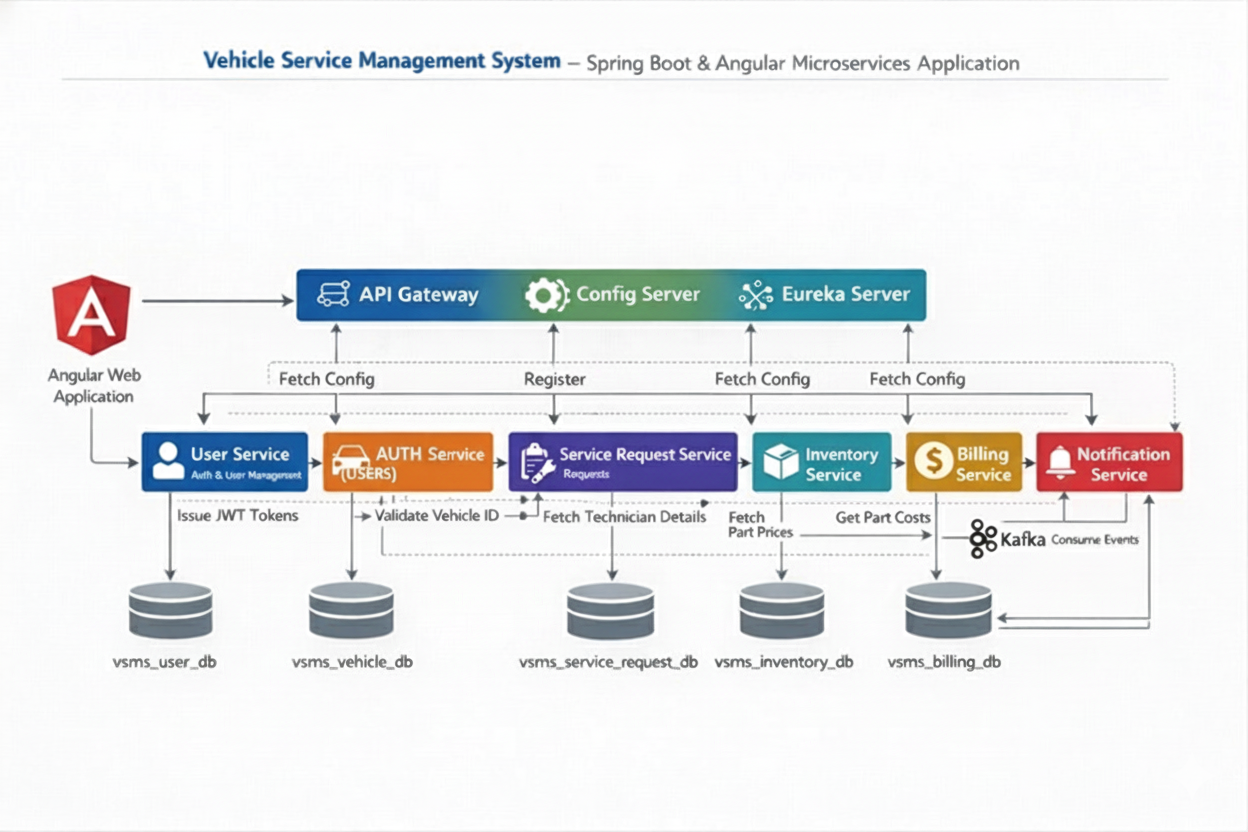
**SOFTWARE DESIGN DOCUMENT (SDD)**

**Vehicle Service Management System (VSMS)**



**1. DOCUMENT CONTROL**

|  |  |
| --- | --- |
| **Item** | **Details** |
| Project Name | Vehicle Service Management System |
| Version | 1.0 |
| Author | Srihari E. |
| Date | (Date) |
| Status | Final |

**2. PURPOSE OF THE DOCUMENT**

This document describes the **system architecture, design decisions, component structure, APIs, data models, and non-functional aspects** of the Vehicle Service Management System.

It is intended for:

* Developers
* Reviewers
* Interview discussions
* Maintenance & enhancement planning

