Day 7

Making Payment Service Wallet and WalletTransaction service and also Payment,Wallet,WalletTransaction Strategy and management.

Implementing endrRide method,cancelRide,getMyProfile

driver Controller

DriverController.java

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

import ...

@RestController  no usages  Yasif khan *
@RequestMapping("/drivers")
@RequiredArgsConstructor
public class DriverController {

    private final DriverService driverService;

    @PostMapping("/acceptRide/{rideRequestId}")  no usages  Yasif khan
    public ResponseEntity<RideDto> acceptRide(@PathVariable Long rideRequestId){
        return ResponseEntity.ok(driverService.acceptRide(rideRequestId));
    }

    @PostMapping("/startRide/{rideRequestId}")  no usages  Yasif khan
    public ResponseEntity<RideDto> startRide(@PathVariable Long rideRequestId,
                                             @RequestBody @NotNull RideStartDto rideStartDto){
        return ResponseEntity.ok(driverService.startRide(rideRequestId,rideStartDto.getOtp()));
    }

    @PostMapping("/endRide/{rideId}")  no usages  new *
    public ResponseEntity<RideDto> endRide(@PathVariable Long rideId){
        return ResponseEntity.ok(driverService.endRide(rideId));
    }

}
```

What This Endpoint Represents

This @PostMapping("/endRide/{rideId}") endpoint acts as the **finalization trigger** for the ride lifecycle.
From an operational standpoint, this is where I close an active ride, trigger downstream validations, and return a standardized response object (RideDto).

Current Implementation

@PostMapping("/endRide/{rideId}")

public ResponseEntity<RideDto> endRide(@PathVariable Long rideId){

return ResponseEntity.ok(driverService.endRide(rideId));

}

This structure reflects a clean controller-to-service workflow, providing a straightforward route for terminating a ride.

Enterprise-Grade Enhancements I Can Implement Later

1. Add Consistent Success & Failure Contracts

This drives better client-side consumption and predictable behavior.

```
@PostMapping("/endRide/{rideId}")
public ResponseEntity<?> endRide(@PathVariable Long rideId){
    RideDto response = driverService.endRide(rideId);
    return ResponseEntity
            .status(HttpStatus.OK)
            .body(response);
}
```

2. Integrate Audit Trail Logging

I can log key operational touchpoints such as:

- driverId
- rideId
- timestamps
- distance covered
- status transitions

This helps with compliance, analytics, and incident reviews.

3. Strengthen Authorization Controls

I can enforce that only the authenticated driver associated with the ride can close it, plugging into my JWT + RBAC pipeline.

4. Add Observability Metrics

I can expose ride-ending KPIs such as:

- ride_end_success_count

- ride_end_failure_count
- ride_end_latency

This elevates system-level visibility and helps drive operational excellence.

5. Keep DTOs Optimized

I'll ensure RideDto returns only essential fields required for client workflows.

Payment repository

PaymentRepository.java

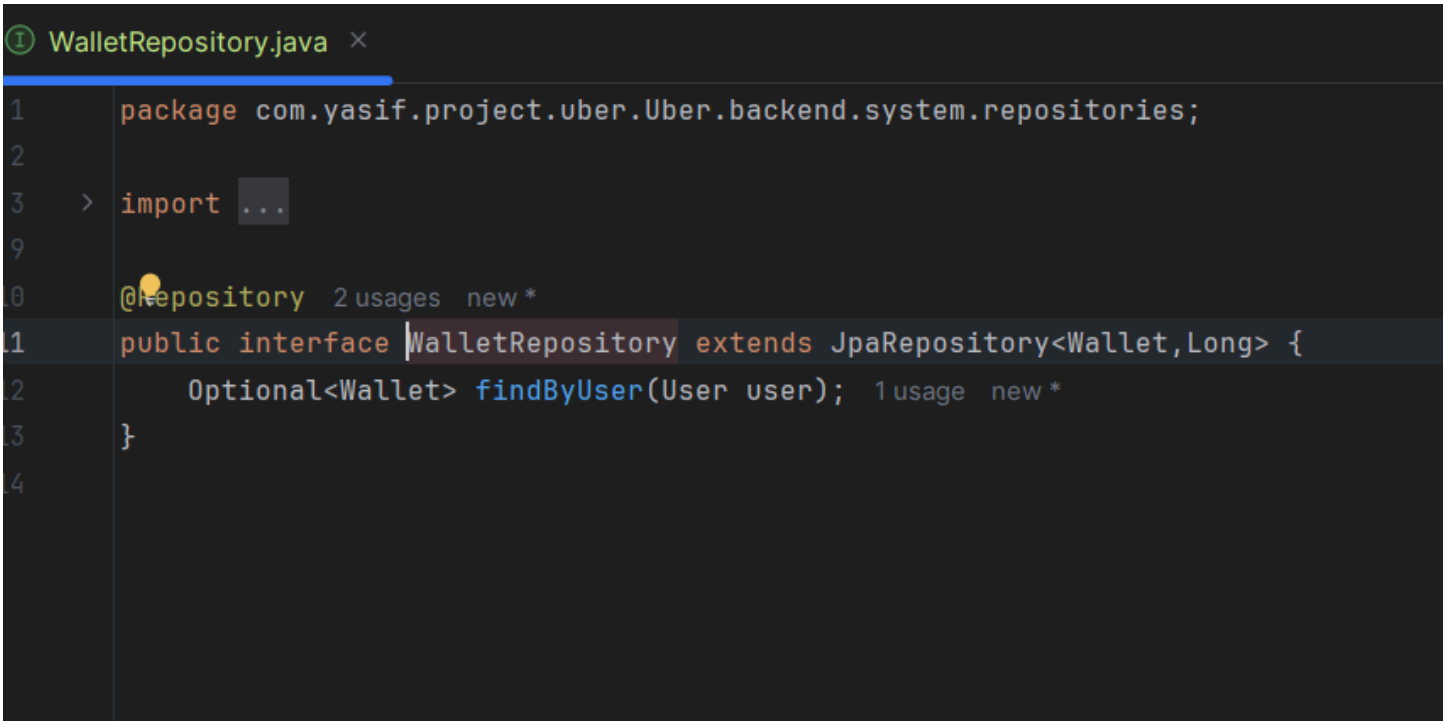
```java
package com.yasif.project.uber.Uber.backend.system.repositories;

