Theory

Vendor Cab and Driver Onboarding & Hierarchy Management System

---

1. Introduction

The **Vendor Cab and Driver Onboarding System** digitizes end-to-end vendor management for a cab fleet operator. Vendors can register their organizations, onboard drivers, allocate vehicles, and maintain compliance through document tracking in one portal.

Real-world challenges addressed:

- Manual onboarding causing delays and inconsistent data.
- Lack of hierarchical visibility across regional vendors.
- Compliance gaps due to expired driver licenses and vehicle permits.

Importance of hierarchical vendor management:
A multi-level vendor structure (Super → Regional → City) allows corporate teams to delegate day-to-day operations while retaining centralized governance. This structure ensures controlled access, faster onboarding, and scalable expansion across cities without losing oversight.

---

2. System Overview

Technology Stack:

- **Backend:** Spring Boot (Java 21), JPA, MySQL
- **Frontend:** React with WindSurf UI components, Vite tooling
- **Database:** MySQL 8 (configurable for PostgreSQL with minor dialect changes)

Core Modules:

- Vendor Management
- Driver Registry
- Vehicle Management
- Document Library
- Role-Permission Engine
- Notification & Alerts
- Authentication & Audit Logging

Integration Flow:

- The React frontend calls RESTful APIs hosted by Spring Boot.
- JWT-based authentication secures every request.
- MySQL stores all master data (vendors, drivers, vehicles) and transactional records (documents, notifications).

Figure 2.1 – System Architecture Diagram (Backend-Frontend Interaction)

> *Placeholder: Include layered diagram showing user → React UI → API Gateway → Spring Services → Database & File Storage.*

---

3. Multi-Level Vendor Hierarchy

The platform models vendors in a parent-child chain, enabling Super Vendors to supervise Regional and City nodes. Each node carries its own users, drivers, and vehicles while inheriting control policies from the parent.

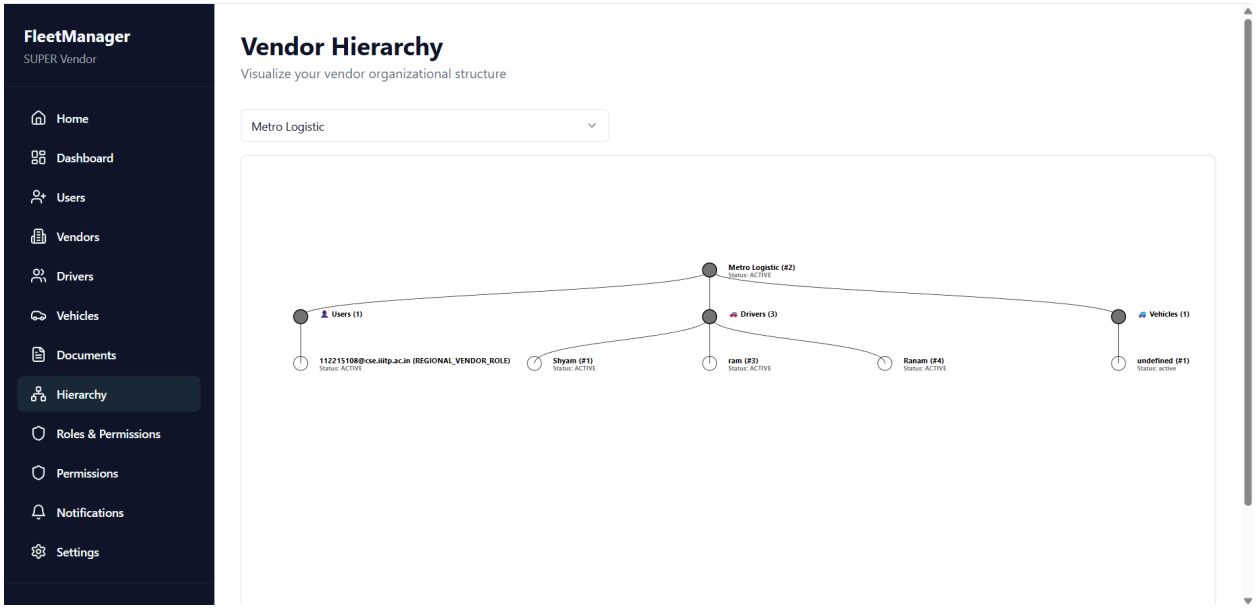
Hierarchy Logic:

- Vendors reference their parent through a `parentVendor` relationship.
- Recursive traversal assembles complete trees for UI visualization and authorization.

```
@GetMapping("/{vendorId}")
@RequiresPermission("CAN_VIEW_VENDOR")
public ResponseEntity<HierarchyNode> getHierarchy(HttpServletRequest request,
@PathVariable Long vendorId) {
   Long actingVendorId = (Long) request.getAttribute("vendorId");
   Vendor acting = vendorRepository.findById(actingVendorId)
           .orElseThrow(() -> new RuntimeException("Acting vendor not found"));
   Vendor target = vendorRepository.findById(vendorId)
           .orElseThrow(() -> new RuntimeException("Target vendor not found"));
   if (!target.getVendorId().equals(actingVendorId) &&
           !hierarchyHelper.isAncestor(acting, target)) {
       throw new RuntimeException("Access denied — can only view your own
subtree");
   }
   return ResponseEntity.ok(hierarchyService.getVendorHierarchy(vendorId));
}
```

Explanation: Recursively aggregates child vendors, their users, drivers, and vehicles, returning an enriched hierarchy DTO to the UI.

Figure 3.1 – Vendor Tree Visualization



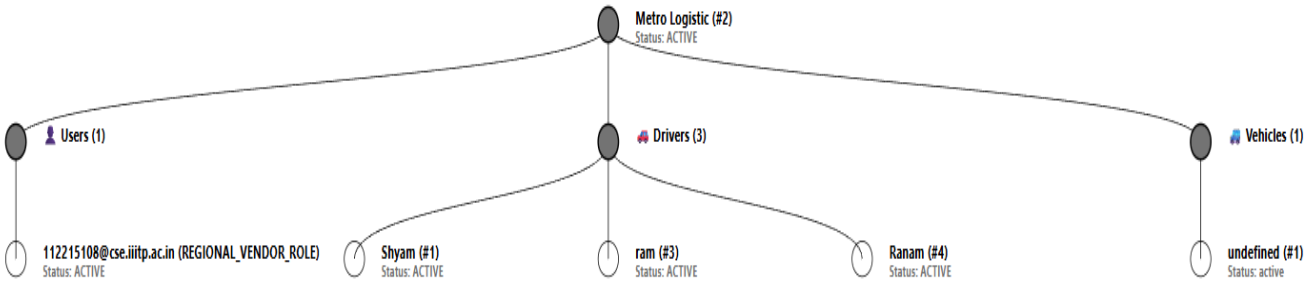
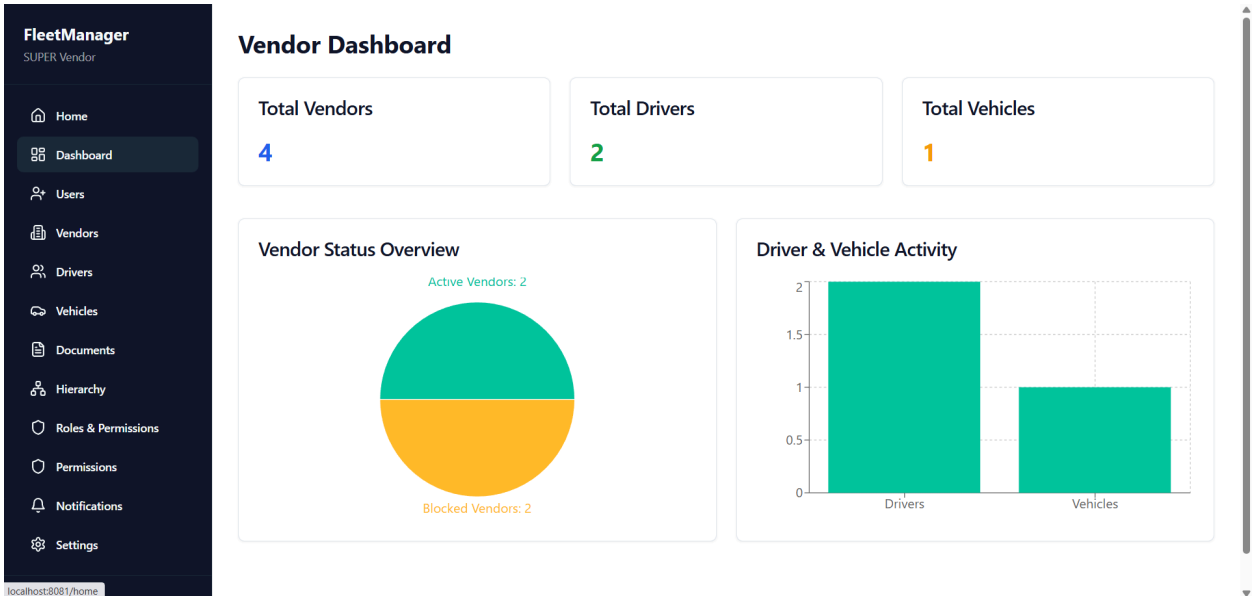