**3. SYSTEM OVERVIEW**

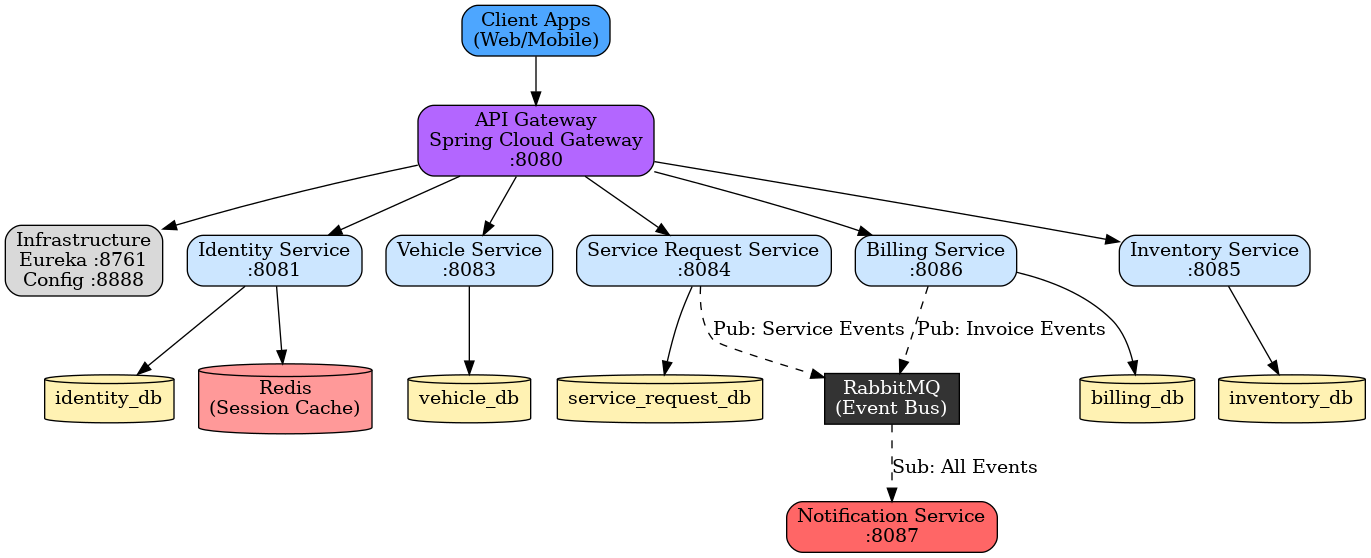
**Business Objective**

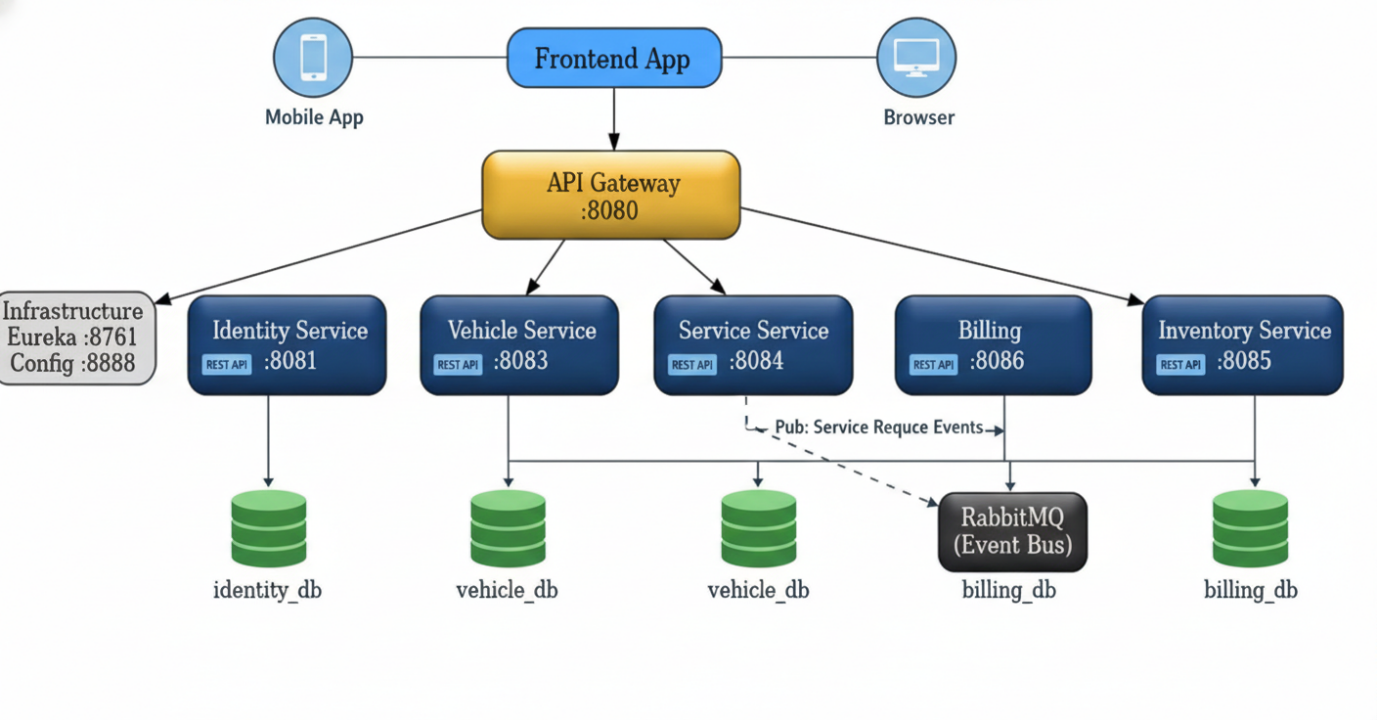
Provide a scalable, secure, and maintainable system to:

* Manage users (Customers, Technicians, Managers, Inventory Managers, Admins)
* Manage vehicles and their service history
* Book, assign, and track vehicle service requests
* Manage spare parts inventory and billing
* Support growth through microservices architecture

**High-Level Features**

* User registration & login (JWT authentication)
* Vehicle registration & management
* Service request booking & lifecycle tracking
* Technician assignment & workload management
* Invoice generation & payment tracking
* Role-based access (CUSTOMER / TECHNICIAN / MANAGER / INVENTORY\_MANAGER / ADMIN)

**4. ARCHITECTURE OVERVIEW (HLD)**



**4. ARCHITECTURE OVERVIEW**

**Architecture Style**

* Microservices Architecture
* REST-based communication
* Event-driven messaging (RabbitMQ)
* Containerized deployment