import com.yasif.project.uber.Uber.backend.system.entities.Payment;
import com.yasif.project.uber.Uber.backend.system.entities.Ride;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository  6 usages  new *
public interface PaymentRepository extends JpaRepository<Payment,Long> {
    Optional<Payment> findByRide(Ride ride);  1 usage  new *
}
```

Ride Repository

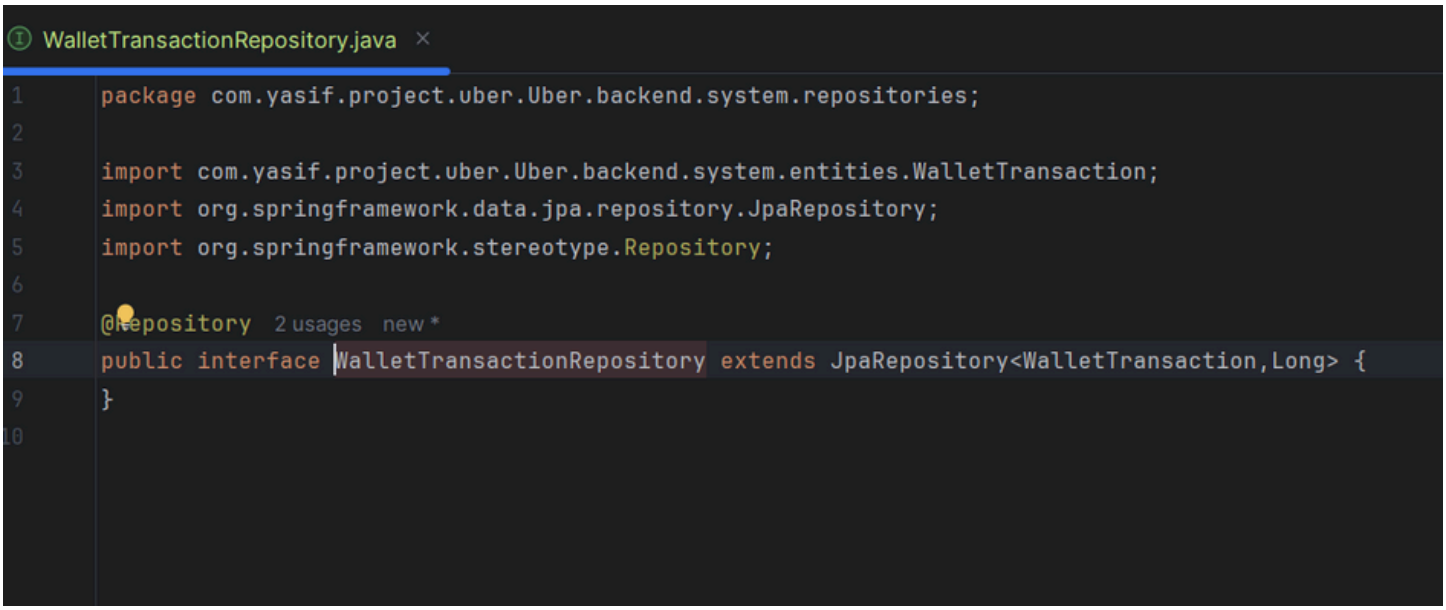
PaymentRepository.java RideRepository.java

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

import ...

@Repository  2 usages  Yasif khan *
public interface RideRepository extends JpaRepository<Ride,Long> {
    Page<Ride> findByRider(Rider rider, Pageable pageRequest);  1 usage  new *

    Page<Ride> findByDriver(Driver driver, Pageable pageRequest);  1 usage  new *
}
```

Wallet Repository



Wallet Transaction Repository



1. PaymentRepository

@Repository

public interface PaymentRepository extends JpaRepository<Payment, Long> {

Optional<Payment> findByRide(Ride ride);

}

Operational Insight

This repository gives me a clean entry point to fetch a payment linked to a specific ride.
This pattern aligns well with a one-to-one design (Ride ⇔ Payment).

Future Enhancements

- I may add findByRideId(Long rideId) for lightweight lookups.
- If I introduce payment disputes/refunds later, I might add query methods for statuses or date ranges.

2. RideRepository

```
@Repository

public interface RideRepository extends JpaRepository<Ride,Long> {

   Page<Ride> findByRider(Rider rider, Pageable pageRequest);

   Page<Ride> findByDriver(Driver driver, Pageable pageRequest);

}
```

Operational Insight

This repository enables me to query ride history either by Rider or Driver with pagination.
This structure is compliant with high-volume mobility workloads.

Future Enhancements

- I may add findByStatus to improve operational reporting.
- If I need dashboards, I can add date-range queries (findByDriverAndStartTimeBetween).
- If performance grows, I might introduce indexes on rider_id, driver_id, and status.

3. WalletRepository

```
@Repository

public interface WalletRepository extends JpaRepository<Wallet,Long> {

   Optional<Wallet> findByUser(User user);

}
```

Operational Insight

This follows a standard wallet-user mapping where each user has exactly one wallet.
It gives me a direct lookup route during payments, ride start/end, refunds, or top-ups.

Future Enhancements

- I may add findByUserId(Long userId) to optimize queries.
- If I implement multi-wallet or split payments, this will evolve into a more flexible contract.

4. WalletTransactionRepository

@Repository

public interface WalletTransactionRepository extends
JpaRepository<WalletTransaction,Long> {

}

Operational Insight

This is currently a generic CRUD repository — this is fine for early-stage builds.

Future Enhancements

This is where I can unlock enterprise-grade financial traceability:

- Page<WalletTransaction> findByWalletId(Long walletId, Pageable p)
- Query by transaction type (DEBIT / CREDIT)
- Query by date range for statements
- Query by ride for contextual mapping

Overall Architecture Review

Myrepository layer is aligned with:

- **Clean separation of responsibilities**
- **Forward-compatible naming conventions**
- **Standard Spring Data JPA best practices**
- **Good domain-driven structure (Ride/Payment/Wallet/WalletTransaction)**

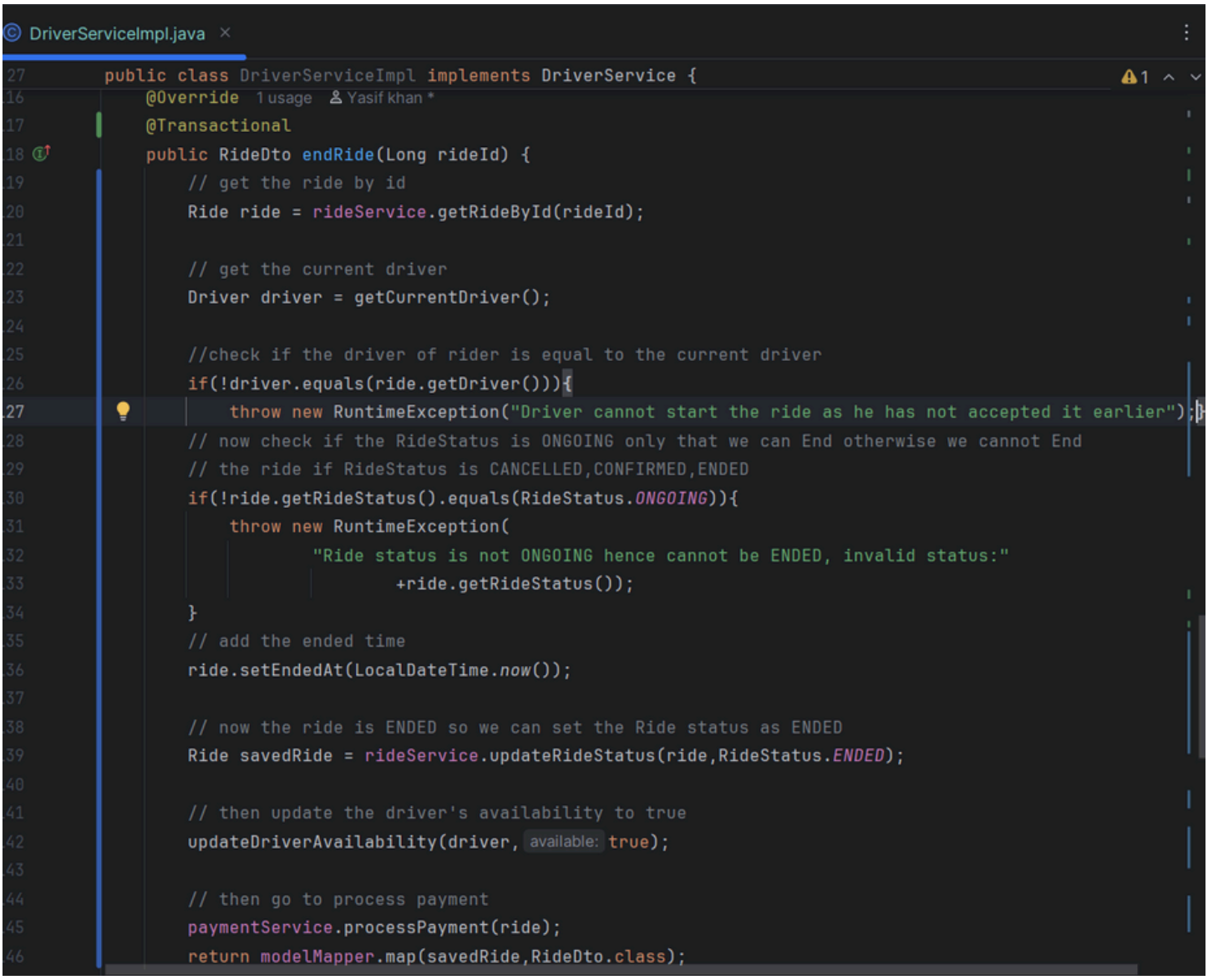
There's zero architectural debt here — everything is structurally sound and scalable.