Figure 3.2 – Super Vendor Dashboard showing child vendors



---

4. Vendor, Driver & Vehicle Management

Onboarding Workflow:

1. **Vendor adds driver:** Capture personal details, license number, and assign to city vendor if needed.
2. **Driver links to vehicle:** Drivers can be optionally linked to vehicles, recording registration details.
3. **Document uploads:** License, insurance, and permit documents are attached to specific drivers or vehicles.

```
@GetMapping("/tree")
@RequiresPermission("CAN_VIEW_VENDOR")
public ResponseEntity<List<DriverResponseDto>>
listAllDriversInTree(HttpServletRequest request) {
   Long vendorId = (Long) request.getAttribute("vendorId");
   return ResponseEntity.ok(driverService.listAllDriversInTree(vendorId));
}
```

Explanation: Retrieves drivers across the full vendor subtree, enabling supervisors to audit regional records in one call.

```
// Listing 4.1: Sample Driver Tree Response
{
  "vendorId": 1,
  "vendorName": "Super Vendor - North",
  "drivers": [
    { "id": 101, "name": "Arjun Singh", "status": "ACTIVE", "vehicle": "KA01AB1234" }
  ],
  "children": [
    {
      "vendorId": 2,
      "vendorName": "City Vendor - Bengaluru",
      "drivers": [
        { "id": 203, "name": "Priya Rao", "status": "ACTIVE", "vehicle": "KA02CD5678" }
      ]
    }
  ]
}
```

Explanation: Shows how supervisors visualize drivers across subordinate vendors in a single payload.

Figure 4.1 – Driver Onboarding Form

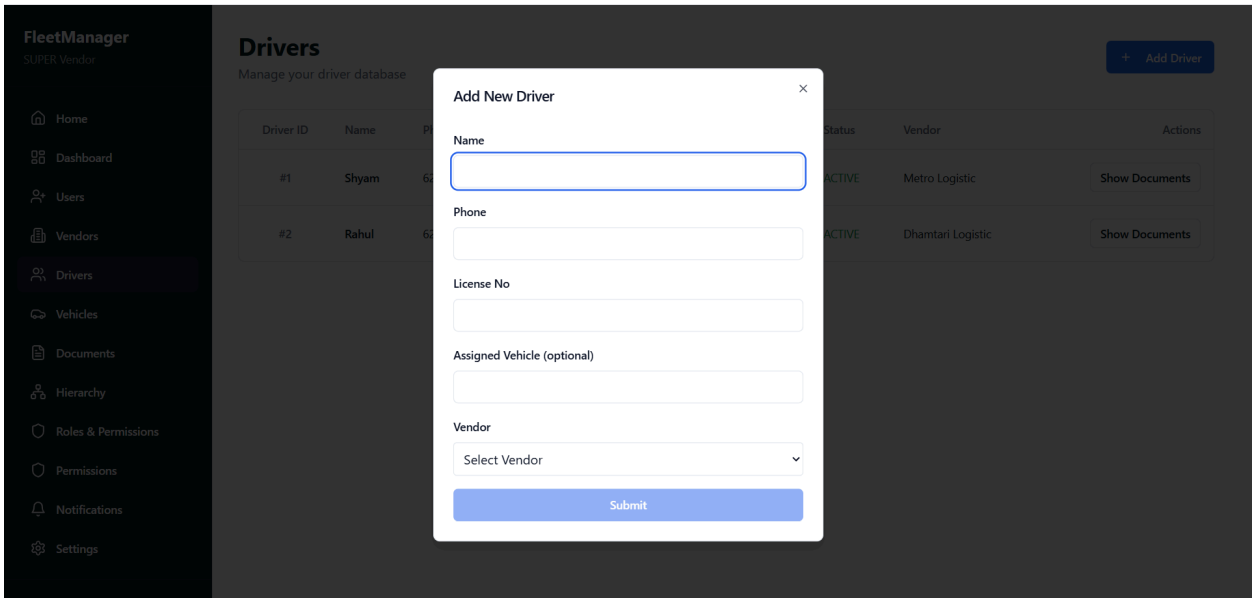
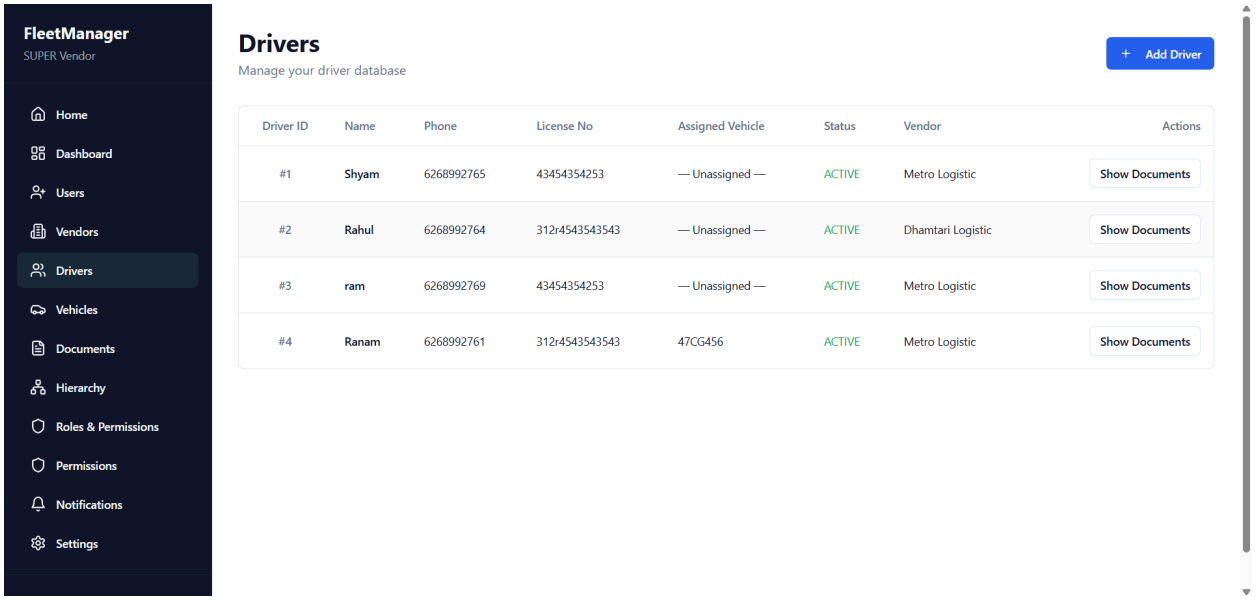


