RideDto.

- ResponseEntity.ok(...):
- Returns the updated ride information with HTTP 200 OK.

Summary: This endpoint starts a ride after verifying OTP and driver, updating the ride status to ONGOING.

Observations & Best Practices

1. **DTO usage:**

- RideDto ensures only necessary ride info is returned to the client.
- RideStartDto safely accepts OTP input without exposing the ride entity directly.

2. **HTTP status handling:**

- Currently, errors (like invalid OTP or ride not found) will throw exceptions, which need to be handled via @ControllerAdvice or @ExceptionHandler to return meaningful HTTP status codes (404, 400, 409, etc.).

3. **Security:**

- i should ensure that the driver calling these endpoints is authenticated and authorized to act on the ride request.
- Use Spring Security to restrict access so only the assigned driver can start or accept the ride.

4. **Validation:**

- @Valid could be added to @RequestBody to validate OTP format or other fields in RideStartDto.

5. **Idempotency:**

- Accepting a ride or starting a ride should be idempotent — repeated calls should not break the system.

Flow Summary

1. Driver hits /drivers/acceptRide/{rideRequestId} → triggers ride acceptance, ride creation, driver availability update.
2. Driver hits /drivers/startRide/{rideRequestId} with OTP → triggers ride start, OTP validation, ride status update.
3. Both endpoints return RideDto for client UI/feedback.

RiderController

GlobalResponseHandler.java | RideRequestServiceImpl.java | DriverController.java | RiderController.java

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

import com.yasif.project.uber.Uber.backend.system.dto.RideRequestDto;
import com.yasif.project.uber.Uber.backend.system.services.RiderService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController  no usages  Yasif khan
@RequestMapping("/riders")
@RequiredArgsConstructor
public class RiderController {

    private final RiderService riderService;

    @PostMapping("/requestRide")  no usages  Yasif khan
    public ResponseEntity<RideRequestDto> requestRide(@RequestBody RideRequestDto rideRequestDto){
        return ResponseEntity.ok(riderService.requestRide(rideRequestDto));
    }

}
```

Class-level

```
@RestController
@RequestMapping("/riders")
@RequiredArgsConstructor
public class RiderController {
```

- @RestController
- Marks this as a Spring REST controller.
- Combines @Controller and @ResponseBody, so all returned objects are automatically serialized to JSON.
- @RequestMapping("/riders")
- Base URL for all endpoints in this controller.
- All requests for rider actions will be prefixed with /riders.
- @RequiredArgsConstructor
- Lombok annotation to generate a constructor for all final fields.
- Allows Spring to inject RiderService via constructor.
- private final RiderService riderService;
- Injects the service layer handling rider operations.
- Keeps controller thin — all business logic lives in the service.

Endpoint: Request a ride

```
@PostMapping("/requestRide")
public ResponseEntity<RideRequestDto> requestRide(@RequestBody RideRequestDto rideRequestDto){
    return ResponseEntity.ok(riderService.requestRide(rideRequestDto));
}
```

- @PostMapping("/requestRide")
- Maps HTTP POST requests to /riders/requestRide.
- @RequestBody RideRequestDto rideRequestDto
- Accepts JSON request body from the rider with ride details, such as pickup location, drop-off location, etc.
- Spring automatically deserializes JSON into RideRequestDto.
- riderService.requestRide(rideRequestDto)
- Delegates the ride creation logic to the service layer (RiderService).
- Typically, this service will:

1. Validate the request data.
2. Persist a new RideRequest in the database.

3. Possibly trigger driver-matching asynchronously (via RideStrategyManager and RideService).

- ResponseEntity.ok(...)
- Wraps the resulting RideRequestDto in an HTTP 200 OK response.

Observations & Best Practices

1. **DTO Usage**

- Using RideRequestDto ensures the controller does not expose internal entities directly.

2. **Validation**

- i could add @Valid to @RequestBody to validate fields (like pickup and drop-off locations).

3. **Security / Authentication**

- Ensure only authenticated riders can request rides.
- The service should associate the ride request with the currently logged-in rider.

4. **Asynchronous Driver Matching**

- Right now, the controller simply creates the ride request.
- Actual driver matching should happen asynchronously in the service layer to avoid blocking the API.

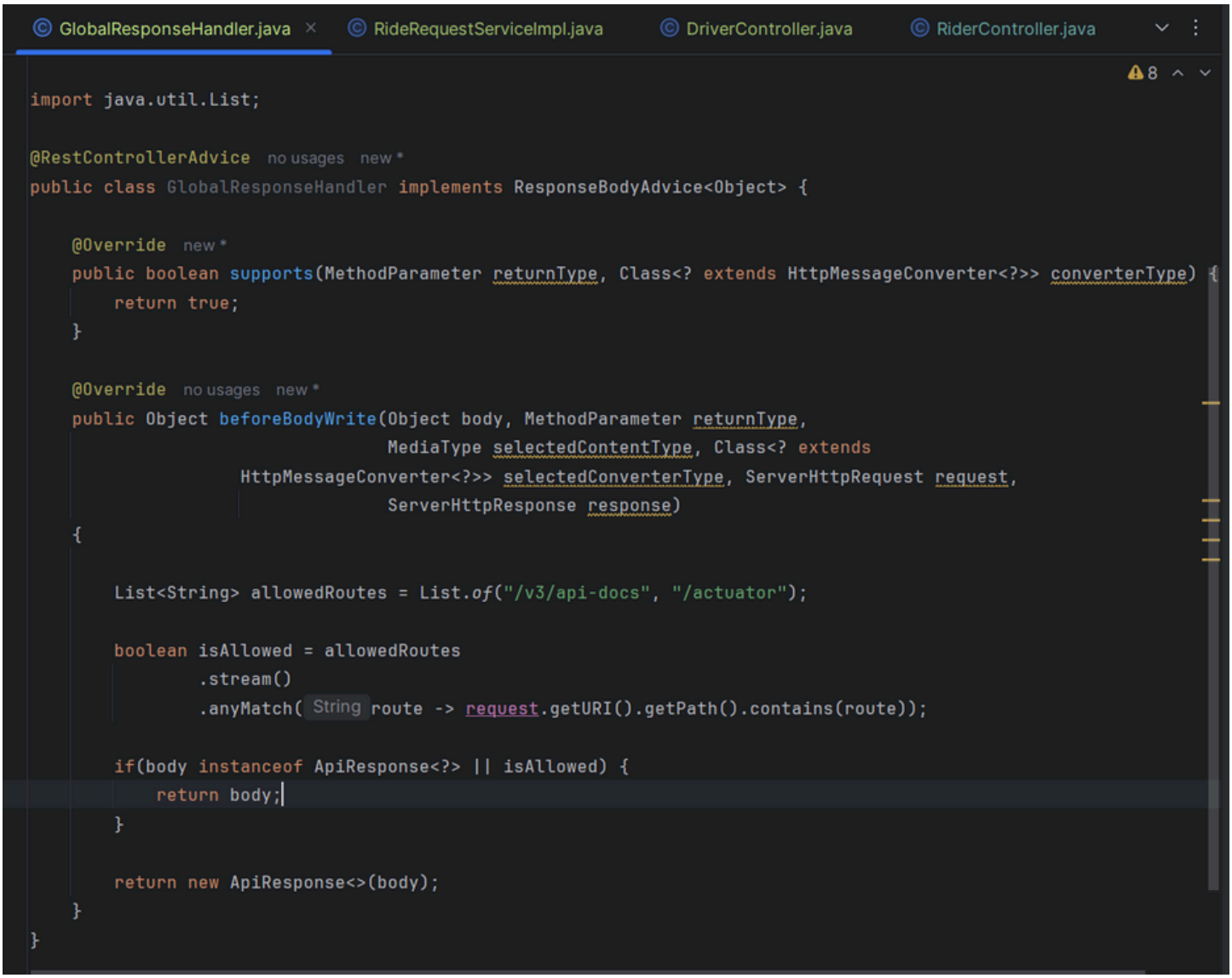
5. **Error Handling**

- If ride creation fails (invalid coordinates, database issue, no available drivers), proper HTTP error codes should be returned via @ControllerAdvice.

Flow Summary

1. Rider sends POST request to /riders/requestRide with ride info.
2. Controller deserializes the request to RideRequestDto.
3. Controller calls riderService.requestRide().
4. Service persists the ride request and may trigger driver-matching logic.
5. Controller returns RideRequestDto to the rider in HTTP 200 OK.

GlobalResponseHandler



Class-level

