SERVICE MANAGEMENT SYSTEM

DETAILED SYSTEM DESIGN DOCUMENT

By: POLAKU JOSHIKA SREE

1. PURPOSE

This document provides a **comprehensive technical description** of the **Service Management System**, including:

- System architecture
- Microservice responsibilities
- REST APIs (detailed)
- Data models
- Security mechanisms
- Error handling
- End-to-end request flow
- Deployment strategy

Intended Audience

- Project Evaluators
- Developers
- Interview Panels
- Maintenance Teams

2. SYSTEM OVERVIEW

2.1 Business Objective

The Service Management System is a **multi-role home services platform** that enables customers to book services, managers to assign technicians,

technicians to complete jobs, and the system to automatically generate invoices and notifications.

Core Goals

- Reduce manual service coordination
- Improve transparency and tracking
- Ensure scalability using microservices
- Maintain strong security and role isolation

3. HIGH-LEVEL ARCHITECTURE

Architecture Style

- Microservices Architecture
- REST + Event-Driven Communication
- Database-per-service
- Containerized Deployment

Core Components

```
Angular UI
  |
API Gateway
  |
------------------------------------------------
| Auth | Catalog | Booking | Billing | Notify |
------------------------------------------------
  |
MongoDB (per service)
  |
RabbitMQ (Async Events)
```

4. TECHNOLOGY STACK

Backend

- Java 17
- Spring Boot 3.x
- Spring Web (REST)
- Spring Security + JWT
- Spring Data MongoDB

Spring Cloud

- Spring Cloud Gateway
- Netflix Eureka
- OpenFeign
- RabbitMQ (Spring AMQP)

Frontend

- Angular 16+
- Forms
- Routing

DevOps

- Docker & Docker Compose
- Jenkins / GitHub Actions
- SonarQube

5. MICROSERVICES & API DESIGN

5.1 AUTH SERVICE (Port: 8081)

Responsibilities

- User registration
- User authentication
- JWT token generation
- Technician profile management
- Role assignment

API BOX — AUTH SERVICE

Register Customer

POST /auth/register/customer

Request

```
{
  "username": "alice",
  "email": "alice@gmail.com",
  "password": "Password@123"
}
```

Response

```
{
  "message": "Customer registered successfully"
}
```

Register Technician

POST /auth/register/technician

Request

```
{
  "username": "tech_john",
  "email": "john@tech.com",
  "password": "Password@123",
  "skills": ["AC Repair", "Electrical"],
  "experienceYears": 5,
  "idProofType": "AADHAAR"
}
```

Login (All Roles)

POST /auth/login

Request

```
{
  "email": "alice@gmail.com",
  "password": "Password@123"
}
```

Response

```
{
  "token": "<JWT_TOKEN>",
  "roles": ["ROLE_CUSTOMER"]
}
```

5.2 SERVICE CATALOG SERVICE (Port: 8082)

Responsibilities

- Manage service categories
- Manage individual services

- Serve pricing & descriptions

API BOX — SERVICE CATALOG

Get All Categories

GET /services/categories

Get Services by Category

GET /services/items/{categoryId}

Response

```
[
 {
   "id": "svc401",
   "name": "Leaky Faucet Repair",
   "basePrice": 450
 }
]
```

5.3 BOOKING SERVICE (Port: 8083)

Responsibilities

- Create bookings
- Assign technicians
- Track booking lifecycle
- Publish booking events

API BOX — BOOKING SERVICE

Create Booking (Customer)

POST /bookings

Request

```
{
  "serviceId": "svc401",
  "problemDescription": "AC not cooling",
  "scheduledDate": "2026-01-10T10:00:00",
  "addressLine1": "Flat 402",
  "city": "Bangalore",
  "zipCode": "560001"
}
```

Assign Technician (Manager)

PUT /bookings/{bookingId}/assign?technicianId={techId}

Complete Booking (Technician)

PUT /bookings/{bookingId}/complete

Booking Status Flow

PENDING → ASSIGNED → COMPLETED

5.4 BILLING SERVICE (Port: 8084)

Responsibilities

- Generate invoice after service completion
- Calculate tax (GST)
- Track payment status

API BOX — BILLING SERVICE

Get Invoice by Booking

GET /billing/invoices/{bookingId}

Response

```
{
  "amount": 450,
  "taxAmount": 81,
  "totalAmount": 531,
  "paymentStatus": "PENDING"
}
```