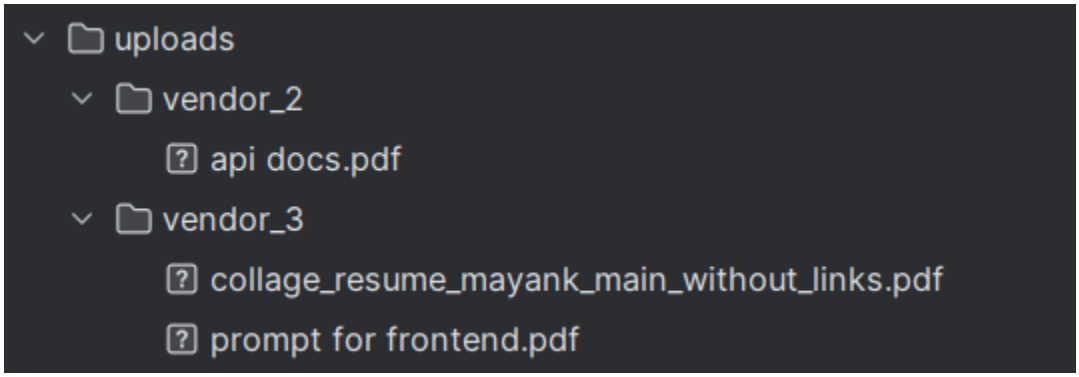
Figure 4.2 – Vehicle Assignment Screen



---

5. Document Verification and Expiry Alerts

Documents are uploaded via the frontend and stored in vendor-specific folders under `/uploads/vendor_<ID>`. Metadata is persisted in MySQL, maintaining relationships to vendors, drivers, and vehicles.




```
// Code Snippet 5.1: Document Upload Handler
String dirPath = UPLOAD_BASE_PATH + "vendor_" + vendorId;
file.transferTo(new File(filePath));
```

Explanation: Ensures each vendor's documents are isolated, simplifying backup, retention, and future migration to cloud storage.

Expiry Alerts:

- Daily scheduler checks documents expiring within seven days.
- Alerts are generated and optionally emailed to vendor contacts.

// Code Snippet 5.2: Document Expiry Scheduler

```java
@Component
@RequiredArgsConstructor
public class DocumentExpiryScheduler {

   private final DocumentRepository documentRepository;
   private final NotificationService notificationService;

   // 🕒 Run daily at 9 AM
   @Scheduled(cron = "0 0 9 * * ?")
   public void checkForExpiringDocuments() {
       LocalDate now = LocalDate.now();
       LocalDate sevenDaysLater = now.plusDays(7);

       List<Document> expiringDocs = documentRepository.findAll().stream()
               .filter(doc -> doc.getExpiryDate() != null &&
                       doc.getExpiryDate().isAfter(now) &&
                       doc.getExpiryDate().isBefore(sevenDaysLater))
               .toList();

       for (Document doc : expiringDocs) {
           notificationService.notifyVendorForExpiringDocument(doc);
       }

       System.out.println("✅ Checked for expiring documents at " + now);
   }
}
```

Explanation: Automates compliance reminders, reducing manual follow-ups and missed renewals.

---

6. Authentication and Role-Permission Management

Authentication:

- Users log in via `/api/auth/login`.
- JWT tokens embed vendor ID, role, and permission claims.
- Spring Security enforces stateless sessions with a custom filter chain.

Authorization:

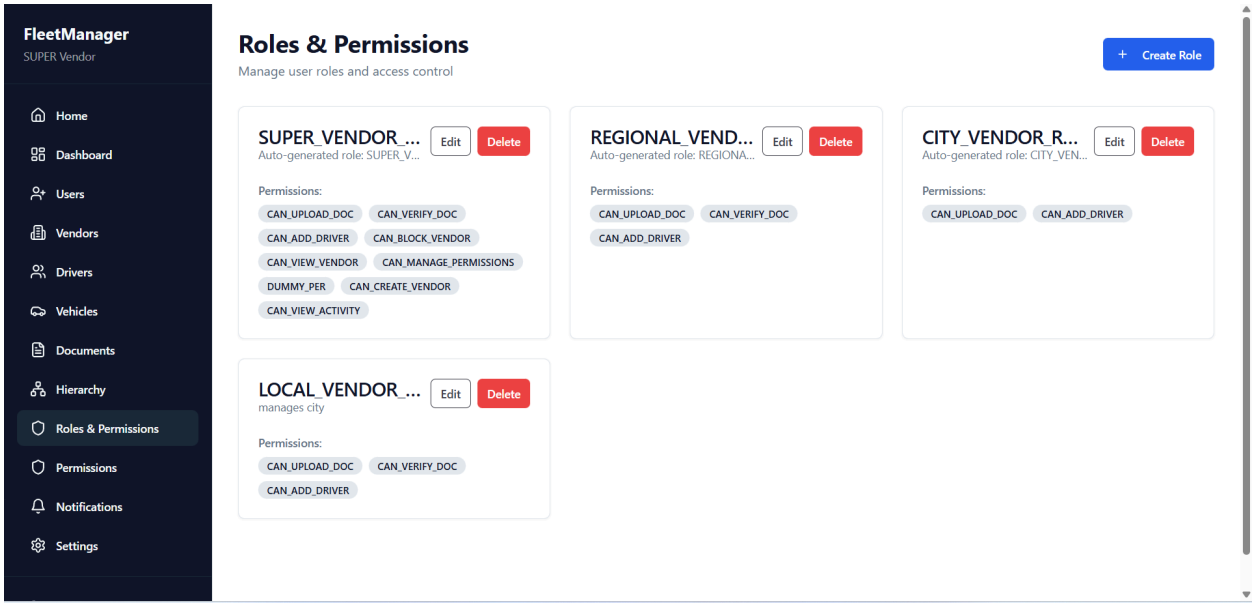
- Roles are mapped to permissions (e.g., `CAN_ADD_DRIVER`, `CAN_UPLOAD_DOC`).

- Controllers use `@RequiresPermission` Aspect to guard endpoints.

```
@PostMapping("/add")
@RequiresPermission("CAN_ADD_DRIVER")
public ResponseEntity<DriverResponseDto> addDriver(HttpServletRequest request,
                                                   @RequestBody DriverRequestDto
driverRequest) {
   Long vendorId = (Long) request.getAttribute("vendorId");
   DriverResponseDto response = driverService.addDriver(vendorId,
driverRequest);
   return ResponseEntity.ok(response);
}
```

Explanation: Prevents unauthorized vendors from onboarding drivers outside their hierarchy.

Figure 6.1 – Role-Permission Mapping



.

---

7. Notification and Alert System

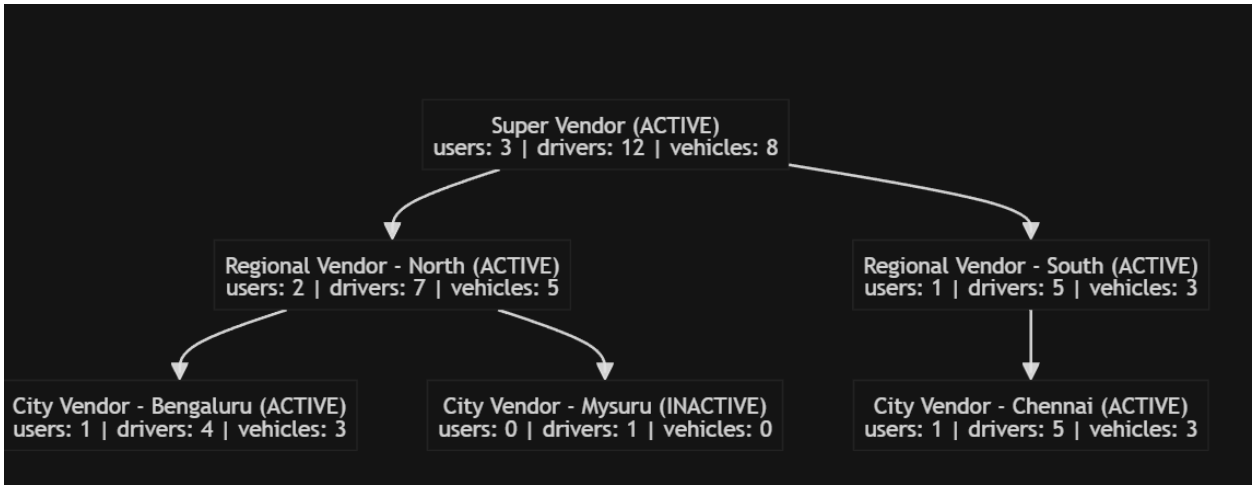
Notification Entity: Records messages with type (e.g., `DOCUMENT_EXPIRY`), timestamps, and vendor association in the database.