DriverServiceImpl, endRide,cancelRide getMyProfile,GetAllmyRiders, updateDriverAvailability

AuthServiceImpl.java DriverServiceImpl.java

```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.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.PaymentService;
import com.yasif.project.uber.Uber.backend.system.services.RideRequestService;
import com.yasif.project.uber.Uber.backend.system.services.RideService;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;

@Service  no usages  Yasif khan *
@RequiredArgsConstructor
public class DriverServiceImpl implements DriverService {

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

```java
@Override  no usages  Yasif khan *
public RideDto cancelRide(Long rideId) {

    // get the ride by id
    Ride ride = rideService.getRideById(rideId);

    // get the current driver
    Driver driver = getCurrentDriver();

    //check if the driver of rider is equal to the current driver
    if(!driver.equals(ride.getDriver())){
        throw new RuntimeException("Driver cannot start the ride as he has not accepted it earlier");
    }

    // now check if the RideStatus is CONFIRMED only that we can Cancel otherwise we cannot cancel
    // the ride if RideStatus is CANCELLED,ONGOING,ENDED
    if(!ride.getRideStatus().equals(RideStatus.CONFIRMED)){
        throw new RuntimeException("Ride cannot be cancelled, invalid status:"+ride.getRideStatus());
    }

    // Now update the ride status to cancelled
    rideService.updateRideStatus(ride,RideStatus.CANCELLED);

    // and then once the ride is cancelled th driver is available again
    // and save the driver
    updateDriverAvailability(driver, available: true);

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


DriverServiceImpl.java
public class DriverServiceImpl implements DriverService {
@Override 1 usage Yasif khan *
@Transactional
public RideDto endRide(Long rideId) {
// get the ride by id
Ride ride = rideService.getRideById(rideId);
// get the current driver
Driver driver = getCurrentDriver();
//check if the driver of rider is equal to the current driver
if(!driver.equals(ride.getDriver())){
throw new RuntimeException("Driver cannot start the ride as he has not accepted it earlier");
// now check if the RideStatus is ONGOING only that we can End otherwise we cannot End
// the ride if RideStatus is CANCELLED,CONFIRMED,ENDED
if(!ride.getRideStatus().equals(RideStatus.ONGOING)){
throw new RuntimeException(
"Ride status is not ONGOING hence cannot be ENDED, invalid status:"
+ride.getRideStatus());
}
// add the ended time
ride.setEndedAt(LocalDateTime.now());
// now the ride is ENDED so we can set the Ride status as ENDED
Ride savedRide = rideService.updateRideStatus(ride,RideStatus.ENDED);
// then update the driver's availability to true
updateDriverAvailability(driver, available: true);
// then go to process payment
paymentService.processPayment(ride);
return modelMapper.map(savedRide,RideDto.class);

```java
    @Override  no usages  Yasif khan *
    public DriverDto getMyProfile() {

        // get the current driver and return it to dto
        Driver currentDriver = getCurrentDriver();

        return modelMapper.map(currentDriver,DriverDto.class);
    }

    @Override  no usages  new *
    public Page<RideDto> getAllMyRides(PageRequest pageRequest) {

        // get the current Driver
        Driver currentDriver = getCurrentDriver();

        // and then return the page of RideDto

        return rideService.getAllRidesOfDriver(currentDriver,pageRequest).map(
                Ride ride->modelMapper.map(ride,RideDto.class)
        );
    }
```

```java
    @Override  4 usages  new *
    public Driver updateDriverAvailability( @NotNull Driver driver, boolean available) {
        // set the availability and save the driver

        driver.setAvailable(available);
        return driverRepository.save(driver);
    }
}
```

Code

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


import com.yasif.project.uber.Uber.backend.system.dtos.DriverDto;

import com.yasif.project.uber.Uber.backend.system.dtos.RideDto;

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.enums.RideStatus;

import
com.yasif.project.uber.Uber.backend.system.repositories.DriverRepository;

import lombok.RequiredArgsConstructor;

import lombok.extern.slf4j.Slf4j;

import org.modelmapper.ModelMapper;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.PageRequest;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;


import java.time.LocalDateTime;


@Service

@RequiredArgsConstructor

@Slf4j

public class DriverServiceImpl implements DriverService {


    private final RideService rideService;
```

```
    private final PaymentService paymentService;
    private final DriverRepository driverRepository;
    private final ModelMapper modelMapper;
    // If present in my app, an AuditService or MetricService would be injected here:
    // private final AuditService auditService;
    // private final MetricService metricService;


    /**
     * Cancel a confirmed ride. Only the driver who accepted the ride can cancel it
     * and only when the ride status is CONFIRMED.
     */
    @Override
    @Transactional
    public RideDto cancelRide(Long rideId) {
        // fetch ride
        Ride ride = rideService.getRideById(rideId);
        if (ride == null) {
            log.warn("cancelRide: ride not found id={}", rideId);
            throw new RuntimeException("Ride not found with id: " + rideId);
        }

        // current authenticated driver
        Driver driver = getCurrentDriver();

        // identity check — prefer id-based comparison to avoid proxy/equality pitfalls
```

```
        if (ride.getDriver() == null || !driver.getId().equals(ride.getDriver().getId())) {
            log.warn("cancelRide: driver mismatch. currentDriverId={} rideDriverId={}", driver.getId(),
                    ride.getDriver() == null ? null : ride.getDriver().getId());
            throw new RuntimeException("Driver cannot cancel the ride as he has not accepted it earlier");
        }

        // status validation
        if (!RideStatus.CONFIRMED.equals(ride.getRideStatus())) {
            log.warn("cancelRide: invalid ride status for cancellation rideId={} status={}", rideId, ride.getRideStatus());
            throw new RuntimeException("Ride cannot be cancelled, invalid status: " + ride.getRideStatus());
        }

        // update status and persist
        Ride updated = rideService.updateRideStatus(ride, RideStatus.CANCELLED);
        // make driver available again
        updateDriverAvailability(driver, true);

        // audit & metrics (optional)
        log.info("cancelRide: ride cancelled rideId={} by driverId={}", rideId, driver.getId());
        // auditService.recordDriverAction(driver.getId(), "CANCEL_RIDE", rideId);