**Core Components**

1. Angular Frontend
2. API Gateway (Spring Cloud Gateway)
3. Identity Service (Auth / Users)
4. Vehicle Service
5. Service Request Service
6. Inventory Service
7. Billing Service
8. Notification Service
9. PostgreSQL (main data storage)
10. MongoDB (images / documents)
11. Redis (caching / sessions)
12. RabbitMQ (async event messaging)
13. Eureka Server (service discovery)
14. Config Server (central configuration)
15. CI/CD Pipeline

**5. TECHNOLOGY STACK**

**Backend**

* Java 17
* Spring Boot
* Spring Web
* Spring Data JPA (PostgreSQL)
* Spring Data MongoDB (GridFS)
* Spring Security + JWT
* Spring Cloud Gateway
* Spring Cloud Netflix Eureka
* Spring Cloud Config
* Spring Validation

**Frontend**

* Angular 17+
* Angular Material / PrimeNG
* RxJS

**Databases**

* PostgreSQL (primary business data)
* MongoDB (images / attachments)
* Redis (cache / sessions)

**Messaging**

* RabbitMQ (events and notifications)

**DevOps**

* Docker
* Docker Compose
* Jenkins
* SonarQube
* Nginx (reverse proxy)

**Testing**

* JUnit 5
* Mockito
* Spring Boot Test
* Testcontainers (optional)
* Jasmine / Karma
* Cypress / Playwright (E2E)

**6. MICROSERVICES DESIGN**

**6.1 Identity Service (:8081)**

**Responsibilities**  
• User registration & login  
• JWT authentication  
• Role & profile management

**APIs**  
POST /api/auth/register/customer  
POST /api/auth/login  
GET /api/customers/{id}

**Database**  
• identity\_db (PostgreSQL)  
• Redis (cache)

**6.2 Vehicle Service (:8083)**

**Responsibilities**  
• Vehicle registration  
• Vehicle CRUD  
• Ownership validation

**APIs**  
POST /api/vehicles  
GET /api/vehicles/{id}  
PUT /api/vehicles/{id}

**Database**  
• vehicle\_db (PostgreSQL)

**6.3 Service Request Service (:8084)**

**Responsibilities**  
• Service booking  
• Technician assignment  
• Status updates

**APIs**  
POST /api/service-requests  
GET /api/service-requests/{id}  
PUT /api/service-requests/{id}/assign

**Database**  
• service\_request\_db (PostgreSQL)  
• MongoDB (images)

**6.4 Inventory Service (:8085)**

**Responsibilities**  
• Parts catalog management  
• Stock updates  
• Request approval

**APIs**  
POST /api/parts  
GET /api/parts  
PUT /api/parts/{id}/stock

**Database**  
• inventory\_db (PostgreSQL)

**6.5 Billing Service (:8086)**

**Responsibilities**  
• Invoice generation  
• Payment tracking

**APIs**  
POST /api/invoices/generate/{serviceRequestId}  
GET /api/invoices/{id}  
PUT /api/invoices/{id}/payment

**Database**  
• billing\_db (PostgreSQL)

**6.6 Notification Service (:8087)**

**Responsibilities**  
• Email / SMS alerts  
• RabbitMQ event notifications

**APIs**  
POST /api/notifications/send  
GET /api/notifications/logs

**Database**  
• notification\_db (PostgreSQL)  
• RabbitMQ (events)

**7. DATA DESIGN (LLD)**

**User**

{

"id": 1,

"email": "john@example.com",

"role": "CUSTOMER",

"status": "ACTIVE"

}

**Vehicle**

{

"id": 1,

"customerId": 1,

"registrationNumber": "KA-01-AB-1234",

"make": "Toyota"

}

**Service Request**

{

"id": 1,

"customerId": 1,

"vehicleId": 1,

"status": "IN\_PROGRESS"

}

**Part**

{

"id": 1,

"name": "Engine Oil 5W30",

"unitPrice": 450,

"stockQuantity": 50

}

**Invoice**

{

"id": 1,

"serviceRequestId": 1,

"totalAmount": 2832,

"paymentStatus": "PENDING"

}

**Notification**

{

"id": 1,

"type": "EMAIL",

"message": "Service booked successfully"

}

**8. API DESIGN & VALIDATION**

* RESTful principles
* JSON request/response
* Bean Validation at DTO layer
* Service-level business rule enforcement
* Standard HTTP status codes

**9. ERROR HANDLING STRATEGY**

**Global Exception Handling**

* Centralized using @ControllerAdvice
* Standard error response format

{

"timestamp": "2025-01-01T10:00:00",

"status": 400,

"error": "Validation Error",

"message": "Price must be greater than zero"