```
@RestControllerAdvice
public class GlobalResponseHandler implements ResponseBodyAdvice<Object> {
```

- @RestControllerAdvice:
- A specialized @ControllerAdvice that applies to all @RestControllers.
- Can intercept responses before they are sent to the client.
- implements ResponseBodyAdvice<Object>:
- Allows intercepting and modifying the response body globally.
- Generic <Object> means it applies to all response types.

Purpose: Wrap all API responses in a consistent structure (ApiResponse) unless they are already wrapped or on certain excluded routes.

Method: supports

```
@Override
public boolean supports(MethodParameter returnType, Class<? extends HttpMessageConverter<?>>
converterType) {
   return true;
}
```

- Determines which controller responses should be intercepted.
- Returning true means **all responses** are intercepted.
- Spring calls beforeBodyWrite only for responses that match this method.

Method: beforeBodyWrite

```
@Override
public Object beforeBodyWrite(Object body, MethodParameter returnType,
                    MediaType selectedContentType, Class<? extends
               HttpMessageConverter<?>> selectedConverterType, ServerHttpRequest request,
                    ServerHttpResponse response)
```

- Invoked **just before the response is written to the HTTP response body**.
- Parameters:

1. body — the object returned by the controller.
2. returnType — the method signature of the controller.
3. selectedContentType — content type negotiated for response (e.g., JSON).
4. selectedConverterType — converter used to serialize the response.
5. request — HTTP request object.
6. response — HTTP response object.

Excluded routes

```
List<String> allowedRoutes = List.of("/v3/api-docs", "/actuator");
boolean isAllowed = allowedRoutes
     .stream()
     .anyMatch(route -> request.getURI().getPath().contains(route));
```

- Maintains a list of routes that **should not be wrapped** in ApiResponse.
- /v3/api-docs → Swagger/OpenAPI JSON.
- /actuator → Spring Actuator endpoints.
- isAllowed will be true if the request path contains any of these routes.

Why: Wrapping these endpoints would break tools like Swagger UI or monitoring dashboards.

Wrapping responses

```
if(body instanceof ApiResponse<?> || isAllowed) {
   return body;
}
return new ApiResponse<>(body);
```

- If the response body is **already an ApiResponse**, or the request is on an **excluded route**, return it as-is.
- Otherwise, wrap the response in a standard ApiResponse object.

ApiResponse is presumably a generic wrapper like:

```
public class ApiResponse<T> {
   private T data;
   private String status = "success";
   private String message = null;
public ApiResponse(T data) {
      this.data = data;
   }
}
```

- This provides a consistent structure for all API responses.
- Benefits:
- Clients always know the format (data, status, message).
- Easier error handling and front-end integration.
- Makes future response metadata (pagination, timestamps) easy to add globally.

Summary

1. GlobalResponseHandler intercepts **all controller responses**.
2. Excludes certain routes (/v3/api-docs, /actuator) from wrapping.
3. Wraps everything else in ApiResponse to standardize the API response structure.
4. Prevents duplication if a response is already an ApiResponse.

Notes / Best Practices

1. i could make the excluded route check **configurable** via properties for easier maintenance.
2. Consider handling **null bodies** to avoid sending { "data": null } if not desired.

3. Works well with my DriverController and RiderController because all their responses will automatically be wrapped in a consistent JSON structure.

RideStartDto

Class-level

```
@Data
public class RideStartDto {
```

- @Data (from Lombok):
- Generates boilerplate code automatically:
- Getters and setters for all fields
- toString() method
- equals() and hashCode() methods
- A default constructor
- Makes the class simple and concise.
- public class RideStartDto:
- A simple **Data Transfer Object (DTO)** used to carry data from the client to the server.
- In this case, it represents the data needed to **start a ride**.

Field

```
String otp;
```

- otp stands for **One-Time Password**, which is used to verify the ride before starting.

- This field is expected in the JSON request body sent by the driver when starting a ride.

  Example JSON request:

  ```
  {
    "otp": "1234"
  }
  ```

- Spring automatically maps this JSON into an instance of RideStartDto because my controller method uses:

  `@RequestBody RideStartDto rideStartDto`

  ### Usage in my application

- Used in DriverController.startRide:

  `driverService.startRide(rideRequestId, rideStartDto.getOtp());`

- The OTP from this DTO is sent to the service layer to:

1. Verify that the driver starting the ride is authorized.
2. Check that the OTP matches the ride's OTP.
3. Allow the ride to be started (RideStatus.ONGOING) if verification passes.

   ### Summary

- RideStartDto is a **simple DTO** to carry the OTP from the front-end to my back-end.
- The @Data annotation avoids writing getters/setters manually.
- Its main purpose is to provide **type-safe access to the OTP** for ride-start validation.

Output

Request a Ride

POST request ride | POST accept ride | POST http://localhost:8080/ | POST start ride | No environment

Uber Api test / **request ride** — Save | Share

POST http://localhost:8080/riders/requestRide — **Send**

Docs | Params | Authorization | Headers (9) | Body | Scripts | Settings — Cookies

none | form-data | x-www-form-urlencoded | raw | binary | GraphQL JSON — Schema | Beautify