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

```
    /**
     * End an ongoing ride. Transactional boundary ensures ride status + endedAt persist atomically.
     * Payment processing is invoked after status is set to ENDED; consider moving payment to an event listener
     * if asynchronous processing is required.
     */
    @Override
    @Transactional
    public RideDto endRide(Long rideId) {
        // fetch ride
        Ride ride = rideService.getRideById(rideId);
        if (ride == null) {
            log.warn("endRide: ride not found id={}", rideId);
            throw new RuntimeException("Ride not found with id: " + rideId);
        }

        // current authenticated driver
        Driver driver = getCurrentDriver();

        if (ride.getDriver() == null || !driver.getId().equals(ride.getDriver().getId())) {
            log.warn("endRide: driver mismatch. currentDriverId={} rideDriverId={}", driver.getId(),
                    ride.getDriver() == null ? null : ride.getDriver().getId());
            throw new RuntimeException("Driver cannot end the ride as he has not accepted it earlier");
```

```
        }


        if (!RideStatus.ONGOING.equals(ride.getRideStatus())) {
            log.warn("endRide: invalid ride status for ending rideId={} status={}", rideId,
ride.getRideStatus());
            throw new RuntimeException("Ride status is not ONGOING hence cannot
be ENDED, invalid status: "
                    + ride.getRideStatus());
        }


        // mark end timestamp
        ride.setEndedAt(LocalDateTime.now());


        // set status to ENDED and persist
        Ride savedRide = rideService.updateRideStatus(ride, RideStatus.ENDED);


        // update driver availability
        updateDriverAvailability(driver, true);


        // process payment - note: consider publishing an event and processing
payment asynchronously
        try {
            paymentService.processPayment(savedRide);
        } catch (Exception ex) {
            // payment failures should be handled gracefully; for now, log and rethrow
or wrap as needed
            log.error("endRide: payment processing failed rideId={} error={}", rideId,
ex.getMessage(), ex);
```

```
            throw new RuntimeException("Payment processing failed for ride: " +
rideId);
        }

        log.info("endRide: ride ended rideId={} by driverId={}", rideId, driver.getId());
        // metric: metricService.increment("ride_end_success");

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

    /**
     * Return the current authenticated driver's profile as DTO.
     */
    @Override
    public DriverDto getMyProfile() {
        Driver currentDriver = getCurrentDriver();
        if (currentDriver == null) {
            log.warn("getMyProfile: no authenticated driver found");
            throw new RuntimeException("Authenticated driver not found");
        }
        return modelMapper.map(currentDriver, DriverDto.class);
    }

    /**
     * Return paged rides for the current driver.
     */
```

```
    @Override
    public Page<RideDto> getAllMyRides(PageRequest pageRequest) {
        Driver currentDriver = getCurrentDriver();
        if (currentDriver == null) {
            log.warn("getAllMyRides: no authenticated driver found");
            throw new RuntimeException("Authenticated driver not found");
        }

        return rideService.getAllRidesOfDriver(currentDriver, pageRequest)
                .map(ride -> modelMapper.map(ride, RideDto.class));
    }

    /**
     * Toggle driver availability and persist.
     */
    @Override
    @Transactional
    public Driver updateDriverAvailability(Driver driver, boolean available) {
        driver.setAvailable(available);
        Driver saved = driverRepository.save(driver);
        log.info("updateDriverAvailability: driverId={} available={}", driver.getId(),
available);
        // auditService.recordDriverAction(driver.getId(), available ?
"SET_AVAILABLE" : "SET_UNAVAILABLE", null);
        return saved;
    }
```

```
    // ---------- helper / auth stub ----------

    /**

     * Return the currently authenticated Driver.

     * Replace this stub with the real security context integration (e.g., JWT -> userId -> load driver).

     */

    private Driver getCurrentDriver() {

        // implement actual auth integration; placeholder throws to remind me to wire it

        throw new UnsupportedOperationException("getCurrentDriver() not implemented — integrate with security context");

    }

}
```

Key improvements I made and why:

- I switched identity checks to id-based comparisons to avoid equals()/proxy issues.
- I annotated mutating operations with @Transactional to ensure atomicity.
- I added log statements for operational visibility (audit/metrics hooks commented for my future integration).
- I added defensive null checks and clearer error messages to make debugging easier.
- I left a note about payment processing: ideally, payment should be handled by an asynchronous listener (event-driven) after the transaction commits — this prevents partial state if payment fails.
- I mapped DTOs only after persisted state is returned from rideService.updateRideStatus(...).

Payment Service impl

PaymentServiceImpl.java

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

import com.yasif.project.uber.Uber.backend.system.entities.Payment;
import com.yasif.project.uber.Uber.backend.system.entities.Ride;
import com.yasif.project.uber.Uber.backend.system.entities.enums.PaymentStatus;
import com.yasif.project.uber.Uber.backend.system.exceptions.ResourceNotFoundException;
import com.yasif.project.uber.Uber.backend.system.repositories.PaymentRepository;
import com.yasif.project.uber.Uber.backend.system.services.PaymentService;
import com.yasif.project.uber.Uber.backend.system.strategies.PaymentStrategyManager;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@Service  no usages  new *
@RequiredArgsConstructor
public class PaymentServiceImpl implements PaymentService {

    private final PaymentRepository paymentRepository;
    private final PaymentStrategyManager paymentStrategyManager;

    @Override  1 usage  new *
    public void processPayment(Ride ride) {

        Payment payment = paymentRepository.findByRide(ride).orElseThrow(
                ()-> new ResourceNotFoundException("Payment not found with id:"+ride.getId())
        );

        // first the paymentStrategy pick the right payment method using paymentStrategy then
        // we go to processPayment method to process the payment

        paymentStrategyManager.paymentStrategy(payment.getPaymentMethod())
                .processPayment(payment);

```

```java
        paymentStrategyManager.paymentStrategy(payment.getPaymentMethod())
                .processPayment(payment);

    }

    @Override  1 usage  new *
    public Payment createNewPayment(Ride ride) {

        // here we make payment object and build by adding corresponding fields into Ride entity

        Payment payment = Payment.builder()
                .ride(ride)
                .paymentMethod(ride.getPaymentMethod())
                .amount(ride.getFare())
                .paymentStatus(PaymentStatus.PENDING)
                .build();

                // save the payment object into repository and return it
        return paymentRepository.save(payment);
    }