}

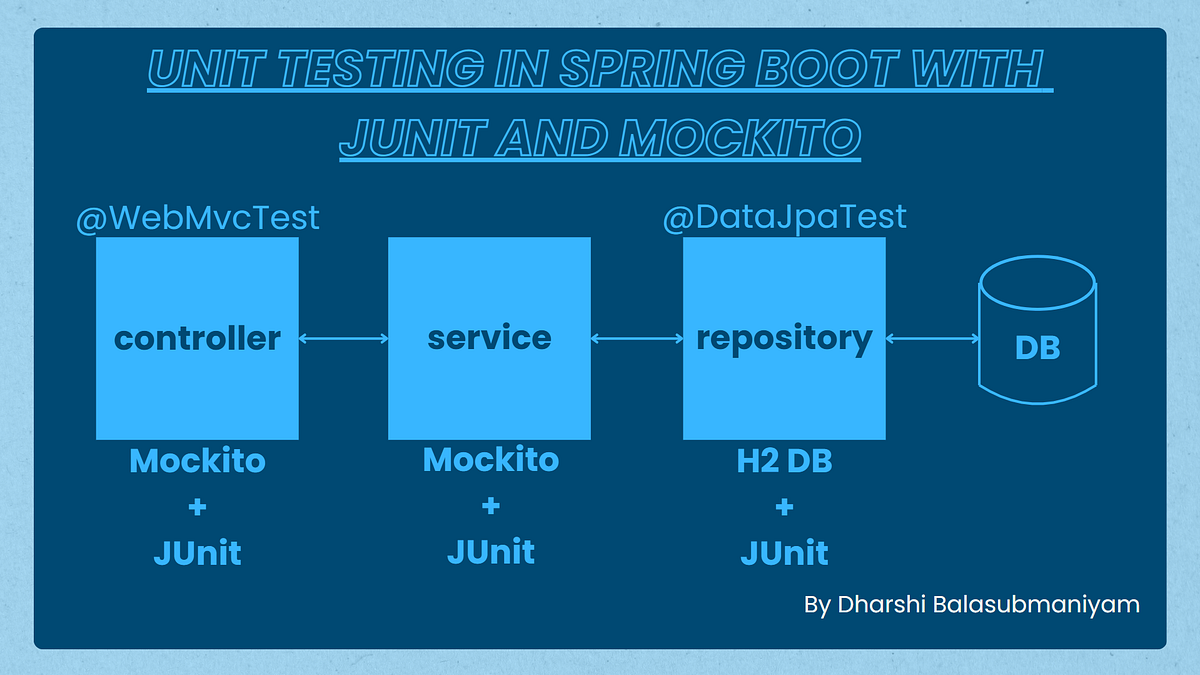
**10. SECURITY DESIGN**

* Password encryption (BCrypt)
* Role-based access control
* JWT authentication (optional enhancement)
* Secure API access

**11. NON-FUNCTIONAL REQUIREMENTS**

| **Area** | **Design Decision** |
| --- | --- |
| Scalability | Stateless services, Docker |
| Performance | Pagination, async calls |
| Availability | Independent services |
| Maintainability | POM-like layered backend |
| Security | Validation, encryption |
| Observability | Logging & monitoring |

**2. TESTING STRATEGY**



**Backend**

* Unit tests (Service layer)
* Controller tests (MockMvc)
* Minimum 90% coverage

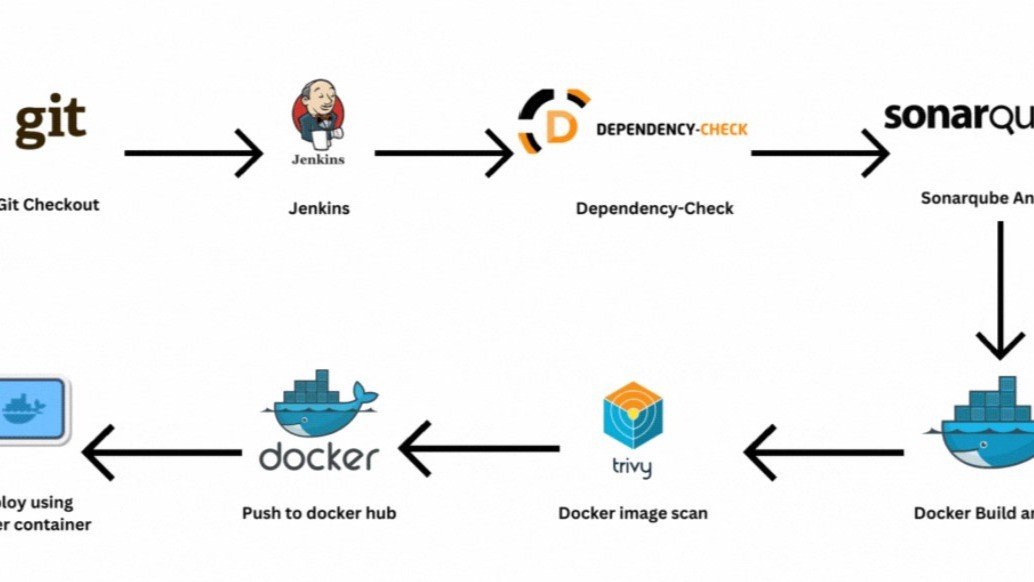
**Frontend**

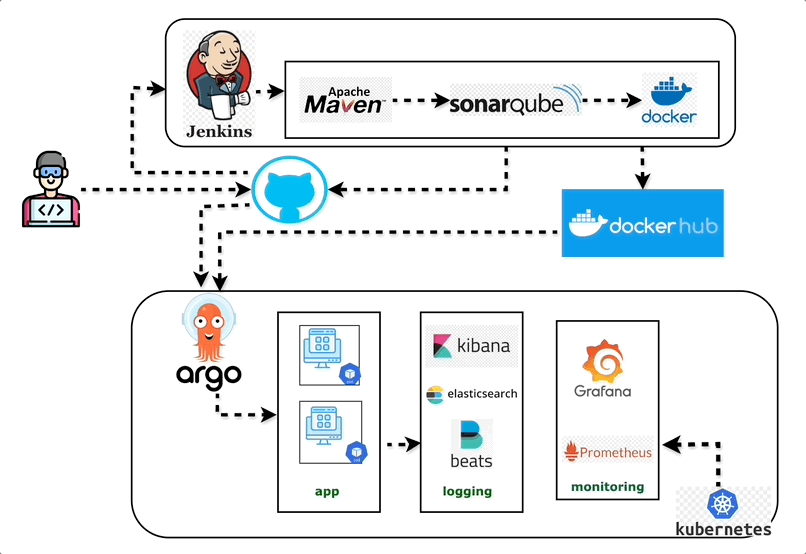
* Component tests
* Service tests

**Quality Gates**

* SonarQube enforced
* Build fails on violations

**13. CI/CD DESIGN**





**Pipeline Flow**

1. Git Commit
2. Jenkins Build
3. Unit Tests
4. SonarQube Scan
5. Quality Gate Check
6. Docker Build
7. Docker Compose Deploy