```
{
    "pickupLocation":{
        "coordinates":[
            74.8982,
            28.56471
        ]
    },
    "dropOffLocation":{
        "coordinates":[
            74.1789,
            28.1739843
        ]
    },
    "paymentMethod": "WALLET"
}
```

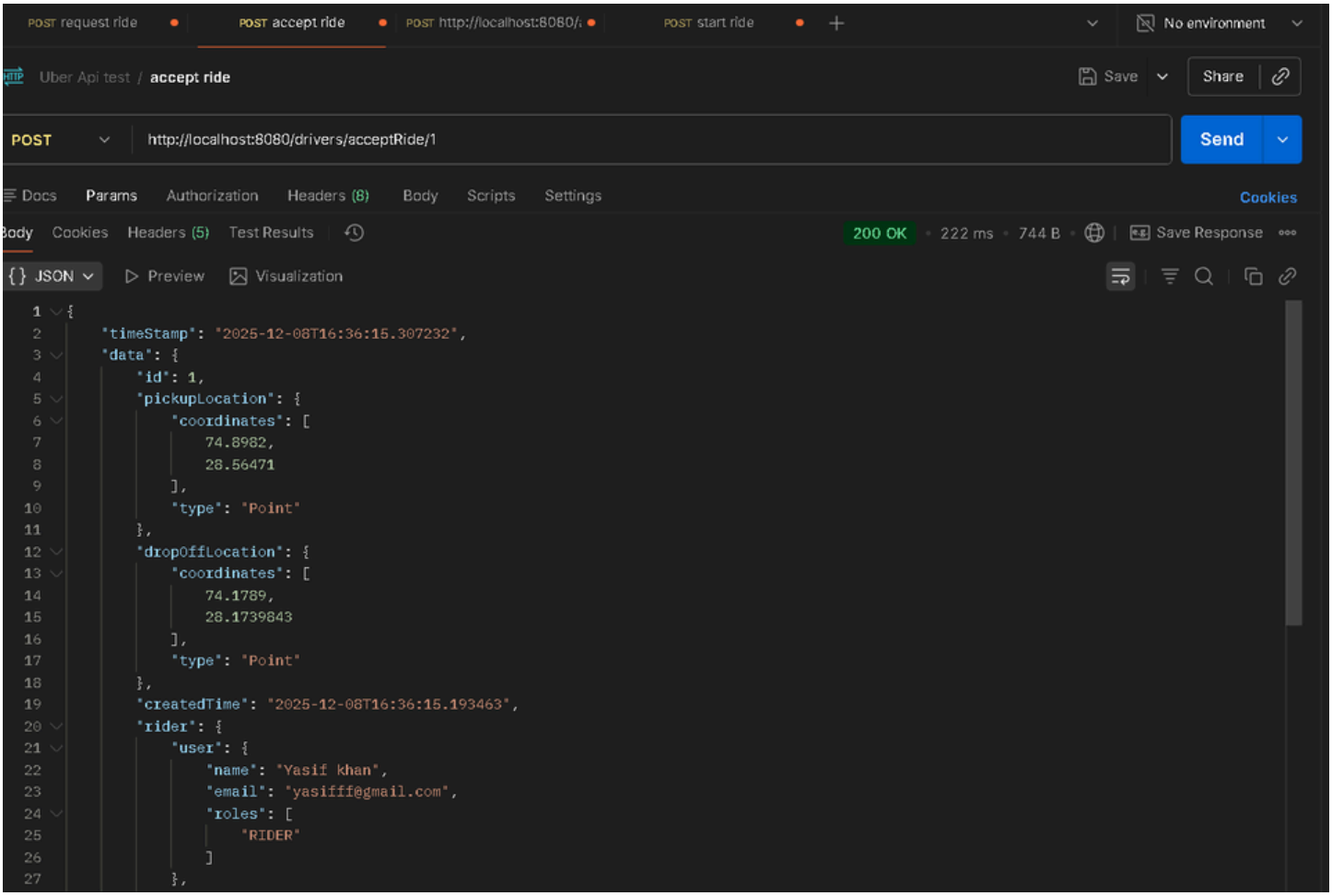
Response | History

Click Send to get a response


```
{
    "timeStamp": "2025-12-08T16:34:25.5256533",
    "data": {
        "id": 2,
        "pickupLocation": {
            "coordinates": [
                74.8982,
                28.56471
            ],
            "type": "Point"
        },
        "dropOffLocation": {
            "coordinates": [
                74.1789,
                28.1739843
            ],
            "type": "Point"
        },
        "requestedTime": "2025-12-08T16:34:25.231903",
        "rider": {
            "user": {
                "name": "Yasif khan",
                "email": "yasifff@gmail.com",
                "roles": [
                    "RIDER"
                ]
            },
```