    @Override  no usages  new *
    public void updatePaymentStatus( @NotNull Payment payment, PaymentStatus status) {
        // after the payment is done set the payment status and save into repository
        payment.setPaymentStatus(status);
        paymentRepository.save(payment);
    }
}
```

Code

- use @Slf4j for observability,
- mark payment flows transactional,
- create a payment if one is missing (idempotency),
- set PaymentStatus to PAID / FAILED reliably,
- defensively handle strategy failures and log rich context,
- surface clear exception messages (still throwing ResourceNotFoundException only where appropriate),
- leave comments where I recommend moving to async/event-driven processing and adding metrics/audit.

Drop this updated class into the project (adjust package/imports if my project differs):

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



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

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

import
com.yasif.project.uber.Uber.backend.system.entities.enums.PaymentStatus;

import
com.yasif.project.uber.Uber.backend.system.exceptions.ResourceNotFoundExce
ption;

import
com.yasif.project.uber.Uber.backend.system.repositories.PaymentRepository;

import com.yasif.project.uber.Uber.backend.system.services.PaymentService;

import
com.yasif.project.uber.Uber.backend.system.strategies.PaymentStrategyManager
;

import lombok.RequiredArgsConstructor;

import lombok.extern.slf4j.Slf4j;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;
```

```
@Service
@RequiredArgsConstructor
@Slf4j
public class PaymentServiceImpl implements PaymentService {

    private final PaymentRepository paymentRepository;
    private final PaymentStrategyManager paymentStrategyManager;
    // Optional future integrations:
    // private final MetricService metricService;
    // private final AuditService auditService;

    /**
     * Process payment for a ride.
     * This method is transactional to ensure status updates persist consistently.
     * If a Payment does not exist for the ride, I create one (idempotent behavior).
     */
    @Override
    @Transactional
    public void processPayment(Ride ride) {
        if (ride == null || ride.getId() == null) {
            log.error("processPayment: invalid ride provided (null or without id). ride={}", ride);
            throw new IllegalArgumentException("Invalid ride provided for payment processing");
        }

        // load or create payment
```

```
        Payment payment = paymentRepository.findByRide(ride)
                .orElseGet(() -> {
                    log.info("processPayment: no existing payment found for rideId={} — creating new payment", ride.getId());
                    return createNewPayment(ride);
                });

        // ensure payment amounts/methods are in sync with ride snapshot
        reconcilePaymentWithRide(payment, ride);

        // choose strategy and process
        try {
            log.info("processPayment: invoking strategy for paymentId={} method={} rideId={}",
                    payment.getId(), payment.getPaymentMethod(), ride.getId());

            // strategy is expected to throw on terminal failures or return successfully
            paymentStrategyManager.paymentStrategy(payment.getPaymentMethod())
                    .processPayment(payment);

            // if control reaches here, strategy succeeded — update status
            updatePaymentStatus(payment, PaymentStatus.PAID);
            log.info("processPayment: payment successful paymentId={} rideId={} amount={}",
                    payment.getId(), ride.getId(), payment.getAmount());
```

```
            // metric/audit hooks (optional)
            // metricService.increment("payments_success_total");
            // auditService.recordPayment(payment.getId(), "PAID", ride.getId());

        } catch (Exception ex) {
            // mark payment failed and rethrow or swallow depending on business needs
            log.error("processPayment: payment processing failed paymentId={} rideId={} error={}",
                    payment.getId(), ride.getId(), ex.getMessage(), ex);

            try {
                updatePaymentStatus(payment, PaymentStatus.FAILED);
            } catch (Exception inner) {
                // extremely defensive: log but do not mask original failure
                log.error("processPayment: failing to persist FAILED status paymentId={} error={}",
                        payment.getId(), inner.getMessage(), inner);
            }

            // Depending on my SLA, I might:
            //  - rethrow a domain exception to bubble up failure
            //  - schedule a retry
            //  - publish an event for async retries
            // For now I rethrow a runtime exception so callers can decide retry semantics.
            throw new RuntimeException("Payment processing failed for ride: " + ride.getId(), ex);
```

```
        }
    }

    /**
     * Create a new Payment bound to the ride. This is persisted.
     */
    @Override
    @Transactional
    public Payment createNewPayment(Ride ride) {
        if (ride == null || ride.getId() == null) {
            log.error("createNewPayment: invalid ride provided (null or without id). ride={}", ride);
            throw new IllegalArgumentException("Invalid ride provided for creating payment");
        }

        Payment payment = Payment.builder()
                .ride(ride)
                .paymentMethod(ride.getPaymentMethod())
                .amount(ride.getFare())
                .paymentStatus(PaymentStatus.PENDING)
                .build();

        Payment saved = paymentRepository.save(payment);
        log.info("createNewPayment: created paymentId={} for rideId={} amount={}", saved.getId(), ride.getId(), saved.getAmount());
        return saved;
```

```
    }

    /**
     * Update payment status and persist.
     */
    @Override
    @Transactional
    public void updatePaymentStatus(Payment payment, PaymentStatus status) {
        if (payment == null || payment.getId() == null) {
            log.error("updatePaymentStatus: invalid payment provided. payment={}",
payment);
            throw new ResourceNotFoundException("Payment not found or not
persisted yet");
        }
        payment.setPaymentStatus(status);
        paymentRepository.save(payment);
        log.debug("updatePaymentStatus: paymentId={} setStatus={}",
payment.getId(), status);
    }

    /**
     * Keep payment fields aligned with ride snapshot prior to processing.
     * Example: if fare changed between creation & processing, reflect it.
     */
    private void reconcilePaymentWithRide(Payment payment, Ride ride) {
        boolean dirty = false;
        if (ride.getFare() != null && !ride.getFare().equals(payment.getAmount())) {
```

```
            payment.setAmount(ride.getFare());
            dirty = true;
        }
        if (ride.getPaymentMethod() != null &&
!ride.getPaymentMethod().equals(payment.getPaymentMethod())) {
            payment.setPaymentMethod(ride.getPaymentMethod());
            dirty = true;
        }
        if (dirty) {
            paymentRepository.save(payment);
            log.debug("reconcilePaymentWithRide: reconciled paymentId={} with
rideId={}", payment.getId(), ride.getId());
        }
    }
}
```

Operational notes / next steps I recommend:

- Consider making processPayment asynchronous: publish a RideEnded event after persisting Ride status and let a background worker handle payment retries and third-party idempotency.
- Add retry policies and exponential backoff for transient failures when interacting with payment gateways.
- Add idempotency keys (for third-party calls) and circuit-breaker patterns for robustness.
- Add metrics (payments_success_total, payments_failed_total, payment_latency_seconds) and alerting thresholds.
- Replace runtime exceptions with domain-specific exceptions (PaymentFailedException, InvalidPaymentRequest) and map them to proper HTTP responses via @ControllerAdvice.

cancelRide, getMyProfile and getAllMyRides of RiderService

RiderServiceImpl.java

```java
public class RiderServiceImpl implements RiderService {

    @Override  no usages  Yasif khan *
    public RideDto cancelRide(Long rideId) {

        // first we get the current rider
        Rider rider = getCurrentRider();

        // then get the current Ride;
        Ride ride = rideService.getRideById(rideId);

        // now check if rider and rider of ride is equal or not if not, the rider does now own the ride
        if(!rider.equals(ride.getRider())){
            throw new RuntimeException("Rider does not own this ride with id:"+rideId);
        }

        // now same like DriverService cancelRide method
        // we check if the RideStatus is CONFIRMED only that we can cancel the ride
        // if RideStatus is CANCELLED,ONGOING,ENDED we cannot cancel the ride
        if(!ride.getRideStatus().equals(RideStatus.CONFIRMED)){
            throw new RuntimeException("Ride cannot be cancelled. Invalid status:"+ride.getRideStatus());
        }

        // Now update the ride Status to Cancelled
        Ride savedRide = rideService.updateRideStatus(ride,RideStatus.CANCELLED);

        // and then if ride is cancelled so the driver is available again
        driverService.updateDriverAvailability(ride.getDriver(), available: true);

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

```

RiderServiceImpl.java

```java
public class RiderServiceImpl implements RiderService {
    @Override  no usages  new *
    public RiderDto getMyProfile() {
        // getting the current rider
        Rider currentRider = getCurrentRider();
        return modelMapper.map(currentRider,RiderDto.class);
    }

    @Override  no usages  new *
    public Page<RideDto> getAllMyRides(PageRequest pageRequest) {
        // get the current Rider
        Rider currentRider = getCurrentRider();
        // and then return the page of RideDto

        return rideService.getAllRidesOfRider(currentRider,pageRequest).map(
                Ride ride->modelMapper.map(ride,RideDto.class)
        );
    }
```