**14. DEPLOYMENT DESIGN**

* Docker image per microservice
* docker-compose for orchestration
* Environment-specific configs

**15. ASSUMPTIONS & CONSTRAINTS**

**Assumptions**

* Services communicate over REST
* MongoDB available
* Docker environment present

**Constraints**

* No distributed transactions
* Event-driven architecture out of scope

**16. FUTURE ENHANCEMENTS**

* Spring Cloud Gateway
* Kafka-based async communication
* Kubernetes deployment
* Centralized logging (ELK)

**17. CONCLUSION**

This design ensures:  
✔ Clean separation of concerns  
✔ Scalability & maintainability  
✔ Testability & CI/CD readiness  
✔ Interview-ready explanation

**SEQUENCE DIAGRAMS — VEHICLE SERVICE MANAGEMENT SYSTEM**

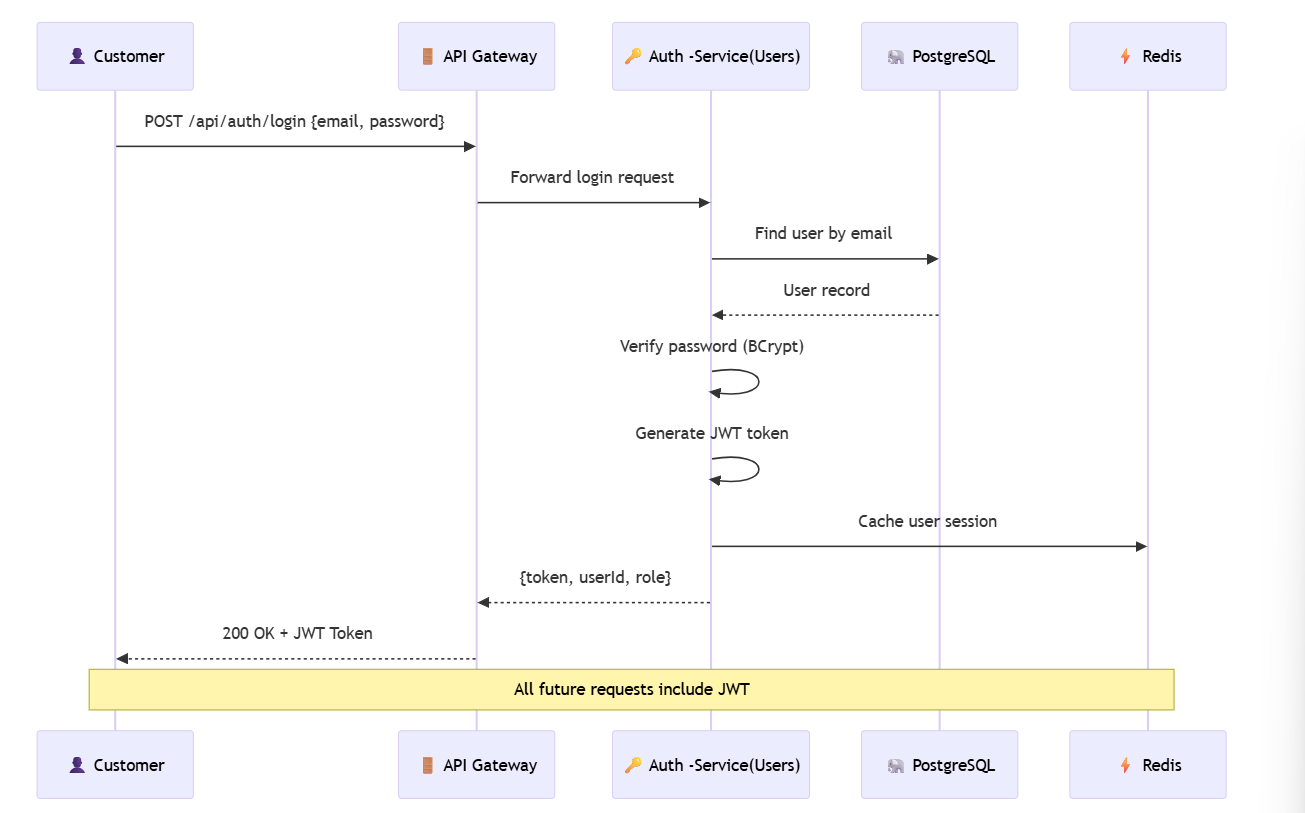
**1. User Authentication Flow**

**Scenario**

A registered user logs in using the Angular UI to access the system.