Alert Workflow:

1. Scheduler identifies expiring documents.
2. Notification service logs an entry and optionally sends an email using configured SMTP credentials.

```
// Code Snippet 7.1: Notification Trigger
notificationService.notifyVendorForExpiringDocument(document);
```

Explanation: Centralizes alert creation; adding SMS or push channels later requires wiring into this method.



Explanation : Vendor hierarchy structure

8. System Features, Design Choices & Trade-offs

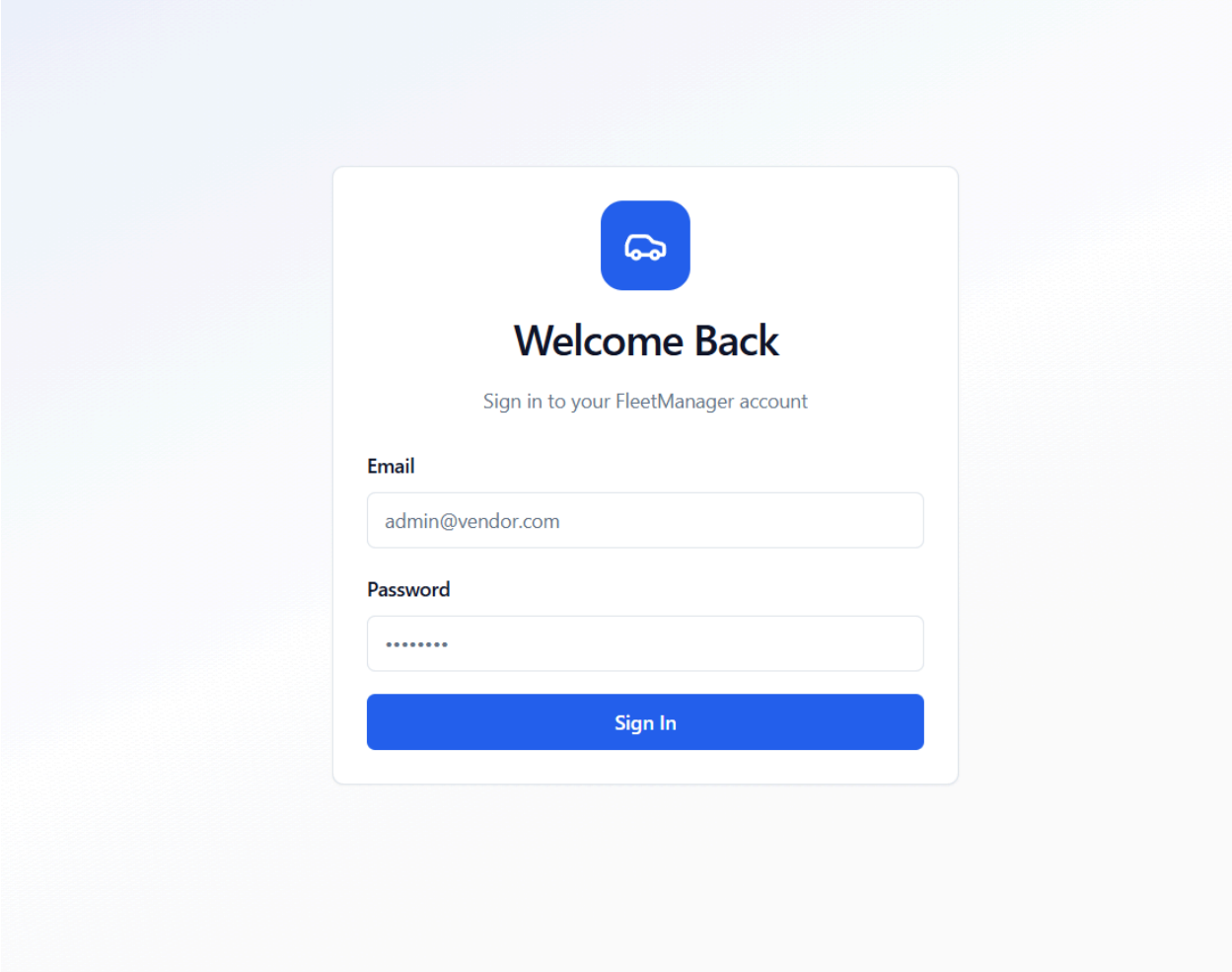
- **Authentication & Security:** JWT, role-based access, CORS configuration for curated origins, Bcrypt password hashing.
- **Fault Tolerance:** Document uploads validate directory creation and handle IO exceptions gracefully.
- **OOP Principles:**
  - Entity relationships (Vendor ↔ Driver ↔ Vehicle).
  - Helper utilities (`VendorHierarchyHelper`) encapsulate reuse.
  - Aspect-oriented permission checks reduce controller clutter.
- **Cost Estimation & Performance:**
  - MySQL tuned with lazy relationships; caching potential for read-heavy APIs.
  - Background jobs scheduled outside peak hours reduce runtime load.
- **Trade-offs:**

    - **SQL vs NoSQL:** Chosen SQL for ACID compliance and complex joins; future read scaling can add caches.
    - **Local Storage vs Cloud S3:** Local upload for simplicity; S3 migration planned for durability and CDN integration.
- **Monitoring & Logging:**
    - Activity logs capture driver status changes and document deletions.
    - HTTP request logging and scheduler outputs provide observability.
- **Error Handling:**
    - Centralized with Spring's `@ControllerAdvice` (extendable for custom error codes and localization).