5.5 NOTIFICATION SERVICE (Port: 8085)

Responsibilities

- Consume RabbitMQ events
- Send transactional emails

EVENT BOX — RABBITMQ EVENTS

Event Name	Trigger
BOOKING_CREATED	Booking created

BOOKING_ASSIGNED	Technician assigned
BOOKING_COMPLETED	Job completed

6. END-TO-END FLOW (HOW EVERYTHING WORKS)

Booking Creation Flow

1. Customer selects service in Angular UI
2. Angular → API Gateway → Booking Service
3. Booking Service calls Catalog Service (Feign)
4. Booking saved in MongoDB
5. BOOKING_CREATED event sent to RabbitMQ
6. Notification Service sends email

Technician Assignment Flow

1. Manager assigns technician
2. Booking Service updates status
3. BOOKING_ASSIGNED event published
4. Notification Service emails technician

Service Completion & Billing Flow

1. Technician completes job
2. Booking status updated to COMPLETED
3. BOOKING_COMPLETED event published
4. Billing Service generates invoice
5. Invoice saved in MongoDB
6. Notification sent to customer

7.DATA DESIGN (LOW-LEVEL DESIGN)

The Service Management System follows a **Database-per-Service design pattern**, where each microservice maintains its own MongoDB database. This approach ensures **loose coupling, independent scalability, data security, and clear ownership of data**.

Auth Service – Data Design

User Entity

- Stores core authentication and authorization details for all users.
- Represents Customers, Technicians, Managers, and Admins.
- Contains username, email, encrypted password, and assigned roles.
- Used for JWT generation and role-based access control.
- Passwords are stored using **BCrypt encryption** for security.

Technician Profile Entity

- Stores additional details specific to technicians.
- Linked to the User entity via user identifier.
- Maintains technician skills, experience, approval status, and availability.
- Used during technician assignment and workload management.
- Approval status ensures only verified technicians can be assigned.

Service Catalog – Data Design

Service Category Entity

- Represents high-level groupings of services (e.g., Plumbing, AC Services).
- Used to organize services for better browsing and filtering.
- Managed only by Admin users.

Service Entity

- Represents individual service offerings.
- Linked to a Service Category.
- Stores service name, description, base price, and image reference.
- Pricing data is treated as the source of truth during booking creation.

Booking Service – Data Design

Booking Entity

- Represents a service request raised by a customer.
- Acts as the **core transactional entity** of the system.
- Stores references to customer, service, and assigned technician.
- Tracks booking lifecycle using status values:
  REQUESTED → ASSIGNED → IN_PROGRESS → COMPLETED
- Includes scheduling details and service address.
- Immutable after completion to preserve data integrity.

Booking Status History Entity

- Maintains a record of booking status changes.
- Used for auditing and tracking service progress.
- Helps generate operational reports.

Billing Service – Data Design

Invoice Entity

- Represents the financial snapshot of a completed booking.
- Linked to a booking and customer.
- Stores base amount, tax amount, discounts, and total payable amount.
- Tracks payment status (Pending, Paid).
- Created only after booking completion event.

Notification Service – Data Design

Notification Entity

- Stores records of system-generated notifications.
- Captures notification type (Booking, Assignment, Invoice).
- Linked to the recipient user.
- Used for tracking email delivery and audit purposes.