**Flow**

1. User enters email and password in Angular UI
2. Angular sends POST /api/auth/login to API Gateway
3. Gateway forwards request to Auth Service
4. Auth Service queries PostgreSQL for user by email
5. Password is verified using BCrypt
6. JWT token is generated with userId, role, expiry
7. Session is cached in Redis
8. Token is returned to Angular UI



**Participants**

|  |  |
| --- | --- |
| **Participant** | **Role** |
| Customer | End user logging in |
| API Gateway | Routes request, will validate JWT on future calls |
| Auth Service | Handles authentication logic |
| PostgreSQL | Stores user credentials |
| Redis | Caches session for performance |

**Key Design Points**

* Password hashing with BCrypt (never stored in plain text)
* JWT contains claims: userId, role, expiry
* Gateway validates JWT signature on every request
* Redis cache reduces database calls for session validation
* No direct database access from frontend

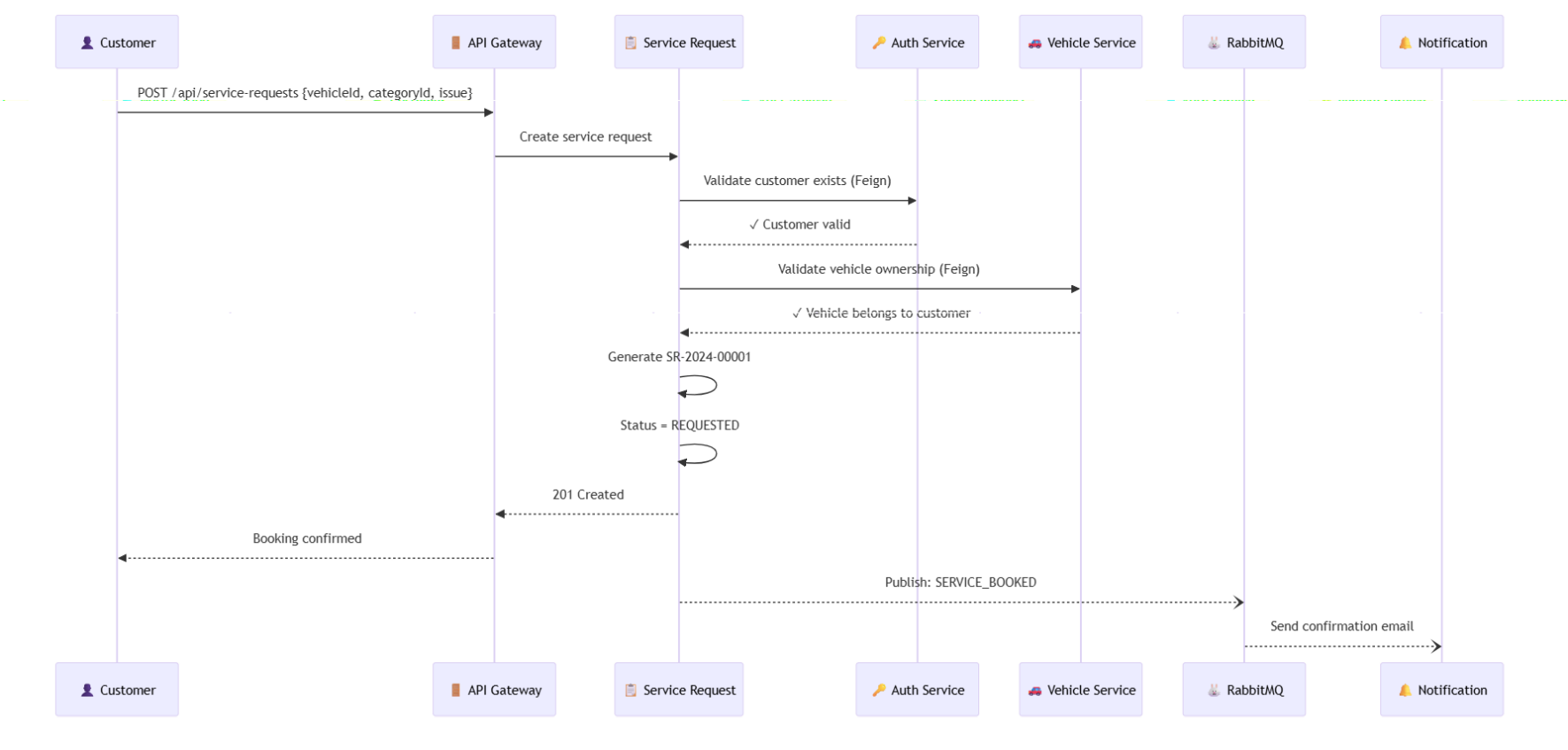
**2. Service Request Booking Flow— Sequence Diagram**

**Scenario**

A new user registers using the Angular UI.

**Flow**

1. User enters details in Angular UI
2. Angular sends POST /users/register to User Service
3. User Service validates request (email, password)
4. User Service checks email uniqueness in MongoDB
5. Password is encrypted
6. User is saved in MongoDB
7. Success response sent back to UI



**Participants**

|  |  |
| --- | --- |
| Participant | Role |
| Customer | Books the service |
| API Gateway | JWT validation + routing |
| Service Request | Core booking logic |
| Auth Service | Validates customer exists |
| Vehicle Service | Validates vehicle ownership |
| RabbitMQ | Async event messaging |
| Notification | Sends confirmation email |

**Key Design Points**

* Cross-service validation using OpenFeign clients
* Vehicle ownership verified before booking
* Request number auto-generated with pattern SR-YYYY-NNNNN
* Async notification doesn't block the response
* Event-driven architecture for loose coupling

**2. Technician Assignment Flow— Sequence Diagram**

**Scenario**

A manager assigns an available technician to a pending service request.