```
            ]
        },
        "rating": 4.9
    },
    "paymentMethod": "WALLET",
    "rideRequestStatus": "PENDING",
    "fare": 2143.548
},
"apiError": null
}
```

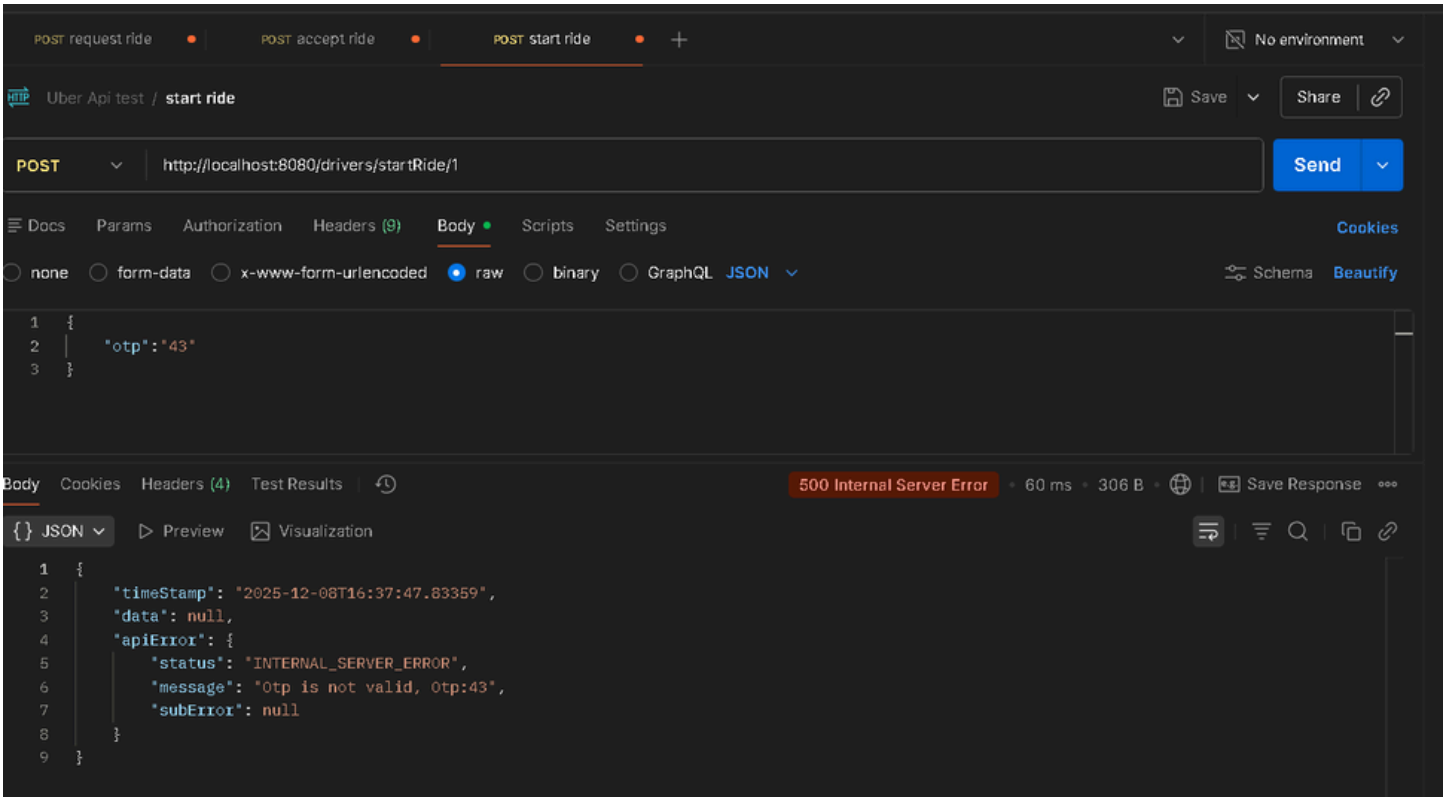

Accept Ride



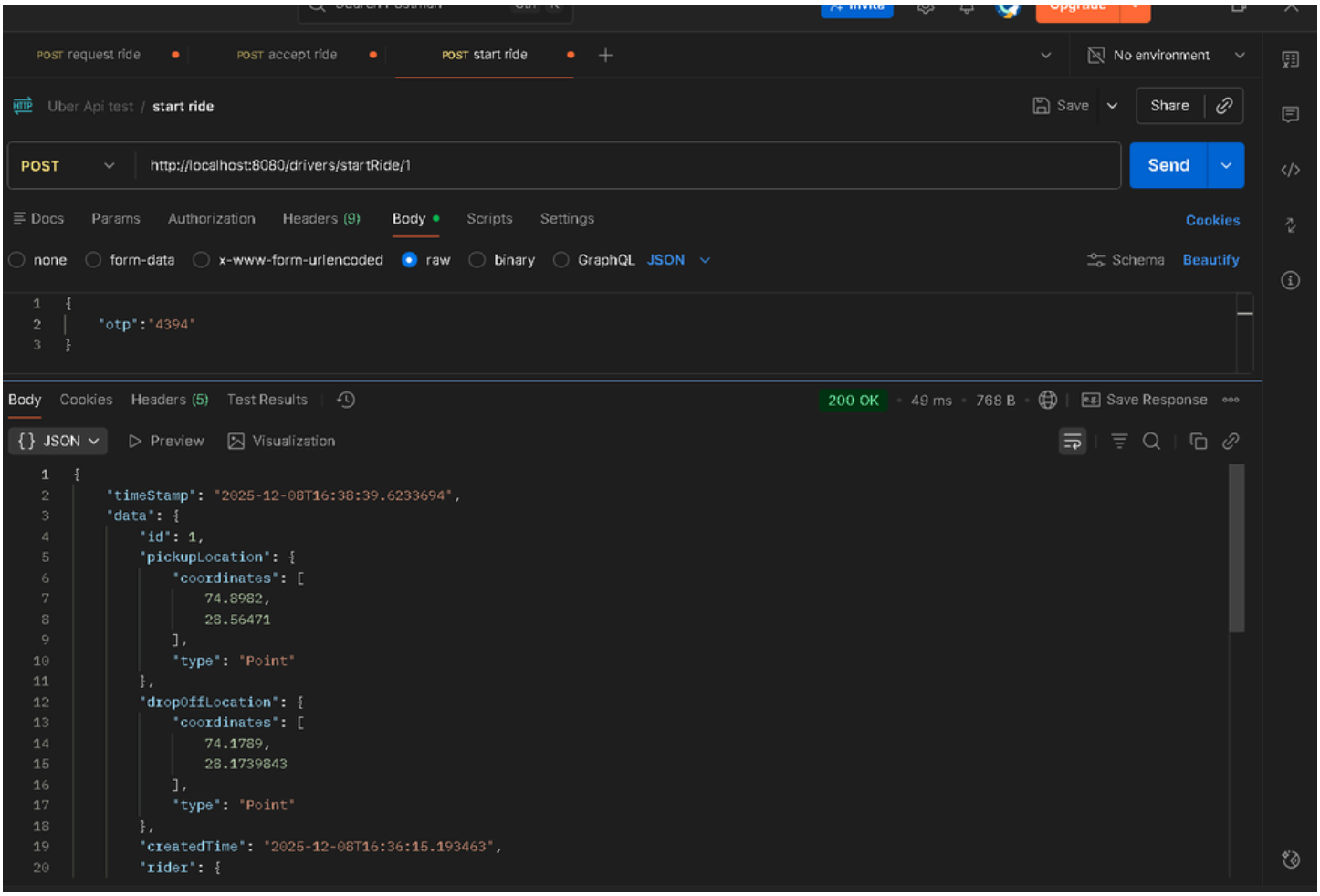