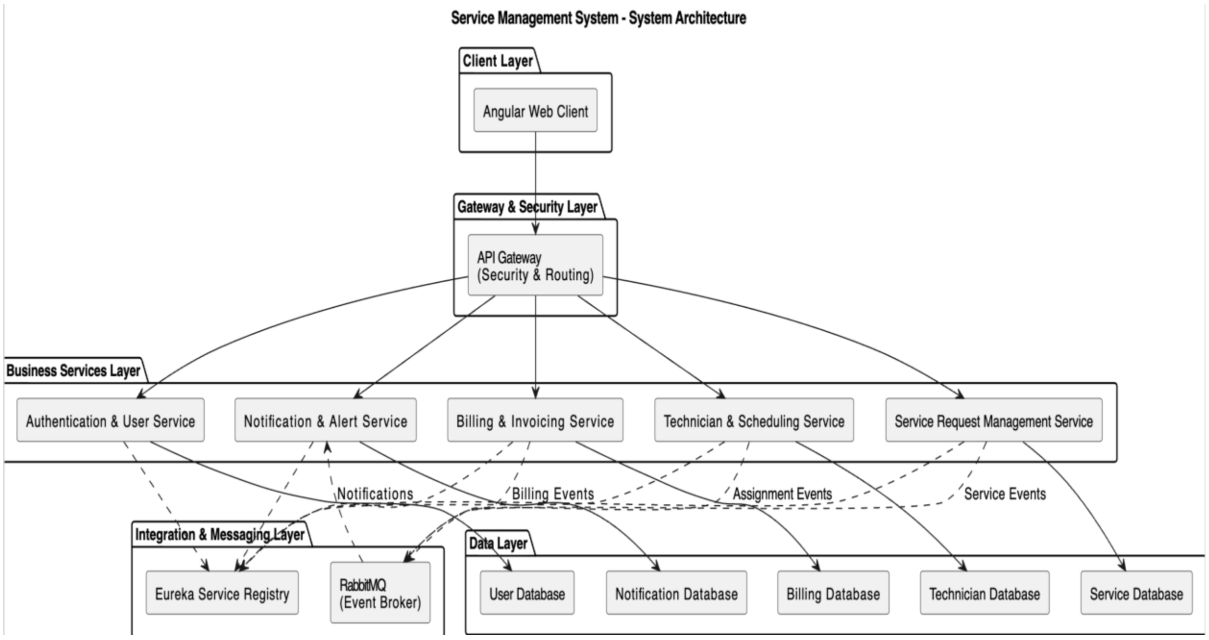
Cross-Service Data Considerations

- Services communicate using **IDs instead of shared schemas** to avoid tight coupling.
- No direct database access between microservices.
- All inter-service communication happens via:
  - REST APIs (OpenFeign)
  - RabbitMQ events
- Read-heavy operations (dashboards) rely on optimized queries and projections.

Data Integrity & Consistency

- Each service validates data before persistence.
- Booking Service validates service data via Service Catalog before booking creation.
- Billing Service relies on booking completion events to ensure correctness.
- Transactions are eventually consistent using event-driven communication.

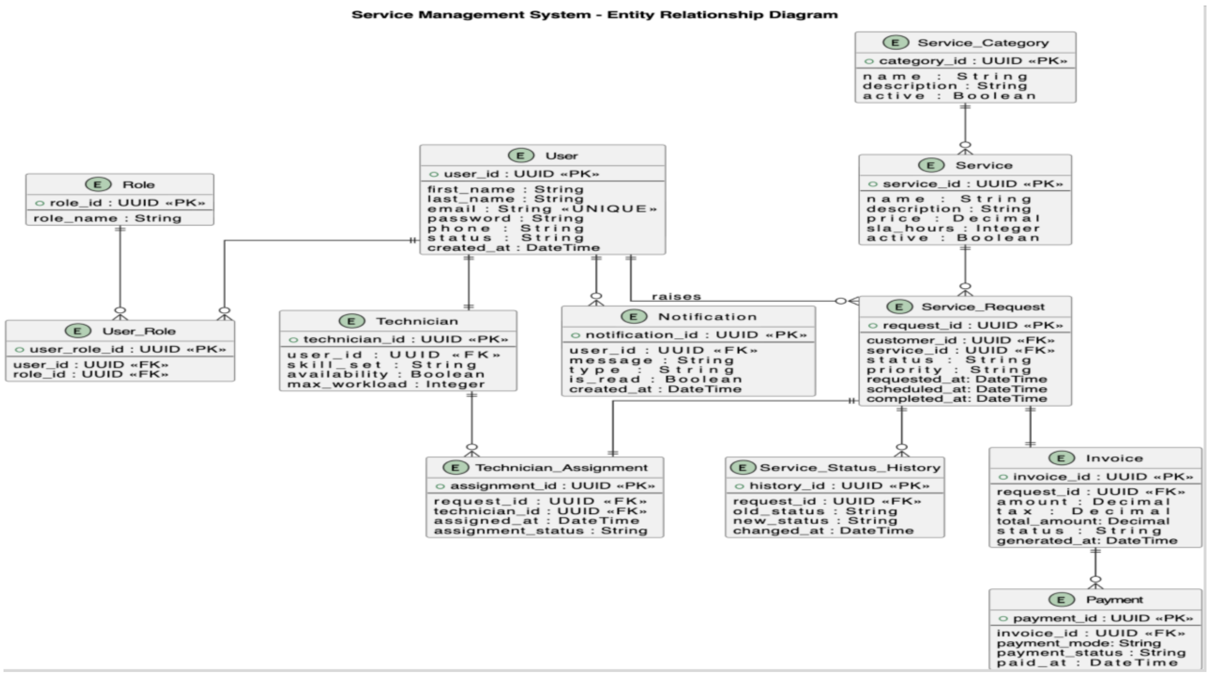
8.System Architecture Diagram:



This diagram represents the **overall system architecture** of the Service Management System.
The system follows a **layered microservices architecture** to ensure scalability, security, and maintainability.