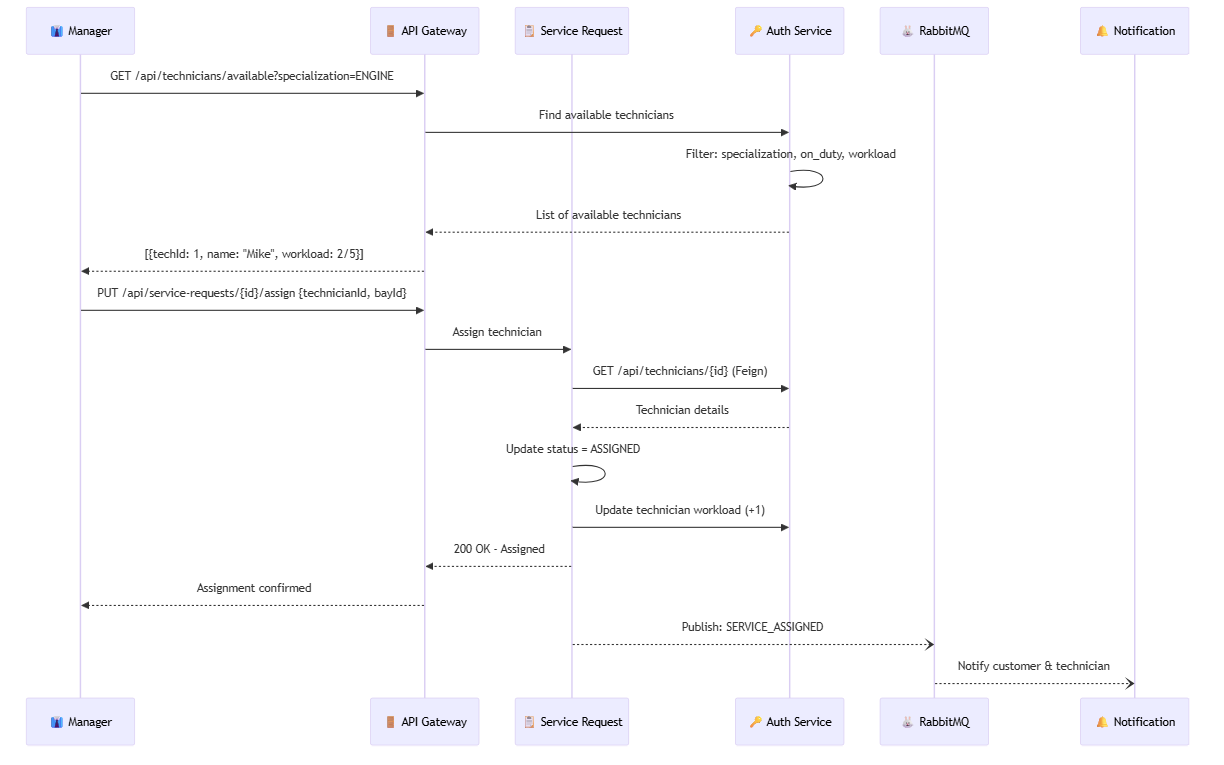
**Flow**

1. Manager views pending service requests in Angular UI
2. Manager clicks "Assign" on a request
3. Angular fetches available technicians filtered by specialization
4. Manager selects technician and service bay
5. Angular sends PUT /api/service-requests/{id}/assign to Gateway
6. Service Request calls Auth Service to get technician details
7. Service request is updated with technicianId, bayId, status=ASSIGNED
8. Auth Service updates technician workload (+1)
9. SERVICE\_ASSIGNED event published to RabbitMQ
10. Notification Service notifies both customer and technician