```
{
    "timeStamp": "2025-12-08T16:36:15.307232",
    "data": {
        "id": 1,
        "pickupLocation": {
            "coordinates": [
                74.8982,
                28.56471
            ],
            "type": "Point"
        },
        "dropOffLocation": {
            "coordinates": [
                74.1789,
                28.1739843
            ],
            "type": "Point"
        },
        "createdTime": "2025-12-08T16:36:15.193463",
        "rider": {
            "user": {
                "name": "Yasif khan",
                "email": "yasifff@gmail.com",
                "roles": [
                    "RIDER"
                ]
            },
```

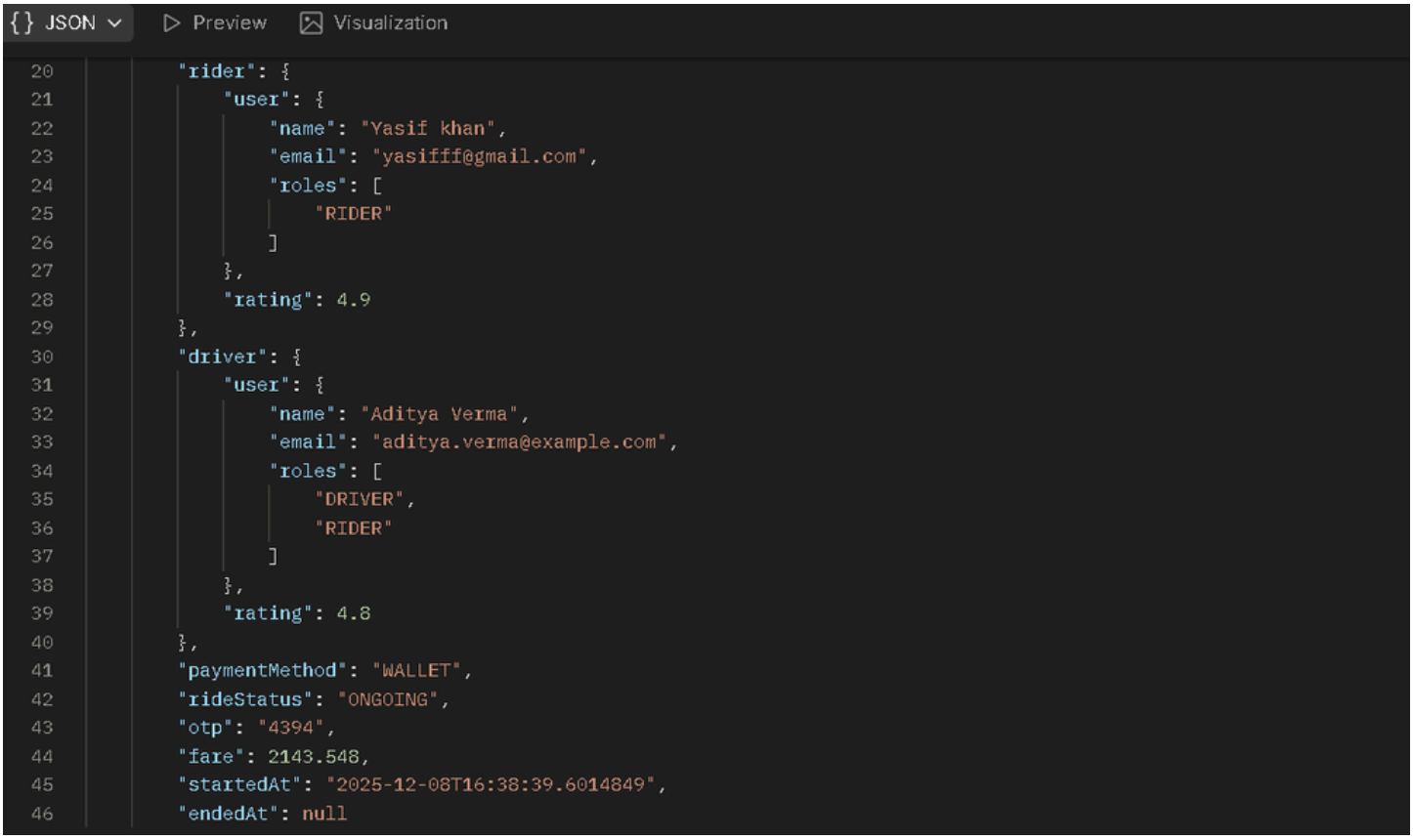
```
                ]
            },
            "rating": 4.9
        },
        "driver": {
            "user": {
                "name": "Aditya Verma",
                "email": "aditya.verma@example.com",
                "roles": [
                    "DRIVER",
                    "RIDER"
                ]
            },
            "rating": 4.8
        },
        "paymentMethod": "WALLET",
        "rideStatus": "CONFIRMED",
        "otp": "4394",
        "fare": 2143.548,
        "startedAt": null,
        "endedAt": null
    },
    "apiError": null
}
```


Start Ride : Enter Invalid OTP

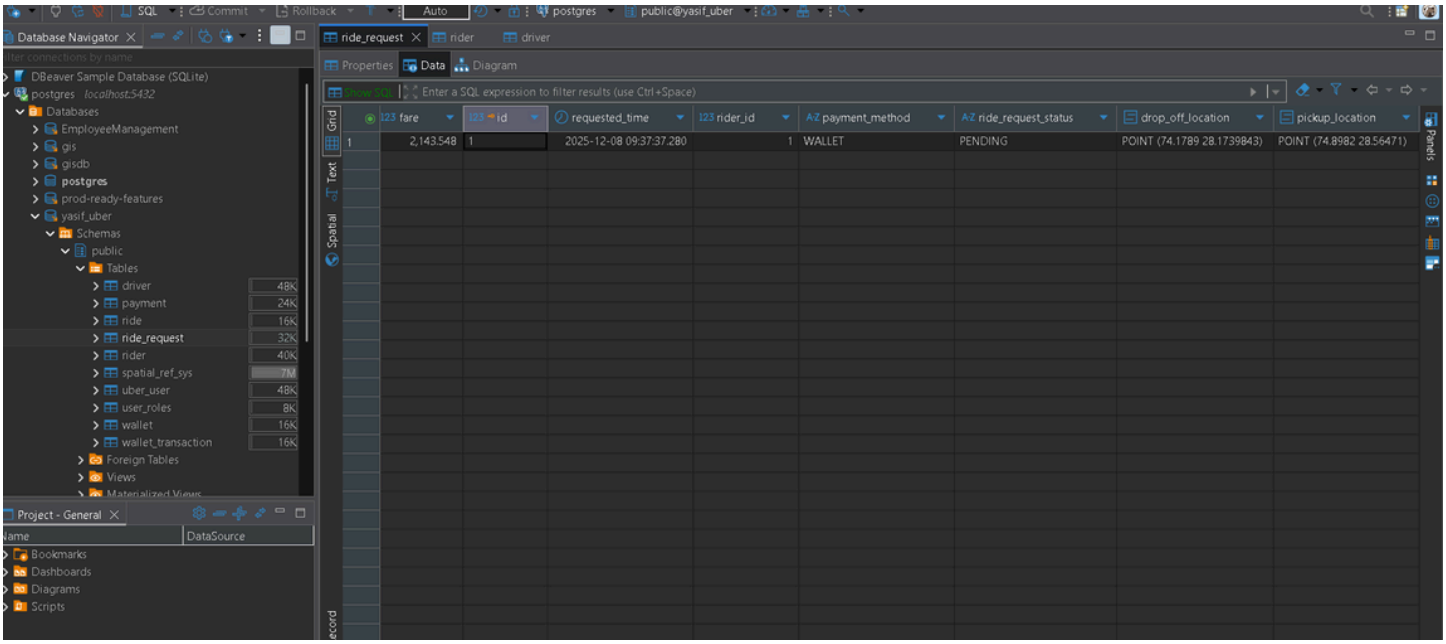