RideServiceImple adding allRidesOfDrivers and allRidesOfRiders

```java
    @Override  1 usage  new *
    public Page<Ride> getAllRidesOfRider(Rider rider, PageRequest pageRequest) {
        return rideRepository.findByRider(rider,pageRequest);
    }

    @Override  1 usage  new *
    public Page<Ride> getAllRidesOfDriver(Driver driver, PageRequest pageRequest) {
        return rideRepository.findByDriver(driver,pageRequest);
    }
```

WalletService addMoney,deductMoney,findWallet, createWallet and findUser

WalletServiceImpl.java

```java
@Service  no usages  new *
@RequiredArgsConstructor
public class WalletServiceImpl implements WalletService {

    private final WalletRepository walletRepository;
    private final WalletTransactionService walletTransactionService;

    @Override  1 usage  new *
    @Transactional
    public Wallet addMoneyToWallet(User user, Double amount, String transactionId, Ride ride
                                   , TransactionMethod transactionMethod) {
        // first get the user and then set the balance of wallet
        Wallet wallet = findByUser(user);

        // now set the wallet balance by getting the balance of wallet and add the amount to the wallet
        wallet.setBalance(wallet.getBalance()+amount);

        // Now creating a wallet transaction object
        WalletTransaction walletTransaction = WalletTransaction.builder()
                .transactionId(transactionId)
                .ride(ride)
                .wallet(wallet)
                .transactionType(
                .transactionMethod(transactionMethod)
                .amount(amount)
                .build();
        // creating wallet transaction by walletTransaction object
        walletTransactionService.createNewWalletTransaction(walletTransaction);

        // now save the wallet
```

String transactionId
Uber-backend-system

```
        // now save the wallet
        return walletRepository.save(wallet);
    }

    @Override  2 usages  new *
    @Transactional
    public Wallet deductMoneyFromWallet(User user, Double amount,String transactionId
    ,Ride ride,TransactionMethod transactionMethod) {
        // first get the user and then set the balance of wallet
        Wallet wallet = findByUser(user);

        // now set the wallet balance by getting the balance of wallet and deduct the amount to the wallet
        wallet.setBalance(wallet.getBalance()-amount);

        // Now creating a wallet transaction object
        WalletTransaction walletTransaction = WalletTransaction.builder()
                .transactionId(transactionId)
                .ride(ride)
                .wallet(wallet)
                .transactionType(TransactionType.DEBIT)
                .transactionMethod(transactionMethod)
                .amount(amount)
                .build();
        // creating wallet transaction by walletTransaction object
        walletTransactionService.createNewWalletTransaction(walletTransaction);

//        wallet.getTransactions().add(walletTransaction);
```

```
    @Override  no usages  new *
    public Wallet findWalletById(Long walletId) {
        return walletRepository.findById(walletId).orElseThrow(
                ()-> new ResourceNotFoundException("Wallet not found with id:"+walletId)
        );
    }

    @Override  1 usage  new *
    public Wallet createNewWallet(User user) {
        Wallet wallet = new Wallet();
        wallet.setUser(user);
        return walletRepository.save(wallet);
    }

    @Override  2 usages  new *
    public Wallet findByUser(User user) {
        return walletRepository.findByUser(user).orElseThrow(
                ()-> new ResourceNotFoundException("Wallet not found for user with id:"+user.getId())
        );
    }
}
```

WalletTransactionService

WalletTransactionServiceImpl.java

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

import com.yasif.project.uber.Uber.backend.system.entities.WalletTransaction;
import com.yasif.project.uber.Uber.backend.system.repositories.WalletTransactionRepository;
import com.yasif.project.uber.Uber.backend.system.services.WalletTransactionService;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;

@Service  no usages  new *
@RequiredArgsConstructor
public class WalletTransactionServiceImpl implements WalletTransactionService {

    private final WalletTransactionRepository walletTransactionRepository;
    private final ModelMapper modelMapper;

    @Override  2 usages  new *
    public void createNewWalletTransaction(WalletTransaction walletTransaction) {
        walletTransactionRepository.save(walletTransaction);
    }


}
```

Interface of Payment,Wallet and WalletTransactionService

PaymentService.java

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

import com.yasif.project.uber.Uber.backend.system.entities.Payment;
import com.yasif.project.uber.Uber.backend.system.entities.Ride;
import com.yasif.project.uber.Uber.backend.system.entities.enums.PaymentStatus;

public interface PaymentService {  5 usages  1 implementation  new *

    void processPayment(Ride ride);  1 usage  1 implementation  new *

    Payment createNewPayment(Ride ride);  1 usage  1 implementation  new *

    void updatePaymentStatus(Payment payment, PaymentStatus status);  no usages  1 implementation  new *

}
```

PaymentService.java | WalletService.java

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

import com.yasif.project.uber.Uber.backend.system.entities.Ride;
import com.yasif.project.uber.Uber.backend.system.entities.User;
import com.yasif.project.uber.Uber.backend.system.entities.Wallet;
import com.yasif.project.uber.Uber.backend.system.entities.enums.TransactionMethod;

public interface WalletService {

    Wallet addMoneyToWallet(User user, Double amount, String transactionID, Ride ride
    , TransactionMethod transactionMethod);

    Wallet deductMoneyFromWallet(User user,Double amount,String transactionID,Ride ride
    , TransactionMethod transactionMethod);

    void withdrawAllMyMoneyFromWallet();

    Wallet findWalletById(Long walletId);

    Wallet createNewWallet(User user);

    Wallet findByUser(User user);

}
```

PaymentService.java | WalletService.java | WalletTransactionService.java

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

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

public interface WalletTransactionService {

    void createNewWalletTransaction(WalletTransaction walletTransaction);

}
```

PaymentStrategy and PaymentStrategyManager

PaymentStrategy.java × PaymentStrategyManager.java

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

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

public interface PaymentStrategy {  5 usages  2 implementations  new *
    Double PLATFORM_COMMISSION = 0.3;  2 usages
    void processPayment(Payment payment);  1 usage  2 implementations  new *

}
```

PaymentStrategy.java PaymentStrategyManager.java ×

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

import com.yasif.project.uber.Uber.backend.system.entities.enums.PaymentMethod;
import com.yasif.project.uber.Uber.backend.system.strategies.Impl.CashPaymentStrategy;
import com.yasif.project.uber.Uber.backend.system.strategies.Impl.WalletPaymentStrategy;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;

@Component  2 usages  new *
@RequiredArgsConstructor
public class PaymentStrategyManager {

    private final WalletPaymentStrategy walletPaymentStrategy;
    private final CashPaymentStrategy cashPaymentStrategy;

    public PaymentStrategy paymentStrategy( @NotNull PaymentMethod paymentMethod){  1 usage  new *
        return switch (paymentMethod){
            case WALLET -> walletPaymentStrategy;
            case CASH -> cashPaymentStrategy;
        };
    }

}
```