- The **Client Layer** consists of an Angular Single Page Application used by Customers, Technicians, Managers, and Admins.
- All client requests pass through the **API Gateway**, which acts as a centralized entry point responsible for routing, authentication, and security enforcement.
- The **Business Services Layer** contains independent microservices such as Authentication, Booking, Service Management, Billing, and Notification services.
- The **Integration & Messaging Layer** uses RabbitMQ to enable asynchronous event-driven communication between services.
- The **Data Layer** follows a database-per-service pattern using MongoDB, ensuring loose coupling and data isolation.
- **Eureka Service Registry** enables dynamic service discovery and load balancing.This architecture allows individual services to scale independently and improves fault isolation.

Entity Relationship Diagram (ER Diagram):

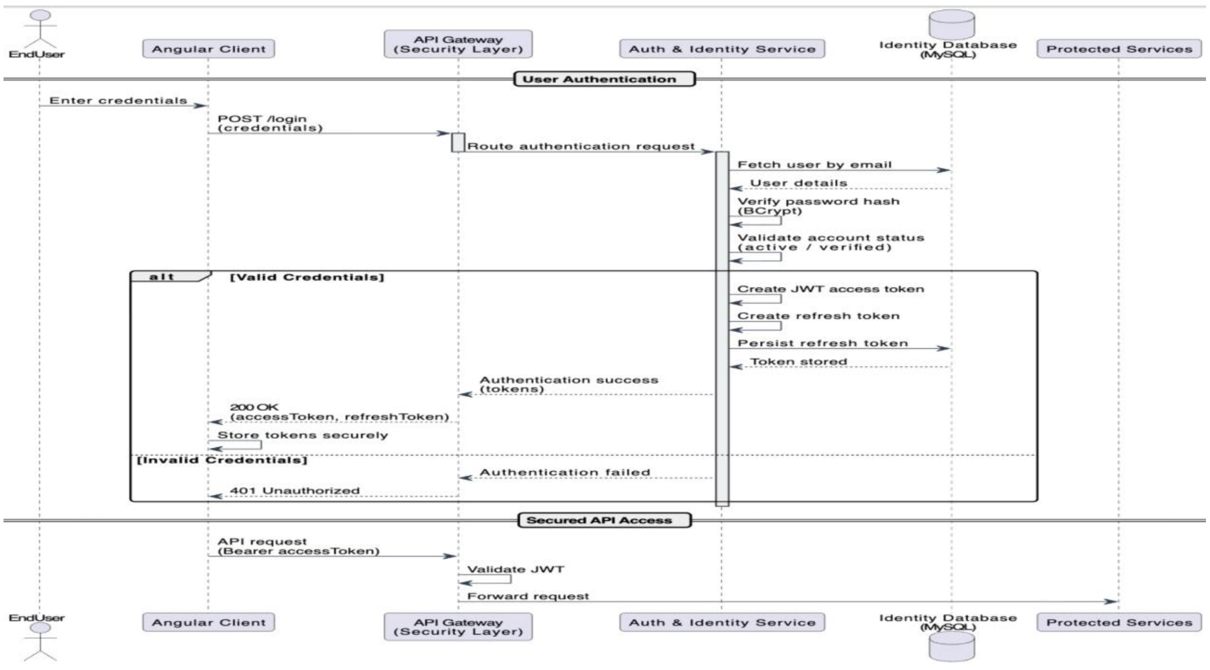


The ER diagram illustrates the **data model and relationships** used across the Service Management System.

- A **User** entity stores common authentication details and is linked to roles such as Customer, Technician, Manager, or Admin.
- The **Technician** entity extends the user with skill set, availability, experience, and approval status.
- **Service Categories** group related services, while **Services** define individual offerings with pricing.
- A **Service Request (Booking)** captures customer service requests, service status, assigned technician, and schedule.
- **Technician Assignment** tracks which technician is assigned to which booking.
- **Invoice** and **Payment** entities handle billing and financial tracking.
- **Notification** logs system-generated alerts sent to users.

This design ensures clear separation of concerns and supports future scalability.

User Authentication & Authorization Flow:

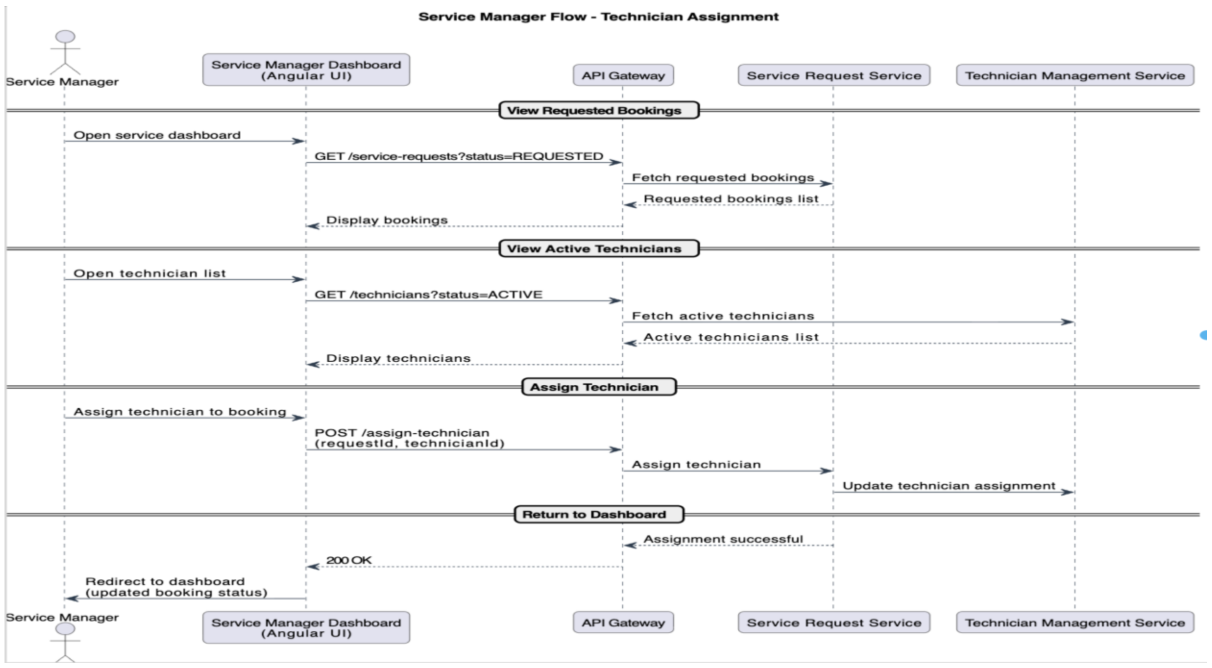


This diagram shows the **JWT-based authentication flow** for all users.

- The user submits login credentials through the Angular UI.
- The request is routed via the **API Gateway** to the Authentication Service.
- The Auth Service validates credentials against the database and verifies the password using BCrypt hashing.
- On successful authentication, a **JWT access token** is generated and returned to the client.
- The Angular frontend stores the token securely and attaches it to subsequent API requests.
- The API Gateway validates the JWT before forwarding requests to protected services.

This approach ensures **stateless, secure, and scalable authentication**.

Service Manager – Technician Assignment Flow:

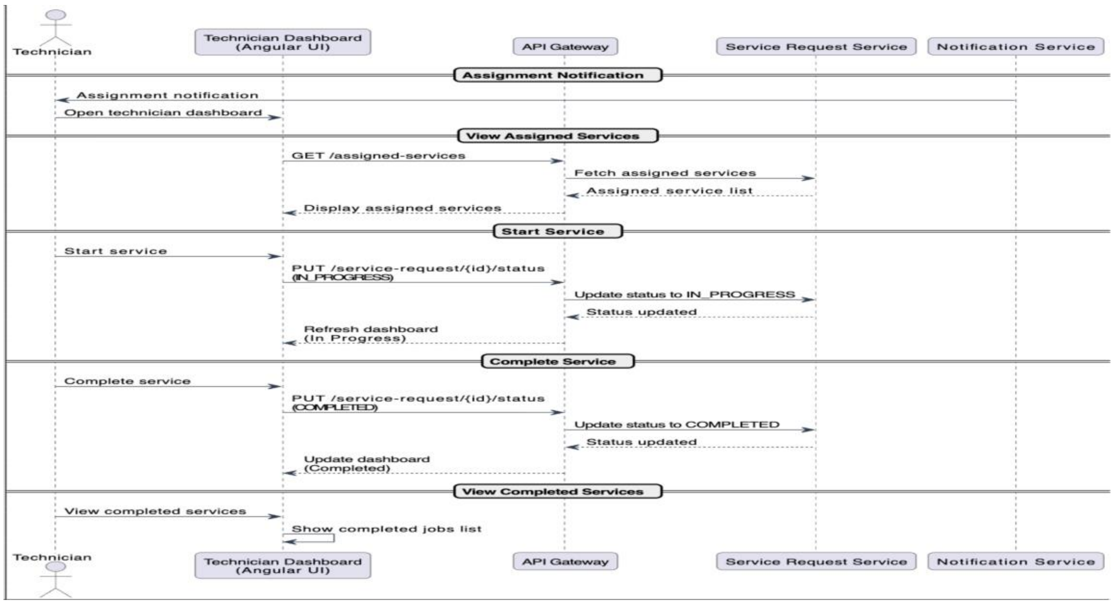


This diagram describes how a **Service Manager assigns technicians to bookings**.