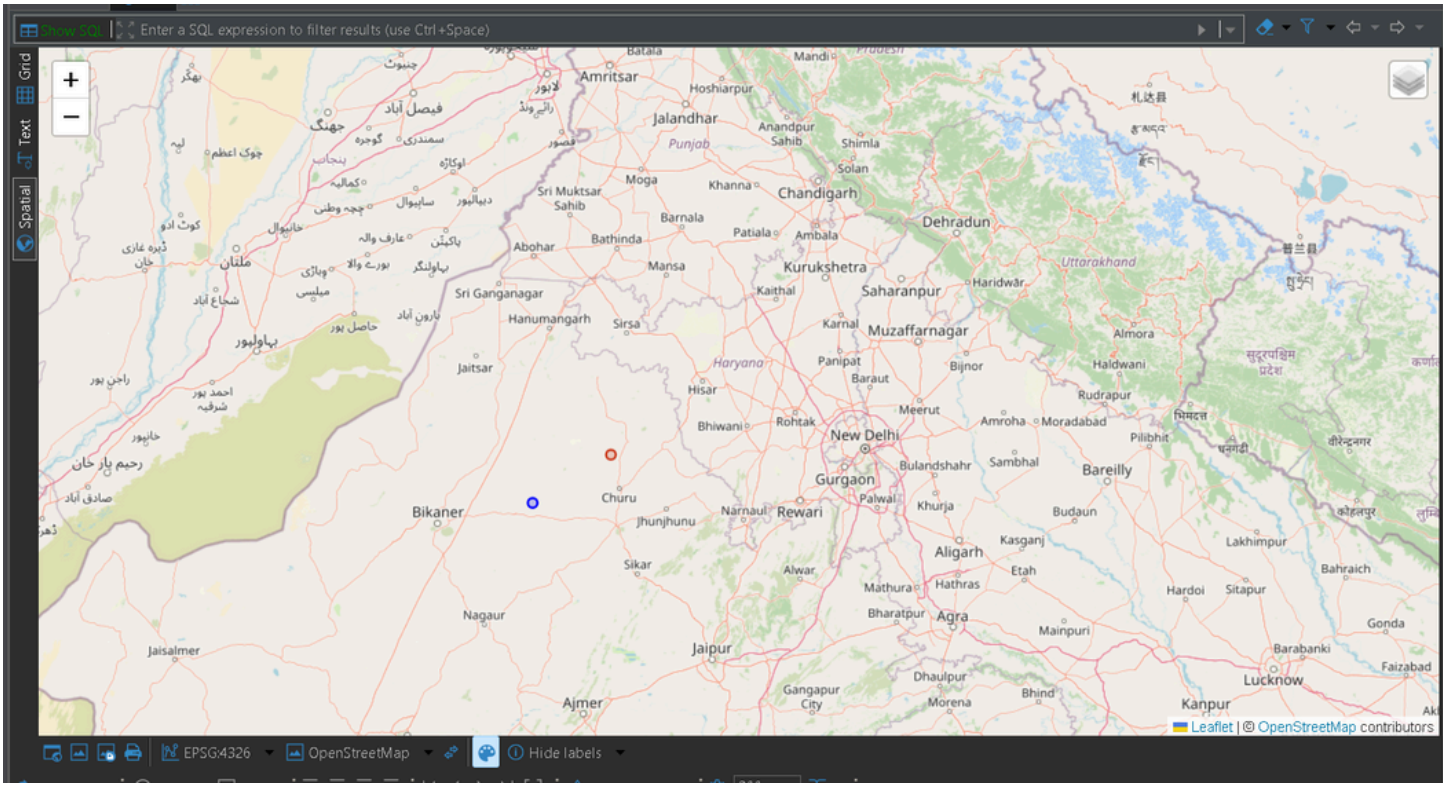
Start Ride : Enter valid OTP

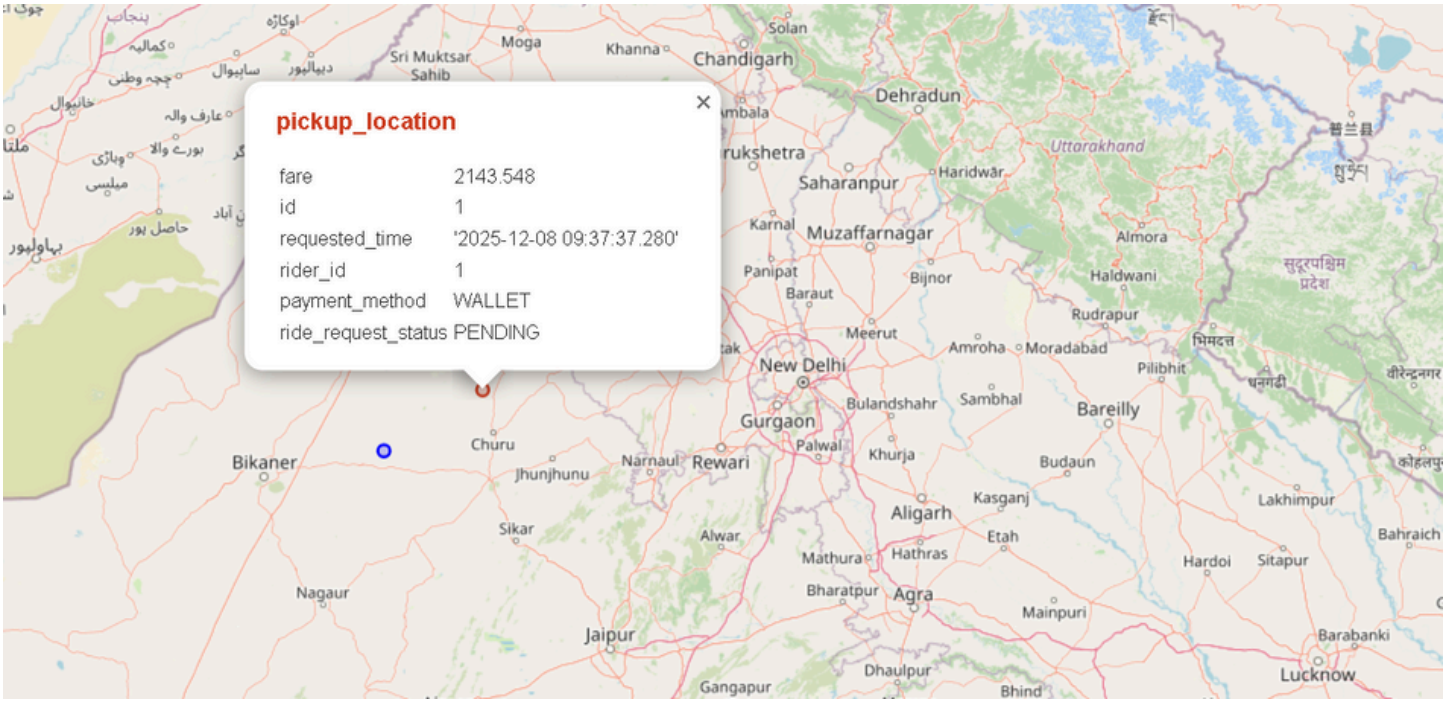

POST request ride
POST accept ride
POST start ride
No environment
Uber Api test / start ride
Save
Share
POST
http://localhost:8080/drivers/startRide/1
Send
Docs
Params
Authorization
Headers (9)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
{
"otp":"4394"
}
Body
Cookies
Headers (5)
Test Results
200 OK
49 ms
768 B
Save Response
{} JSON
Preview
Visualization
{
"timeStamp": "2025-12-08T16:38:39.6233694",
"data": {
"id": 1,
"pickupLocation": {
"coordinates": [
74.8982,
28.56471
],
"type": "Point"
},
"dropOffLocation": {
"coordinates": [
74.1789,
28.1739843
],
"type": "Point"
},
"createdTime": "2025-12-08T16:36:15.193463",
"rider": {

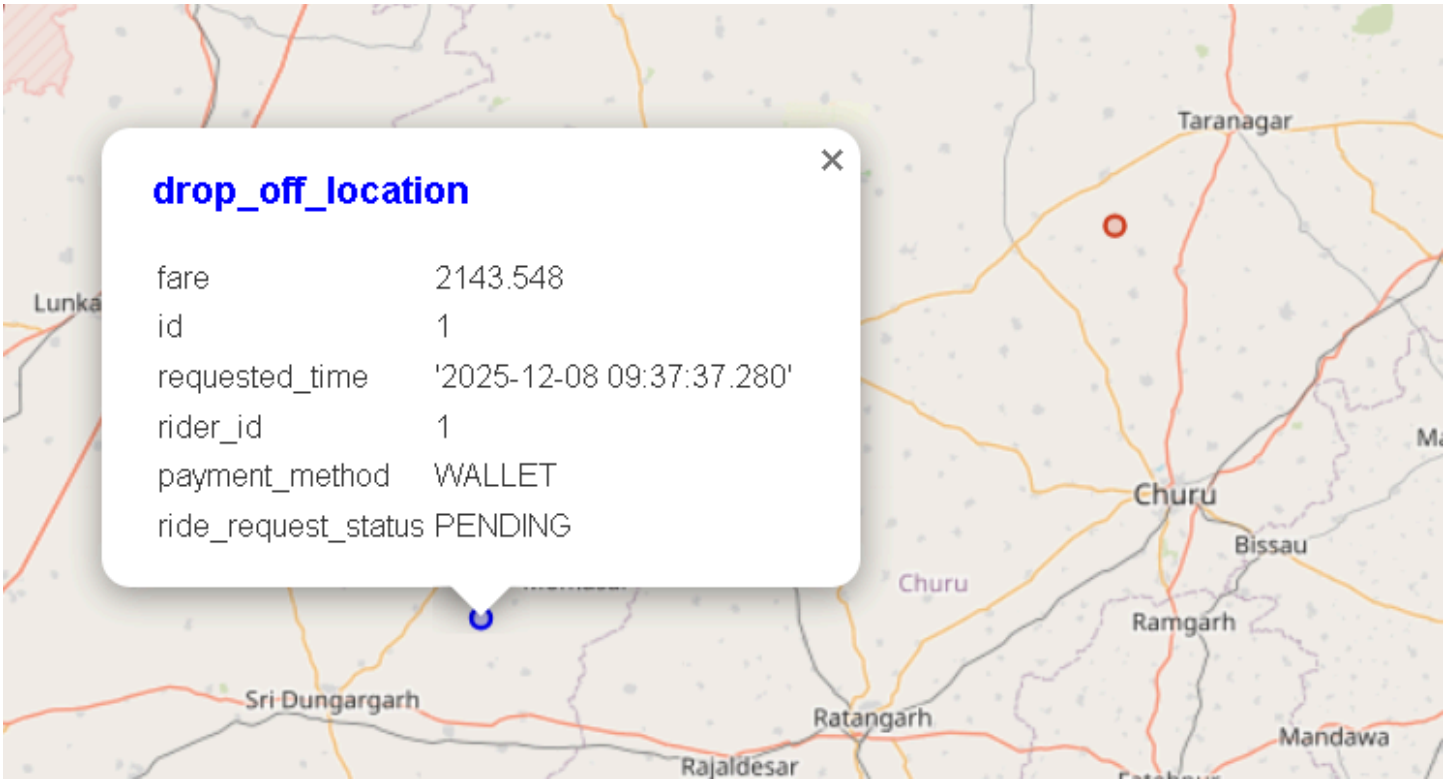

{} JSON
Preview
Visualization
"rider": {
"user": {
"name": "Yasif khan",
"email": "yasifff@gmail.com",
"roles": [
"RIDER"
]
},
"rating": 4.9
},
"driver": {
"user": {
"name": "Aditya Verma",
"email": "aditya.verma@example.com",
"roles": [
"DRIVER",
"RIDER"
]
},
"rating": 4.8
},
"paymentMethod": "WALLET",
"rideStatus": "ONGOING",
"otp": "4394",
"fare": 2143.548,
"startedAt": "2025-12-08T16:38:39.6014849",
"endedAt": null

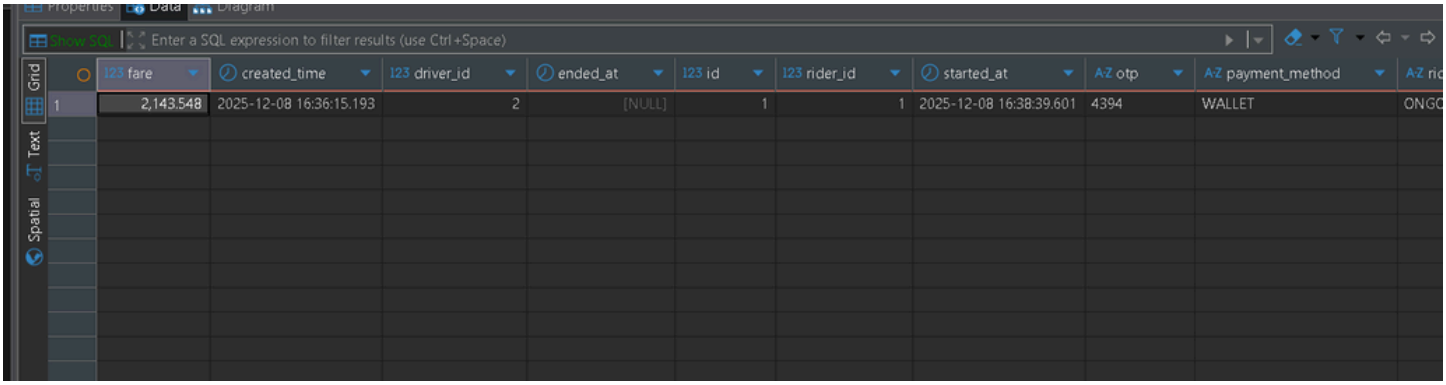
Databses:

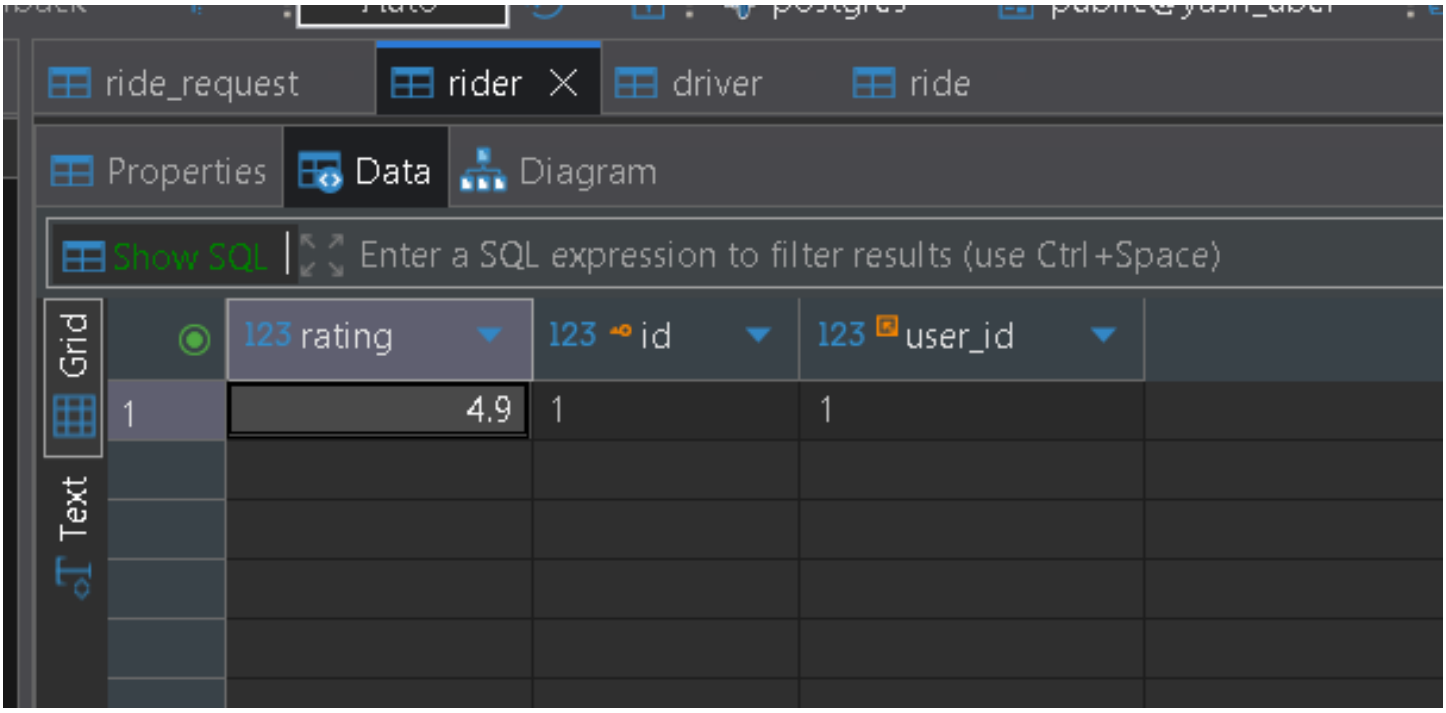

Database Navigator