---

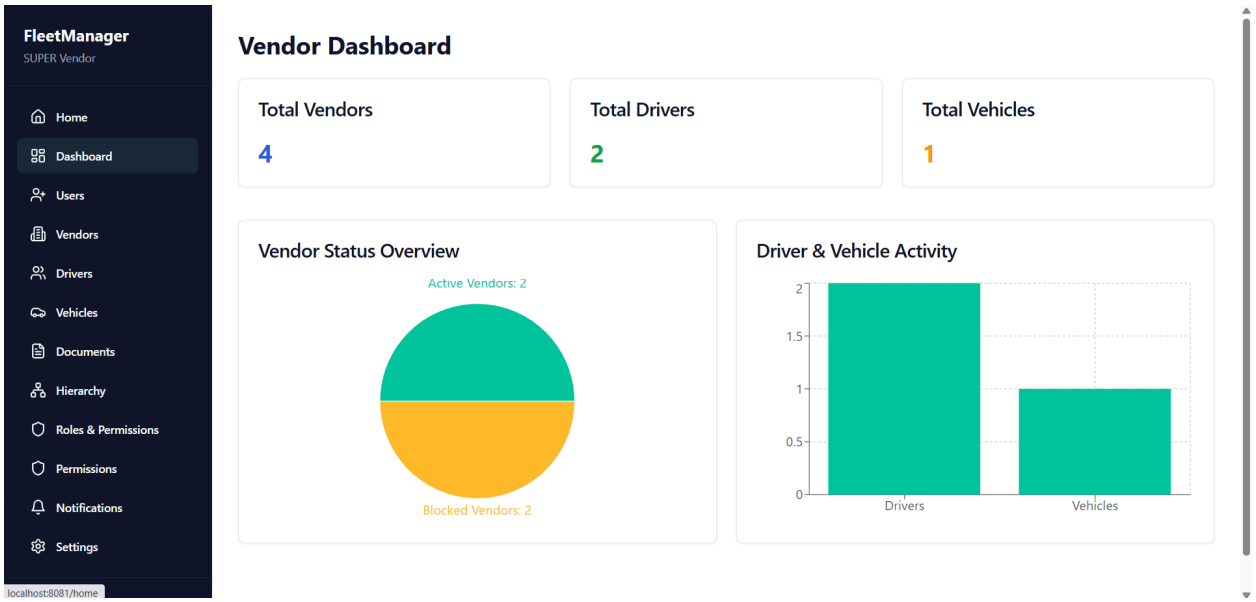
9. Snapshots and Demonstration

Figure 9.1 – Login Page



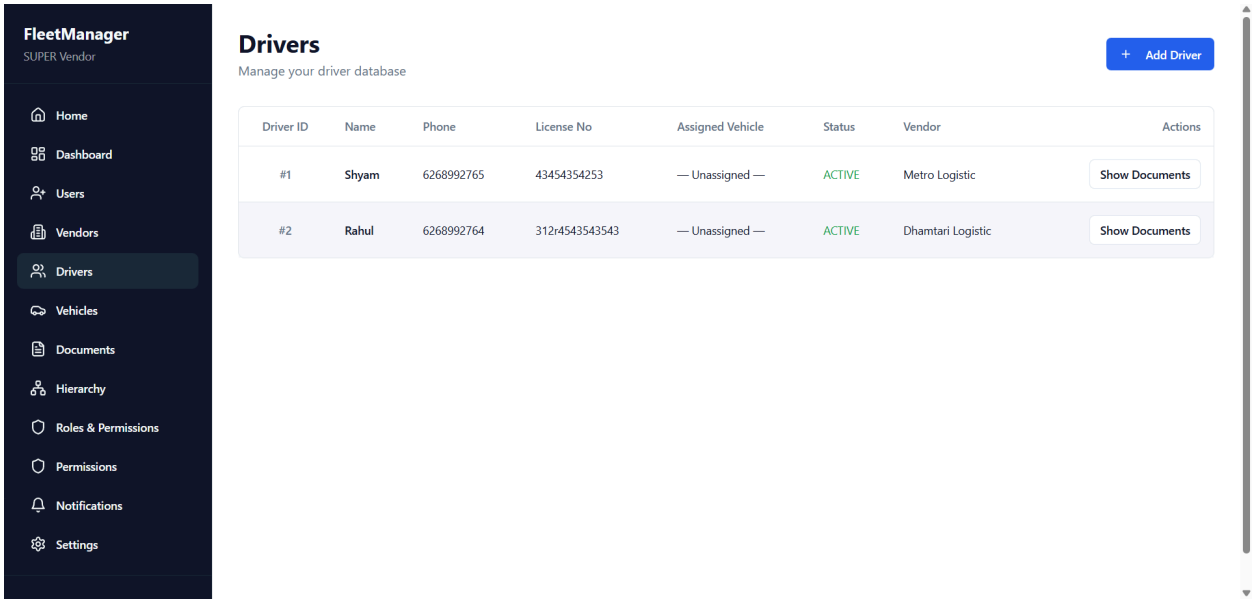
Caption: Displays secure login form, validating credentials before issuing JWT tokens.

Figure 9.2 – Super Vendor Dashboard



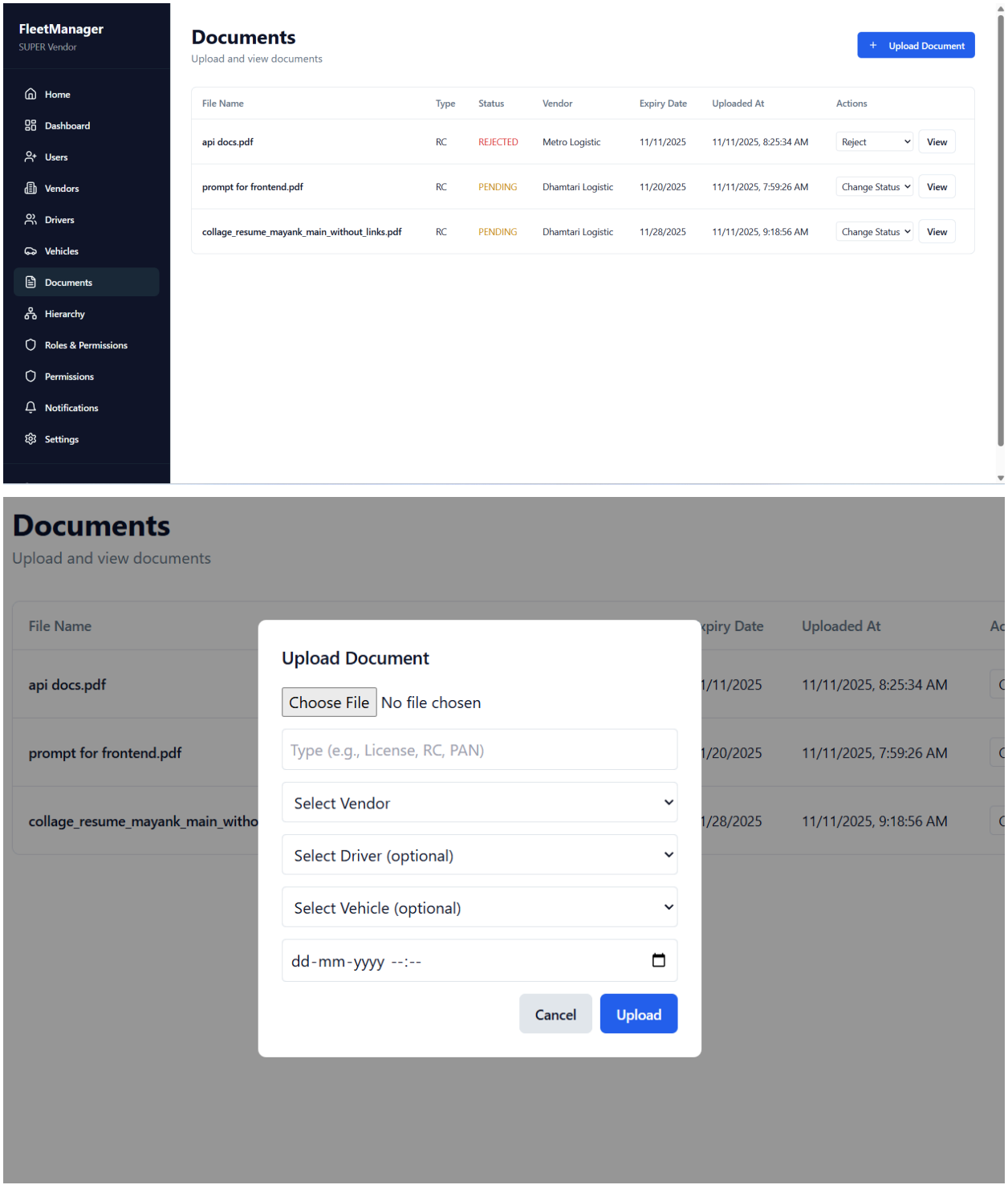
Caption: Aggregated metrics for active vendors, drivers, vehicles; entry point to manage subordinates.

Figure 9.3 – Driver Listing View



Caption: Filterable grid showing driver statuses, vendor assignments, and quick actions.

Figure 9.4 – Document Upload Modal



Caption: Uploads driver or vehicle documents with type classification and expiry date.

10. Code Snippet Highlights

```
// Code Snippet 10.1: HierarchyNode Builder
private HierarchyNode buildHierarchyNode(Vendor vendor) { ... }
```

Commentary: Captures multi-level vendor relationships recursively for the frontend tree visualization.

```
// Code Snippet 10.2: Document Upload Service
String dirPath = UPLOAD_BASE_PATH + "vendor_" + vendorId;
file.transferTo(new File(filePath));
```

Commentary: Handles file persistence with vendor isolation and metadata storage.

```
// Code Snippet 10.3: Driver Tree Fetch API
@GetMapping("/tree")
public ResponseEntity<List<DriverResponseDto>> listAllDriversInTree(...) { ... }
```

Commentary: Returns driver data across the vendor hierarchy for supervisory dashboards.

```
// Code Snippet 10.4: Document Expiry Scheduler
@Scheduled(cron = "0 0 9 * * ?")
public void checkForExpiringDocuments() { ... }
```

Commentary: Automates detection of upcoming document expiries.