- The Manager logs into the dashboard and views all **REQUESTED** service bookings.
- The system fetches active technicians based on availability.
- The Manager selects a technician and assigns them to a booking.
- The Booking Service updates the booking status to **ASSIGNED**.
- An assignment event is generated and sent to the Notification Service.
- The assigned technician receives an email notification.
- Then Service manager logs out successfully.
- The only thing service manager does is to assign a technician looking into the technicians in authdb.

This flow enforces business rules and ensures only eligible technicians are assigned.

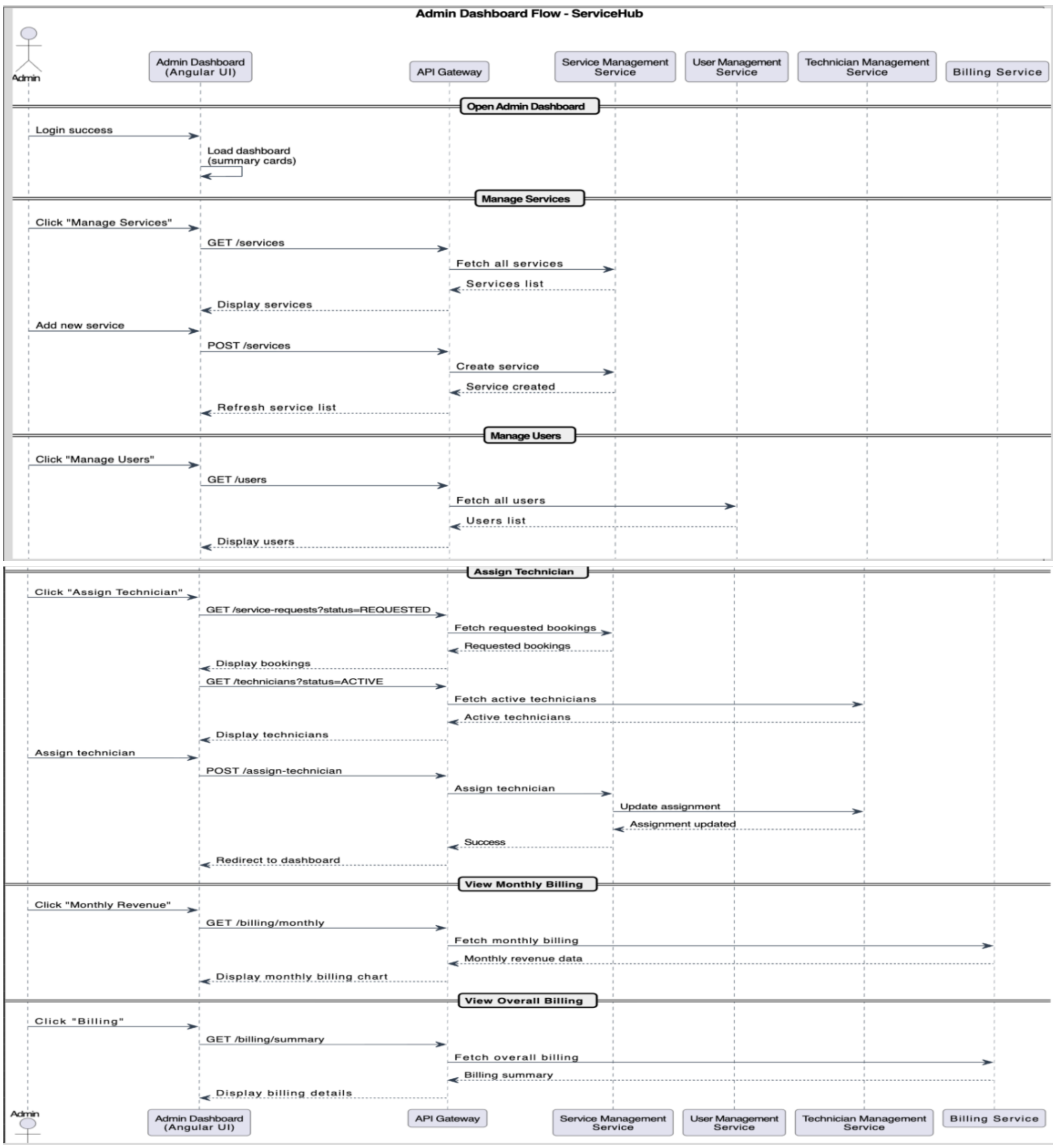
Technician Workflow Diagram:



This diagram illustrates the **end-to-end workflow for technicians**.