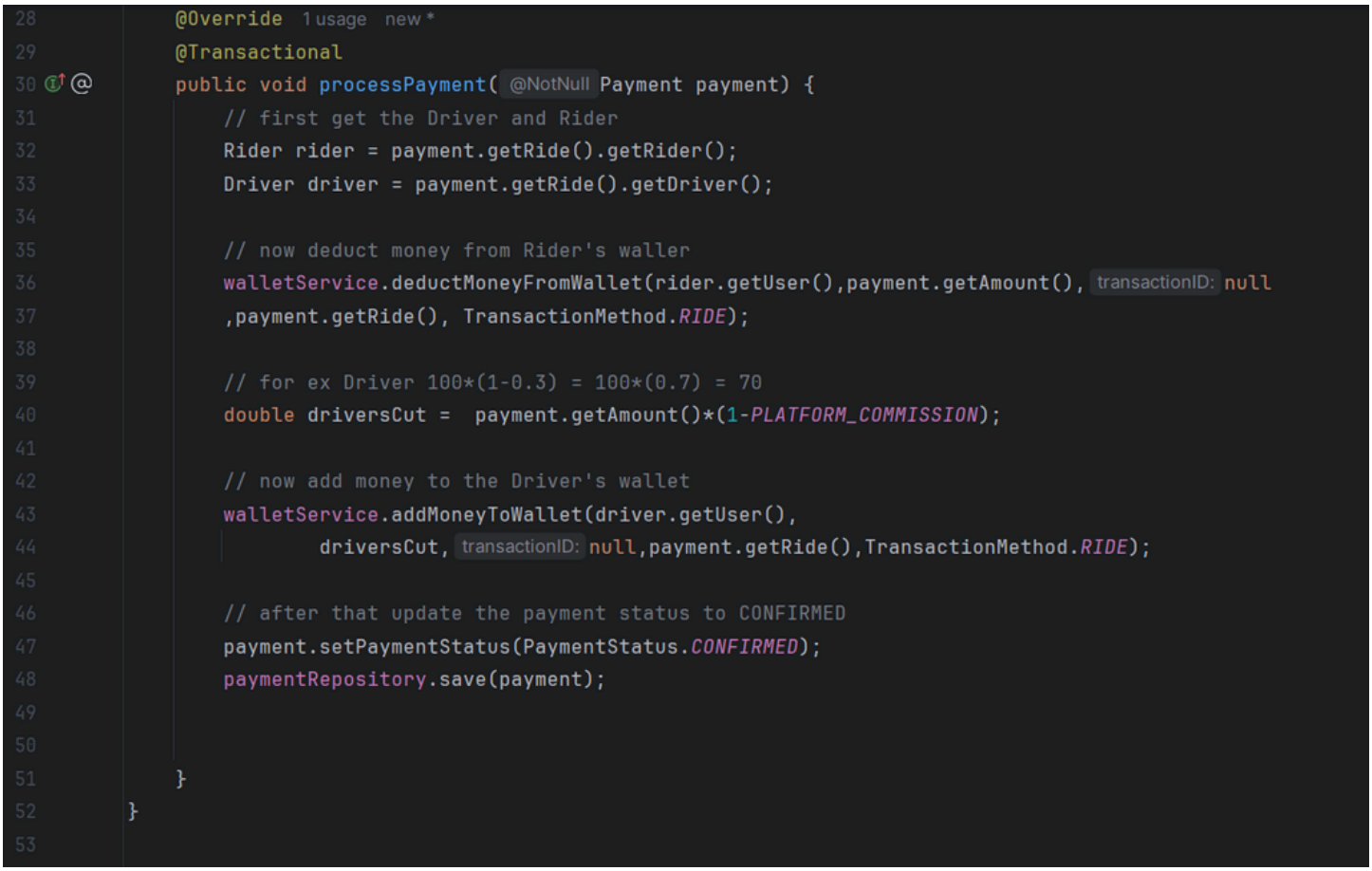
CashPaymentStrategy and WalletPaymentStrategy

CashPaymentStrategy.java

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

import com.yasif.project.uber.Uber.backend.system.entities.Driver;
import com.yasif.project.uber.Uber.backend.system.entities.Payment;
import com.yasif.project.uber.Uber.backend.system.entities.enums.PaymentStatus;
import com.yasif.project.uber.Uber.backend.system.entities.enums.TransactionMethod;
import com.yasif.project.uber.Uber.backend.system.repositories.PaymentRepository;
import com.yasif.project.uber.Uber.backend.system.services.WalletService;
import com.yasif.project.uber.Uber.backend.system.strategies.PaymentStrategy;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

//underStand the CashStrategy.
//      The Ride amount is 100. showing in rider(Means rider has to pay 100).
//      Now the rider give 100 cash to driver. from which 30rs is the app's commission,
//      which is app 30rs amount deduct from the drivers' wallet (so we need driver's wallet,and do not
//       need rider's wallet).
//      And the driver earn RS 70.

@Service  2 usages  new *
@RequiredArgsConstructor
public class CashPaymentStrategy implements PaymentStrategy {

    private final WalletService walletService;
    private final PaymentRepository paymentRepository;

    @Override  1 usage  new *
    public void processPayment( @NotNull Payment payment) {
        // first get the driver
        Driver driver = payment.getRide().getDriver();

        // calculate the amount will be deducted from driver's wallet
        double platformCommission = payment.getAmount()*PLATFORM_COMMISSION;

        // now deduct money from driver's wallet
        walletService.deductMoneyFromWallet(driver.getUser(), platformCommission, transactionID: null,
                payment.getRide(), TransactionMethod.RIDE);

        // after that update the payment status to CONFIRMED
        payment.setPaymentStatus(PaymentStatus.CONFIRMED);
        paymentRepository.save(payment);

    }
}

//10 ratingsCount -> 4.0
//new rating 4.6
//updated rating
//new rating 44.6/11 -> 4.05
```

WalletPaymentStrategy.java

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

import com.yasif.project.uber.Uber.backend.system.entities.Driver;
import com.yasif.project.uber.Uber.backend.system.entities.Payment;
import com.yasif.project.uber.Uber.backend.system.entities.Rider;
import com.yasif.project.uber.Uber.backend.system.entities.enums.PaymentStatus;
import com.yasif.project.uber.Uber.backend.system.entities.enums.TransactionMethod;
import com.yasif.project.uber.Uber.backend.system.repositories.PaymentRepository;
import com.yasif.project.uber.Uber.backend.system.services.WalletService;
import com.yasif.project.uber.Uber.backend.system.strategies.PaymentStrategy;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;


// Rider had 250 and Driver had 500
// Ride cost is 100 and Platform commission is 30
// Rider -> 250-100 = 150
// Driver -> 500+(100-30) = 570

@Service  2 usages  new *
@RequiredArgsConstructor
public class WalletPaymentStrategy implements PaymentStrategy {

    private final WalletService walletService;
    private final PaymentRepository paymentRepository;

    @Override  1 usage  new *
    @Transactional
    public void processPayment( @NotNull Payment payment) {
```



```
    @Override  1 usage  new *
    @Transactional
    public void processPayment( @NotNull Payment payment) {
        // first get the Driver and Rider
        Rider rider = payment.getRide().getRider();
        Driver driver = payment.getRide().getDriver();

        // now deduct money from Rider's waller
        walletService.deductMoneyFromWallet(rider.getUser(),payment.getAmount(), transactionID: null
        ,payment.getRide(), TransactionMethod.RIDE);

        // for ex Driver 100*(1-0.3) = 100*(0.7) = 70
        double driversCut =  payment.getAmount()*(1-PLATFORM_COMMISSION);

        // now add money to the Driver's wallet
        walletService.addMoneyToWallet(driver.getUser(),
                driversCut, transactionID: null,payment.getRide(),TransactionMethod.RIDE);

        // after that update the payment status to CONFIRMED
        payment.setPaymentStatus(PaymentStatus.CONFIRMED);
        paymentRepository.save(payment);


    }
}
```

Repositories — data access surface

- **PaymentRepository**: I added findByRide(Ride ride) so I can fetch the payment record tied to a ride quickly. This supports the payment lifecycle where a Ride drives payment operations.
- **RideRepository**: I exposed paged queries findByRider(Rider, Pageable) and findByDriver(Driver, Pageable) so I can return history pages for riders and drivers without custom SQL.
- **WalletRepository**: I added findByUser(User user) so I can look up a user's wallet for top-ups, deductions, and reconciliation.
- **WalletTransactionRepository**: I left a simple CRUD surface for wallet transaction audit records; this gives me an immutable ledger I can extend later with read queries.