// Code Snippet 10.5: Security Configuration & CORS

```
// Code Snippet 10.5: Security Configuration & CORS
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception { ... }
```

Commentary: Enforces stateless security, JWT filter ordering, and controlled cross-origin access.

---

11. Results and Outcomes

- **Centralized Control:** Supervisors gain a unified dashboard for vendor relationships, driver compliance, and document statuses.
- **Scalable Onboarding:** Parent-child vendor model enables quick onboarding of new regions without sacrificing policy enforcement.
- **Real-time Compliance Tracking:** Scheduled checks and notifications reduce risk of expired licenses and documents.
- **Maintainability:** Spring Boot modular design and DTO mapping simplify future enhancements and code reuse.
- **Reusability:** Components such as notification service, hierarchy helper, and JWT utilities can be repurposed for other internal platforms.

---

12. Conclusion and Future Scope

The system delivers a robust framework for vendor hierarchy management, driver onboarding, and document compliance. It balances **security**, **scalability**, and **usability**, making it suitable for enterprise fleet operations.

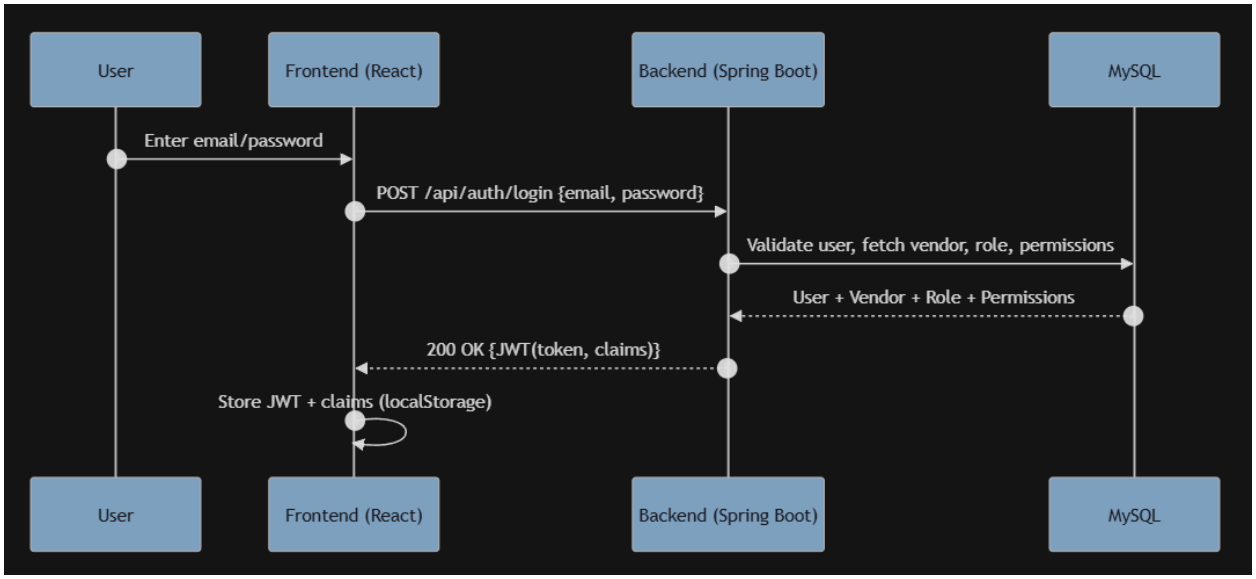
Future Enhancements:

- AI-powered document verification to auto-validate uploaded identity proofs.
- Real-time GPS tracking for drivers and vehicles.
- Analytics dashboards for performance metrics, route optimization, and SLA adherence.
- Migration of document storage to AWS S3 with CDN distribution and lifecycle policies.
- Microservice decomposition for independent scaling of hierarchy, document, and notification services.

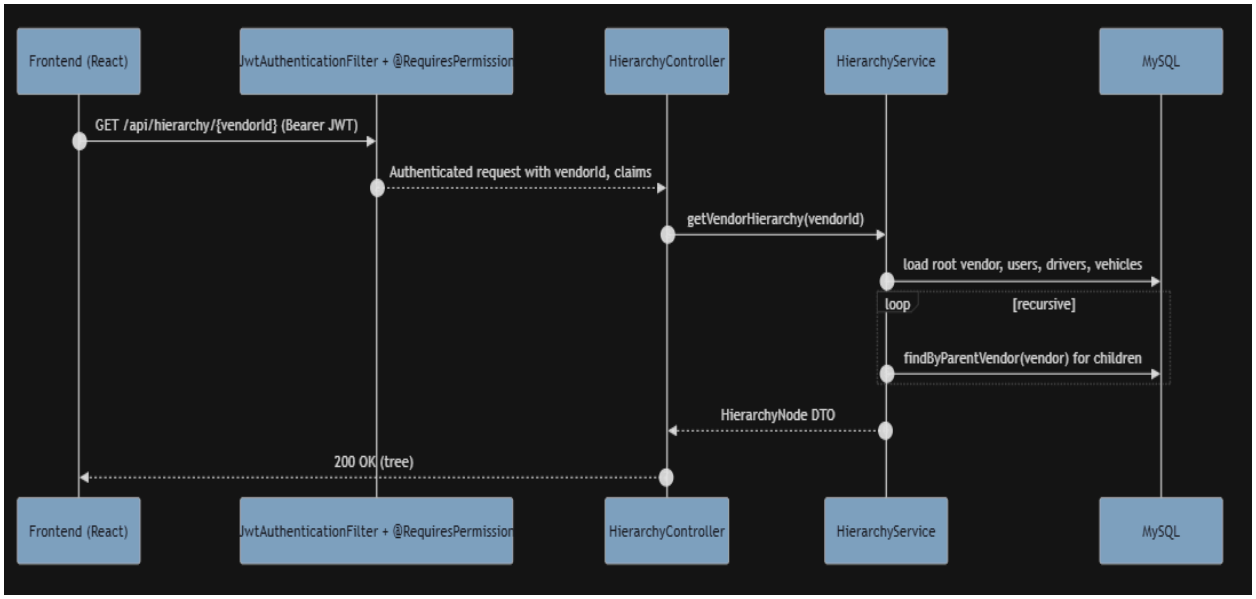
Go To Tab 2 for Sequence diagram

Sequence diagram

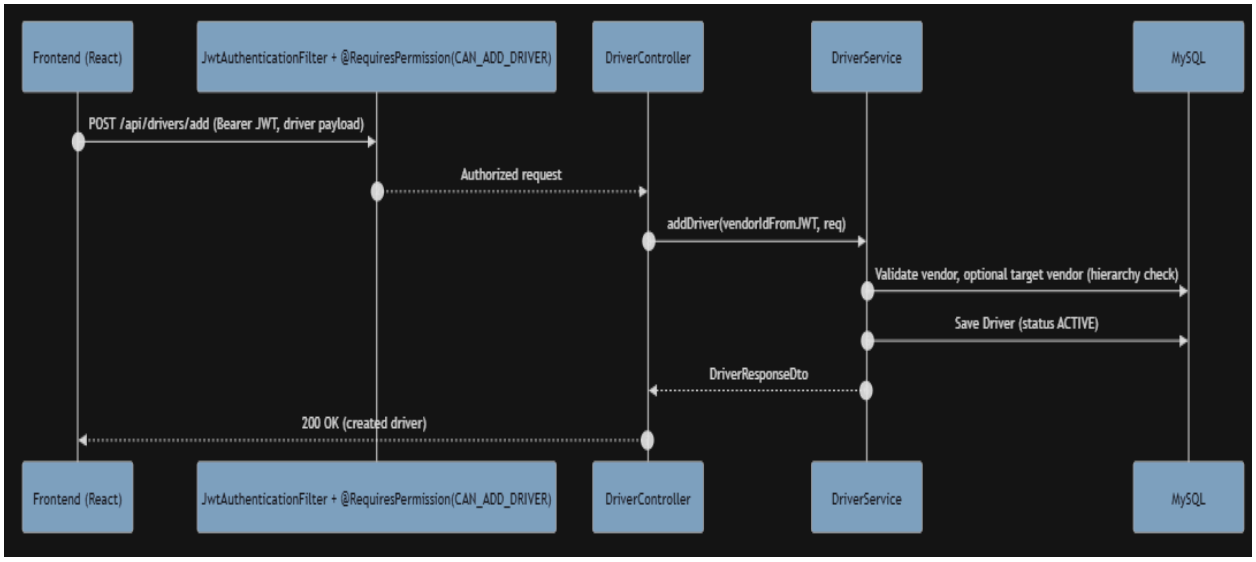
1) Authentication (Login → JWT)