DBeaver Sample Database (SQLite)
postgres localhost:5432
Databases
EmployeeManagement
gis
gisdb
postgres
prod-ready-features
yasif_uber
Schemas
public
Tables
driver 48K
payment 24K
ride 16K
ride_request 32K
rider 40K
spatial_ref_sys 7M
uber_user 48K
user_roles 8K
wallet 16K
wallet_transaction 16K
Foreign Tables
Views
Project - General
Name
DataSource
Bookmarks
Dashboards
Diagrams
Scripts
ride_request
rider
driver
Properties
Data
Diagram
Enter a SQL expression to filter results (use Ctrl+Space)
fare
id
requested_time
rider_id
payment_method
ride_request_status
drop_off_location
pickup_location
1
2,143.548
1
2025-12-08 09:37:37.280
1
WALLET
PENDING
POINT (74.1789 28.1739843)
POINT (74.8982 28.56471)


Enter a SQL expression to filter results (use Ctrl+Space)
Amritsar
Jalandhar
Hoshiarpur
Anandpur Sahib
Shimla
Solan
Punjab
Chandigarh
Moga
Khanna
Sri Muktsar Sahib
Barnala
Patiala
Ambala
Dehradun
Bathinda
Abohar
Mansa
Kurukshetra
Uttarakhand
Haridwar
Saharanpur
Kaithal
Sri Ganganagar
Hanumangarh
Sirsa
Karnal
Muzaffarnagar
Almora
Haldwani
Panipat
Baraut
Bijnor
Haryana
Jaitsar
Hisar
Rudrapur
Meerut
Amroha
Moradabad
Pilibhit
Bhiwani
Rohtak
New Delhi
Bareilly
Gurgaon
Palwal
Bulandshahr
Sambhal
Khurja
Budaun
Churu
Narnaul
Rewari
Jhunjhunu
Bikaner
Kasganj
Lakhimpur
Aligarh
Sikar
Alwar
Etah
Hathras
Mathura
Hardoi
Sitapur
Bahraich
Nagaur
Bharatpur
Agra