- Technicians receive assignment notifications.
- They view assigned services on their dashboard.
- They can see all the ones assigned, InProgress and completed on their dashboard
- Technicians update service status to **IN_PROGRESS** when work begins.
- Then redirects to the dashboard.
- After completing the service, the status is updated to **COMPLETED**.
- Redirects to the dashboard.
- The system records the completion and triggers billing and notifications.
- Then creates a notification that its done.

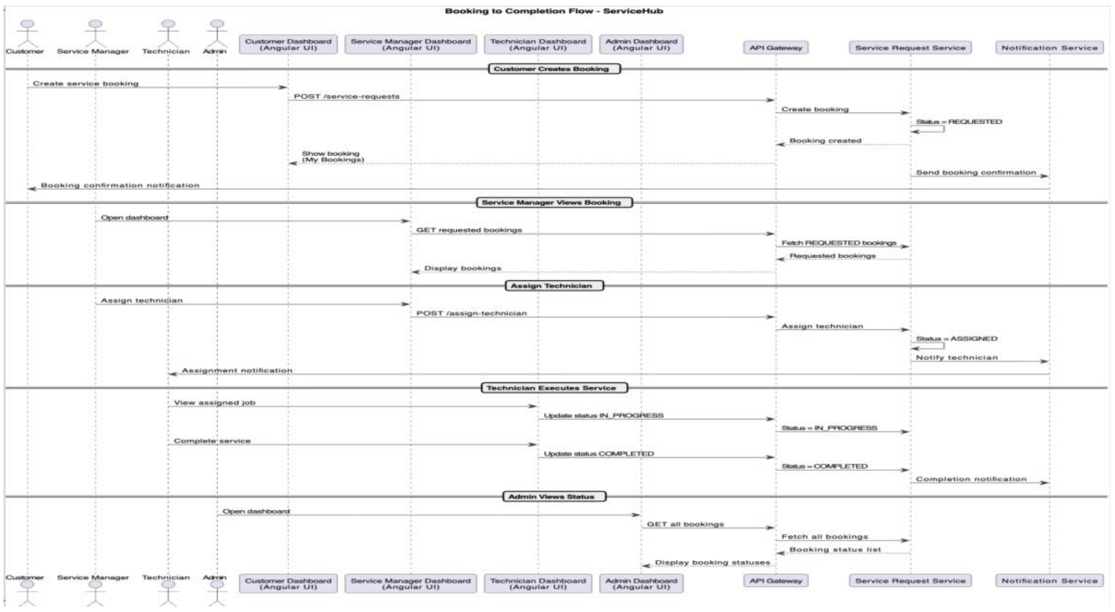
Admin Dashboard Flow:



- The Admin Dashboard serves as the **central control panel** of the Service Management System.
- After login, the Admin is authenticated using **JWT-based security** via the API Gateway.
- The dashboard displays **overall system statistics** such as total users, services, bookings, and revenue.
- Admin can **manage service catalog** by adding, updating, or removing services and categories.
- Admin can **view and monitor all users**, including customers, technicians, and managers.
- Admin has visibility into **technician assignments and booking statuses** across the system.

- The dashboard provides access to **billing summaries and monthly revenue reports**.
- All actions are protected using **role-based access control (RBAC)** to ensure secure administration.

Booking Lifecycle – From Creation to Completion:

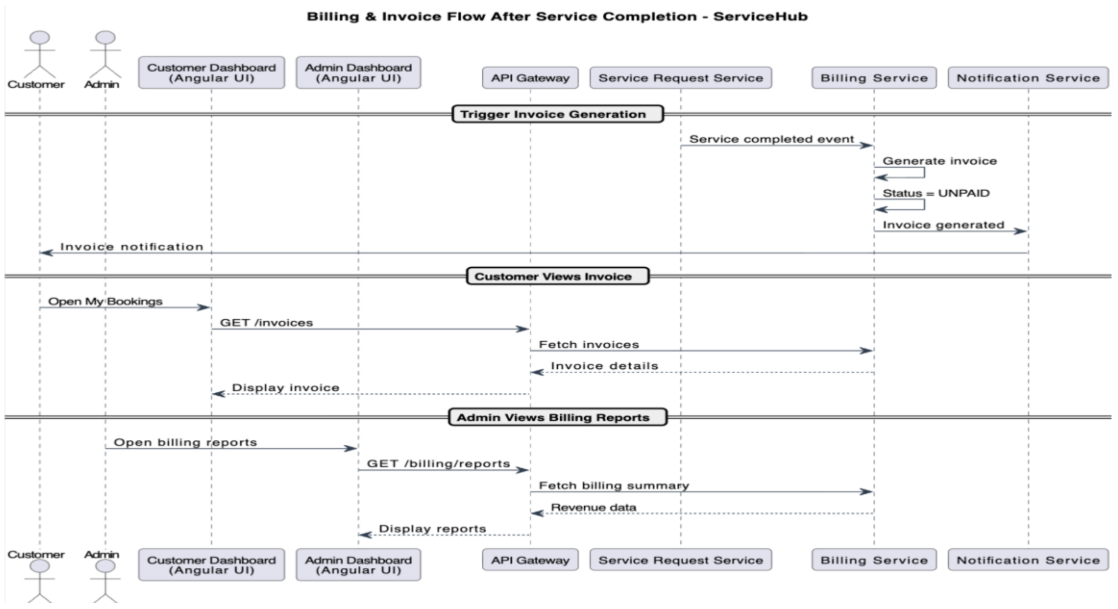


This diagram represents the **core business flow** of the application.

- Customer creates a booking request.
- Manager assigns a technician.
- Technician starts and completes the service.
- Booking transitions through statuses:
  REQUESTED → ASSIGNED → IN_PROGRESS → COMPLETED
- Notifications are sent at each major stage.

This structured lifecycle ensures transparency and traceability.

Billing & Invoice Generation Flow:



This diagram explains **automatic invoice generation** after service completion.

- Booking completion event triggers the Billing Service.
- Invoice is generated with tax and total calculation.
- Invoice is stored in the Billing database.
- Customer receives an invoice notification.
- Admin can view billing summaries and revenue reports.

This automation eliminates manual billing and improves accuracy.

9.TESTING & QUALITY ASSURANCE

Overview

The Service Management System adopts a **layered testing and quality assurance approach** to ensure reliability, security, and maintainability. Testing is integrated into the **CI/CD pipeline**, allowing early detection of defects and enforcing coding standards across all microservices and the frontend application.

Backend Testing Strategy

Unit Testing (JUnit 5 & Mockito)

Unit testing is performed at the **service layer** to validate core business logic in isolation.

JUnit 5 is used as the testing framework, while Mockito is used to mock dependencies such as repositories and Feign clients.

Key areas tested include:

- Booking lifecycle transitions
- Technician assignment validation
- Invoice calculation logic
- Role-based access conditions

This approach ensures fast execution and early detection of logical issues.

API & Controller Testing (Spring Boot Test)

Controller-level testing is implemented using **Spring Boot Test and MockMvc**.
These tests verify:

- Correct REST endpoint mappings
- Request validation and response structures
- Proper HTTP status codes (200, 400, 401, 403, 404)

The service layer is mocked to isolate and validate the web layer independently.

Repository Testing

Repository testing ensures correct interaction with MongoDB.
Using MongoDB test configurations, repository methods and custom queries are validated to confirm accurate data persistence and retrieval.

Frontend Testing (Angular)

Frontend testing ensures UI stability and correct client-side behavior.

- **Component Tests (Jasmine & Karma):**
  Validate component rendering, form validation, and user interactions.
- **Service Tests:**
  Angular services are tested using mocked HTTP responses to ensure correct API calls and error handling.

Code Coverage – JaCoCo

JaCoCo is integrated into the Maven build process to measure **test coverage** for backend services.

- Coverage reports are generated after test execution.
- Focus is on **Service and Controller layers**.
- A minimum coverage threshold is enforced for critical business logic.

This ensures that important functionality is adequately tested.

Static Code Analysis – SonarQube

SonarQube is used for **static code analysis** to maintain code quality.

It identifies:

- Code smells
- Bugs and vulnerabilities
- Code duplication
- Test coverage issues

A **quality gate** is configured to fail the build if critical issues or coverage violations are detected.

CI/CD Integration – Jenkins

Jenkins automates testing and quality checks as part of the CI/CD pipeline.

Pipeline Flow:

1. Code checkout from version control
2. Maven build and test execution
3. JaCoCo coverage generation
4. SonarQube analysis
5. Build failure on test or quality gate failure

This ensures that only high-quality, tested code progresses through the pipeline.

10.Docker Deployment

- Each microservice (Auth, Booking, Service Catalog, Billing, Notification) is containerized using **Docker**.
- The Angular frontend runs in a separate Docker container.
- **MongoDB** and **RabbitMQ** are deployed as independent containers.
- **Docker Compose** is used to run the entire system with a single command.
- Containerization ensures environment consistency and easy deployment.

11.Future Enhancements

- **Smart Technician Assignment**
  Automatically assign technicians based on availability, skills, and workload.
- **Advanced Reports & Analytics**
  Generate insights on bookings, revenue, and technician performance.
- **Mobile Application**
  Develop Android/iOS apps for customers and technicians.
- **GPS & Location Tracking**
  Show service locations on maps and enable route navigation for technicians.
- **Online Payment Integration**
  Enable secure digital payments and invoice settlement.