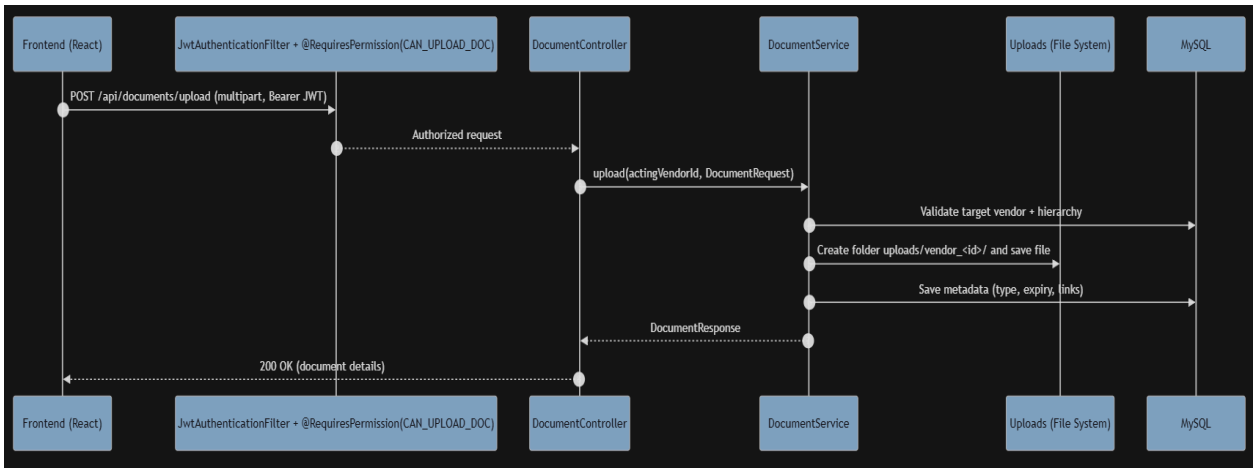
2) Vendor Hierarchy (Tree Fetch)

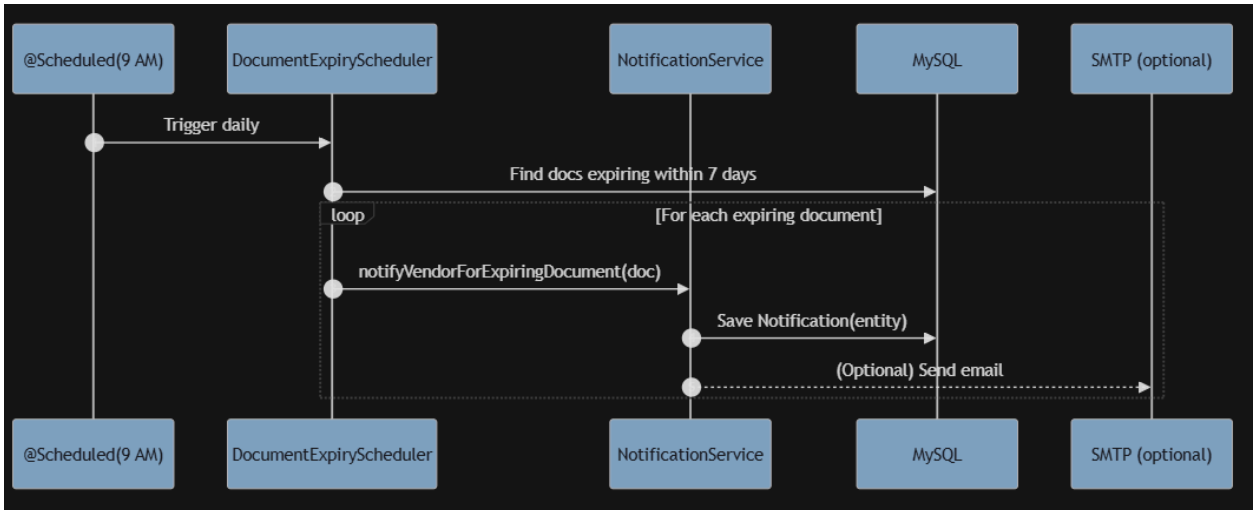


3) Driver Onboarding (Add Driver)

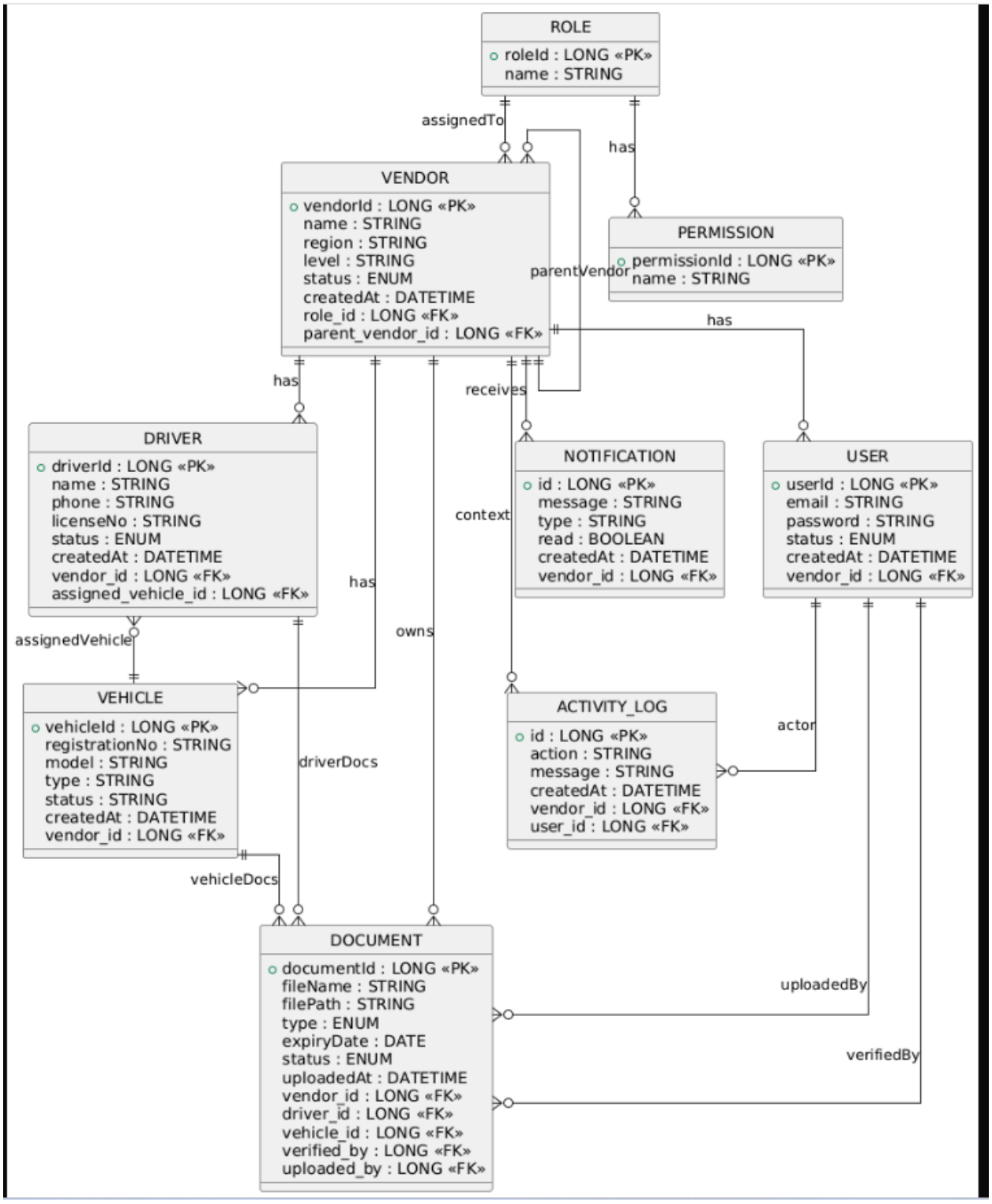


4) Document Upload (Vendor/Driver/Vehicle)