Mainpuri
Gonda
Jaisalmer
Jaipur
Barabanki
Faizabad
Lucknow
Dhaulpur
Gangapur City
Morena
Bhind
Ajmer
Kanpur
Leaflet | © OpenStreetMap contributors
EPSG:4326
OpenStreetMap
Hide labels


pickup_location
fare 2143.548
id 1
requested_time '2025-12-08 09:37:37.280'
rider_id 1
payment_method WALLET
ride_request_status PENDING
Solan
Chandigarh
Dehradun
Uttarakhand
Haridwar
Saharanpur
Muzaffarnagar
Almora
Haldwani
Rudrapur
Bijnor
Meerut
Amroha
Moradabad
Pilibhit
New Delhi
Bareilly
Gurgaon
Sambhal
Bulandshahr
Palwal
Khurja
Budaun
Aligarh
Kasganj
Lakhimpur
Etah
Hathras
Mathura
Hardoi
Sitapur
Agra
Bharatpur
Mainpuri
Bikaner
Churu
Jhunjhunu
Narnaul
Rewari
Sikar
Alwar
Nagaur
Jaipur
Dhaulpur
Gangapur
Bhind
Barabanki
Lucknow


drop_off_location
fare 2143.548
id 1
requested_time '2025-12-08 09:37:37.280'
rider_id 1
payment_method WALLET
ride_request_status PENDING
Taranagar
Churu
Bissau
Ramgarh
Sri Dungargarh
Ratangarh
Mandawa
Rajaldesar


Properties
Data
Diagram
Show SQL
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
Spatial
fare
created_time
driver_id
ended_at
id
rider_id
started_at
otp
payment_method
1
2,143.548
2025-12-08 16:36:15.193
2
[NULL]
1
1
2025-12-08 16:38:39.601
4394
WALLET


ride_request
rider
driver
ride
Properties
Data
Diagram
Show SQL
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
123 rating
123 id
123 user_id
1
4.9
1
1