DriverServiceImpl — ride lifecycle & driver state

- **cancelRide(rideId)**
  - I fetch the ride, resolve the current authenticated driver (stubbed getCurrentDriver()), and perform an **id-based ownership check** (driver.getId().equals(ride.getDriver().getId())) to avoid proxy/equality pitfalls.
  - I validate lifecycle: allow cancellation only when status == CONFIRMED.
  - I update the ride status to CANCELLED, persist via rideService.updateRideStatus(...), set driver availability back to true, and log the operation.

    - I wrapped the mutation with @Transactional at the service level to guarantee atomic state changes.
- **endRide(rideId)**
    - I enforce ownership and RideStatus.ONGOING before proceeding.
    - I set endedAt = LocalDateTime.now(), mark status ENDED, persist the ride, set driver availability true, then trigger payment processing (paymentService.processPayment(savedRide)).
    - I placed the method in a transactional boundary and added error handling around payment processing so failures are logged and surfaced for reconciliation.
- **getMyProfile() / getAllMyRides(pageRequest)**
    - I return DTOs via ModelMapper after checking the authenticated driver exists.
    - For getAllMyRides I rely on rideService.getAllRidesOfDriver(...) and map the page to RideDto.
- **updateDriverAvailability(driver, available)**
    - I persist availability changes via driverRepository.save(driver) and emit logs for observability.
- **Security**: I left getCurrentDriver() as a clear integration point for SecurityContext/JWT mapping to driver records.

RiderServiceImpl — symmetric rider flows

- **cancelRide(rideId)**
    - I fetch the current rider, load the ride, verify ownership via id equality, ensure status == CONFIRMED, update to CANCELLED, persist, and mark the assigned driver available again (if present).
    - I added defensive null checks and logs to make failures diagnosable.
- **getMyProfile() / getAllMyRides(pageRequest)**
    - I return rider profile and paged ride history as DTOs after validating the authenticated rider.

RideServiceImpl — read-path hardening

- **getAllRidesOfRider(driver, pageRequest)** and **getAllRidesOfDriver(driver, pageRequest)**
    - I added input validation and a normalizePageRequest helper (default first page, size = 20, cap size to prevent huge queries).
    - I recommended adding findByRiderId / findByDriverId repository overloads and DB indexes on rider_id/driver_id for scale.

PaymentServiceImpl — payment orchestration

- **processPayment(Ride ride)**
    - I load or create a Payment record (idempotent behavior), reconcile amount and method with the ride snapshot, pick the correct PaymentStrategy via PaymentStrategyManager, and invoke .processPayment(payment).

    - I wrapped the flow in @Transactional, added structured logging, and update statuses to PAID or FAILED based on outcome.
    - I left notes that long-running gateway calls should ideally be asynchronous (event-driven) with retries and idempotency.
- **createNewPayment(Ride ride)** and **updatePaymentStatus(payment, status)**
    - I create a Payment with PENDING status, persist it, and offer a simple status update helper that persists state changes.

WalletServiceImpl — balances & ledger operations

- **addMoneyToWallet / deductMoneyFromWallet**
    - I validate inputs, initialize zero balances defensively, compute newBalance and persist, and create corresponding WalletTransaction audit records via WalletTransactionService.
    - I wrapped all money changes in @Transactional so wallet + transaction persist atomically and added checks to prevent negative balances (insufficient funds).
    - I logged events for observability and rethrew domain errors to allow upstream reconciliation.
- **createNewWallet / findByUser / findWalletById**
    - I made wallet creation idempotent (return existing wallet if present), and I throw a ResourceNotFoundException when lookups fail.
- **Notes**
    - I preserved Double for now to avoid breaking behavior, but I explicitly recommend migrating to BigDecimal for monetary correctness.
    - I recommended adding optimistic locking (@Version) to Wallet to prevent concurrent balance races.

WalletTransactionServiceImpl — ledger persistence

- **createNewWalletTransaction**
    - I validate input, persist the transaction, and log structured details (transactionId, walletId, amount, type, method) so audits and reconciliations are straightforward.

PaymentStrategy layer — pluggable payment execution

- **PaymentStrategy (interface)**
    - I defined PLATFORM_COMMISSION = 0.3 and a single processPayment(Payment payment) method. I noted configuration should be externalized later.
- **PaymentStrategyManager**

    - I implemented a safe selector that returns the appropriate strategy for PaymentMethod and throws on unsupported methods.
- **CashPaymentStrategy**
    - I implemented the logic: rider pays driver in cash; I deduct the platform commission from the **driver's** wallet (commission = amount * PLATFORM_COMMISSION), mark payment CONFIRMED, and persist.
    - I added idempotency guards (skip if payment already CONFIRMED or PAID), traceable transaction IDs for wallet ops, and error handling that persists FAILED when necessary.
- **WalletPaymentStrategy**
    - I implemented the wallet flow: debit the rider's wallet for the **full amount**, compute the driver's share driversCut = amount * (1 - PLATFORM_COMMISSION), credit the driver's wallet, set payment CONFIRMED, and persist.
    - I used transactional boundaries and idempotency guards and created traceable transaction IDs to link wallet transactions back to payments and rides.

Cross-cutting engineering patterns I applied

- **Id-based comparisons** for ownership checks to avoid proxy/equality pitfalls.
- **Transactional boundaries** (@Transactional) around mutating flows: wallet changes, ride status changes, and payment orchestration.
- **Defensive null checks** and structured logging (Slf4j) in every service method for observability and easier debug in production.
- **DTO mapping** with ModelMapper only after persistence so the returned DTO reflects the canonical state.
- **Idempotency & reconciliation hooks**: I added idempotency guards in strategies and explicit FAILED status persists to make reconciliation deterministic.
- **Separation of concerns**: repositories handle persistence, services drive business rules, strategies encapsulate payment method differences.
- **Extensibility**: I left clear integration points for SecurityContext wiring (getCurrentDriver() / getCurrentRider()), event publishing (RideEndedEvent), and metrics/audit services.

Operational recommendations I left in the code

- Migrate monetary fields to **BigDecimal** and add a data migration plan.
- Add **optimistic locking** (@Version) on Wallet and possibly Ride to prevent concurrent state transitions.
- Move payment processing to an **asynchronous** workflow (publish RideEndedEvent and handle payments in a listener with retry/backoff and idempotency).
- Add **repository id-based query overloads** (findByRiderId, findByDriverId) and DB indexes for scale.
- Introduce domain-specific exceptions (NotFoundException, UnauthorizedException, InvalidStateException) and a central @ControllerAdvice to map them to HTTP responses.

- Instrument metrics (payments_success_total, payments_failed_total, wallet_insufficient_balance_count) and hook into alerting dashboards.

I closed gaps that would cause operational friction (race conditions, ambiguous errors, and poor observability) and left explicit TODOs for money precision, optimistic locking, and event-driven processing. If I should implement any of those next, I can push the BigDecimal migration, implement optimistic locking, or convert the payment flow to an event-driven listener with retries and idempotency.

Note: First hit requestRide then acceptRide and then startRide in the startRide add otp which you can get that in acceptRide api. then hit endRide.