@Scheduled(9 AM)
DocumentExpiryScheduler
NotificationService
MySQL
SMTP (optional)
Trigger daily
Find docs expiring within 7 days
loop
[For each expiring document]
notifyVendorForExpiringDocument(doc)
Save Notification(entity)
(Optional) Send email
@Scheduled(9 AM)
DocumentExpiryScheduler
NotificationService
MySQL
SMTP (optional)

ER diagram And Entity


ROLE
roleId : LONG «PK»
name : STRING
assignedTo
has
VENDOR
vendorId : LONG «PK»
name : STRING
region : STRING
level : STRING
status : ENUM
createdAt : DATETIME
role_id : LONG «FK»
parent_vendor_id : LONG «FK»
parentVendor
PERMISSION
permissionId : LONG «PK»
name : STRING
has
has
receives
DRIVER
driverId : LONG «PK»
name : STRING
phone : STRING
licenseNo : STRING
status : ENUM
createdAt : DATETIME
vendor_id : LONG «FK»
assigned_vehicle_id : LONG «FK»
NOTIFICATION
id : LONG «PK»
message : STRING
type : STRING
read : BOOLEAN
createdAt : DATETIME
vendor_id : LONG «FK»
context
USER
userId : LONG «PK»
email : STRING
password : STRING
status : ENUM
createdAt : DATETIME
vendor_id : LONG «FK»
has
assignedVehicle
owns
VEHICLE
vehicleId : LONG «PK»
registrationNo : STRING
model : STRING
type : STRING
status : STRING
createdAt : DATETIME
vendor_id : LONG «FK»
ACTIVITY_LOG
id : LONG «PK»
action : STRING
message : STRING
createdAt : DATETIME
vendor_id : LONG «FK»
user_id : LONG «FK»
actor
driverDocs
vehicleDocs
DOCUMENT
documentId : LONG «PK»
fileName : STRING
filePath : STRING
type : ENUM
expiryDate : DATE
status : ENUM
uploadedAt : DATETIME
vendor_id : LONG «FK»
driver_id : LONG «FK»
vehicle_id : LONG «FK»
verified_by : LONG «FK»
uploaded_by : LONG «FK»
uploadedBy
verifiedBy

DOCUMENT ENTITY

```
@Entity
@Table(name = "documents")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Document {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long documentId;
   private String fileName;
   @ManyToOne
   @JoinColumn(name = "vendor_id", nullable = false)
   private Vendor vendor;           // ✅ always required
   @ManyToOne
   @JoinColumn(name = "driver_id", nullable = true)
   private Driver driver;           // optional
   @ManyToOne
   @JoinColumn(name = "vehicle_id", nullable = true)
   private Vehicle vehicle;
   @Enumerated(EnumType.STRING)
   private DocumentType type; // LICENSE, RC, INSURANCE
   private String filePath; // local storage path
   private LocalDate expiryDate;
   @Enumerated(EnumType.STRING)
   private DocumentStatus status = DocumentStatus.ACTIVE; // ACTIVE, EXPIRED,
DELETED, PENDING
   @ManyToOne
   @JoinColumn(name = "verified_by")
   private User verifiedBy;
   @ManyToOne
   @JoinColumn(name = "uploaded_by")
   private User uploadedBy;
   private LocalDateTime uploadedAt = LocalDateTime.now();
}
```

USER ENTITY

```java
@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long userId;

   @ManyToOne
   @JoinColumn(name = "vendor_id")
   private Vendor vendor;

   private String email;
   private String passwordHash;

   private boolean temporaryPassword = true;
   @Enumerated(EnumType.STRING)
   private UserStatus status; // ACTIVE, BLOCKED, INACTIVE

   private LocalDateTime lastLogin;
   private LocalDateTime createdAt = LocalDateTime.now();
}
```

VENDOR ENTITY

```java
@Table(name = "vendors")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Vendor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long vendorId;

    private String name;
    private String region;//JUST FOR UI
    private String level; // e.g., SUPER, REGIONAL, CITY//JUST FOR UI
    @Enumerated(EnumType.STRING)
    private VendorStatus status; // ACTIVE, BLOCKED, INACTIVE

    @ManyToOne
    @JoinColumn(name = "role_id")
    private Role role;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "parent_vendor_id")
    private Vendor parentVendor;

    private LocalDateTime createdAt = LocalDateTime.now();

    // Relationships
    @OneToMany(mappedBy = "vendor")
    private List<User> users;

    @OneToMany(mappedBy = "vendor")
    private List<Driver> drivers;

    @OneToMany(mappedBy = "vendor")
    private List<Vehicle> vehicles;
}
```

PERMISSION ENTITY

```
@Entity
@Table(name = "permissions")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Permission {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long permissionId;

   private String name;
   private String description;
}
```

ROLE ENTITY

```
@Entity
@Table(name = "roles")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Role {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long roleId;

   private String name; // SUPER_VENDOR, REGIONAL_VENDOR, ADMIN, etc.
   private String description;

   @ManyToMany(fetch = FetchType.EAGER)
   @JoinTable(
       name = "role_permissions",
       joinColumns = @JoinColumn(name = "role_id"),
       inverseJoinColumns = @JoinColumn(name = "permission_id")
   )
   private Set<Permission> permissions = new HashSet<>();

   private LocalDateTime createdAt = LocalDateTime.now();
}
```

Important Util

```
private HierarchyNode buildHierarchyNode(Vendor vendor) {
   List<UserInfo> users = userRepository.findByVendor(vendor)
            .stream()
            .map(u -> new UserInfo(
                    u.getUserId(),
                    u.getEmail(),
                    vendor.getRole() != null ? vendor.getRole().getName() :
"N/A",
                    u.getStatus().name()
            ))
            .collect(Collectors.toList());

   List<DriverInfo> drivers = driverRepository.findByVendor(vendor)
            .stream()
            .map(d -> new DriverInfo(d.getDriverId(), d.getName(),
d.getStatus().name()))
            .collect(Collectors.toList());

   List<VehicleInfo> vehicles = vehicleRepository.findByVendor(vendor)
            .stream()
            .map(v -> new VehicleInfo(v.getVehicleId(), v.getRegistrationNo(),
v.getStatus()))
            .collect(Collectors.toList());

   List<HierarchyNode> children = vendorRepository.findByParentVendor(vendor)
            .stream()
            .map(this::buildHierarchyNode)
            .collect(Collectors.toList());

   return new HierarchyNode(
            vendor.getVendorId(),
            vendor.getName(),
            vendor.getStatus().name(),
            users,
            drivers,
            vehicles,
            children
   );
}
```

Explanation : Builds The Hierarchy Tree

```
@Override
  public void notifyVendorForExpiringDocument(Document document) {
      Vendor vendor = document.getVendor();

      String msg = String.format(
              "Document '%s' is expiring on %s. Please renew it soon.",
              document.getFileName(),
              document.getExpiryDate()
      );

      Notification notification = new Notification();
      notification.setMessage(msg);
      notification.setType("DOCUMENT_EXPIRY");
      notification.setVendor(vendor);
      notification.setCreatedAt(LocalDateTime.now());

      notificationRepository.save(notification);

      // Optionally send an email
//        mailUtil.sendMail(vendor.getEmail(), "Document Expiry Reminder",
msg);
   }
```

EXPLANATION : TO CREATE NOTIFICATION FOR DOC THAT'S ABOUT TO BE EXPIRED