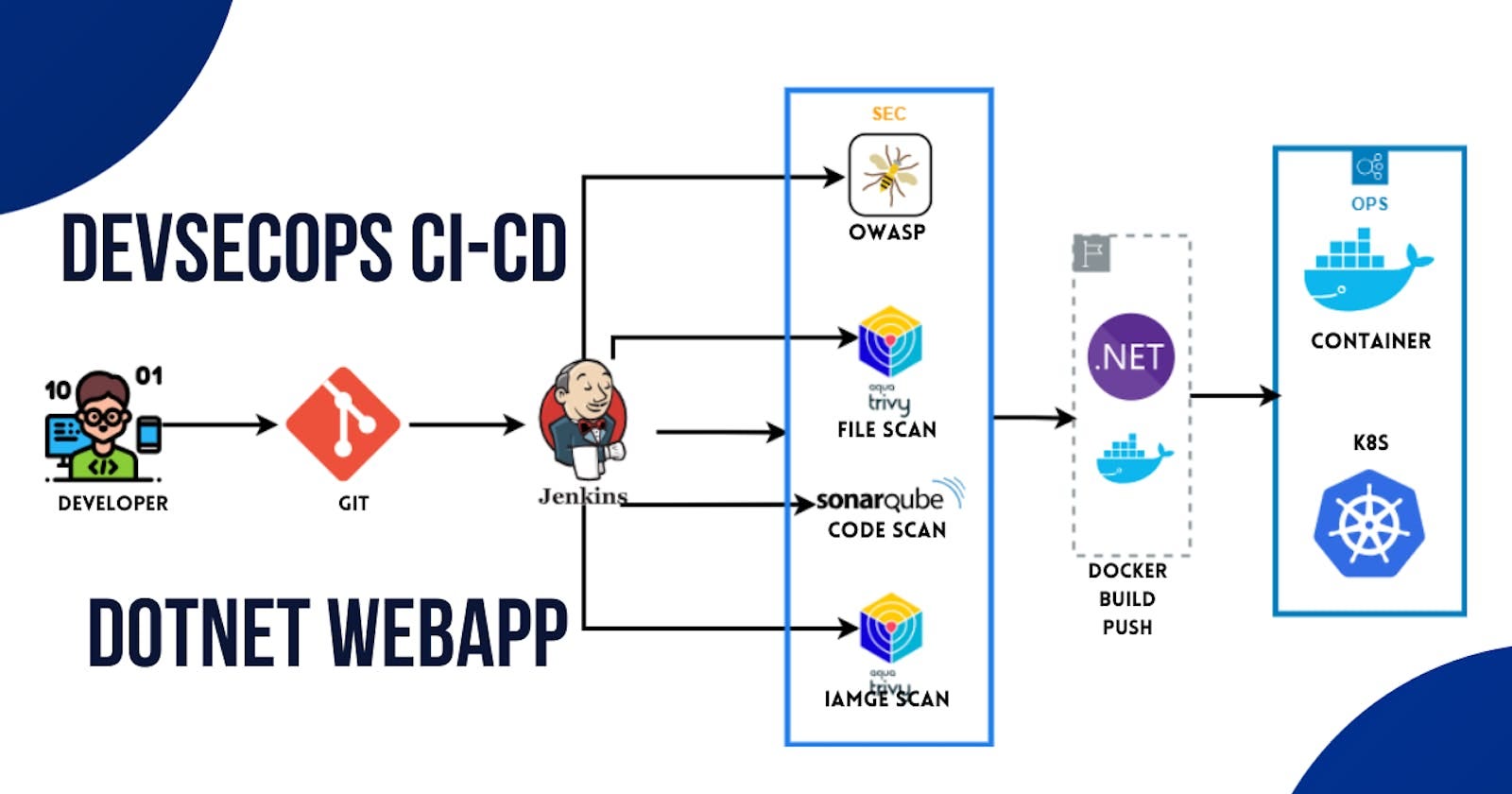
**Participants**

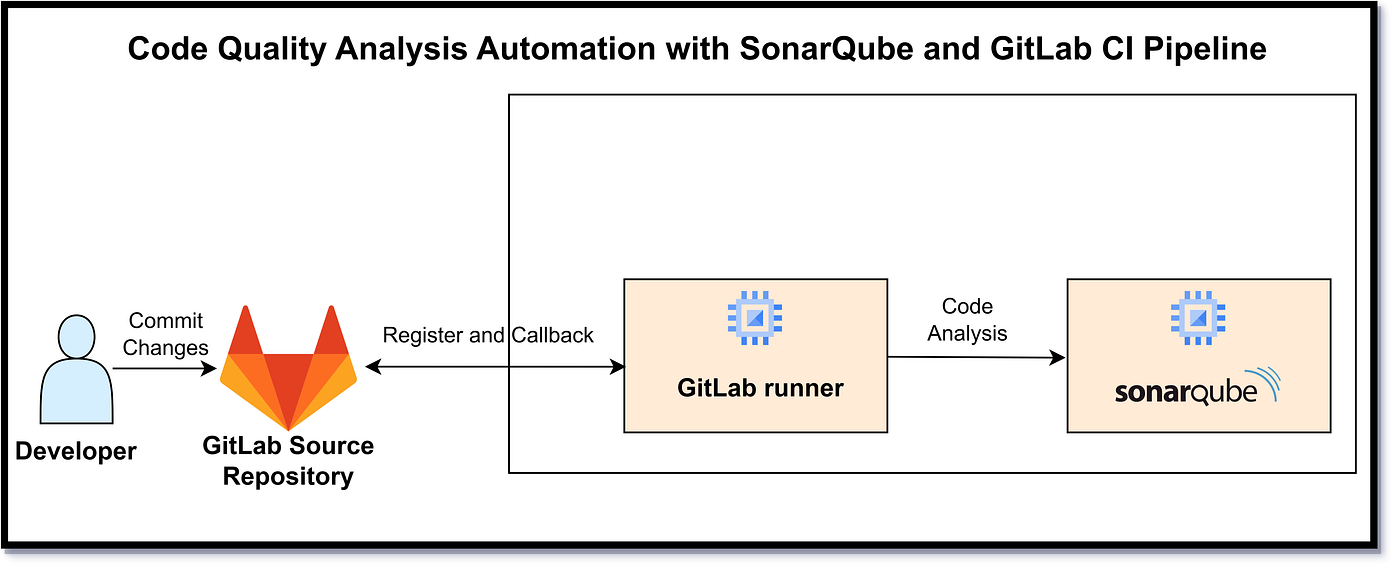
|  |  |
| --- | --- |
| **Participant** | **Role** |
| Manager | Assigns technician to service |
| API Gateway | JWT validation + routing |
| Service Request | Assignment logic |
| Auth Service | Manages technician data & workload |
| RabbitMQ | Async event messaging |
| Notification | Notifies customer & technician |

**Key Design Points**

* Technicians filtered by specialization matching service category
* Only on-duty technicians with available capacity shown
* Workload management prevents overloading technicians
* Service bay allocation tracked
* Both customer and technician notified asynchronously

**4. CI/CD Pipeline — Sequence Diagram**





**Scenario**

Developer pushes code to Git repository.

**Flow**

1. Developer pushes code
2. Jenkins pipeline triggered
3. Jenkins runs unit tests
4. Jenkins runs SonarQube scan
5. Quality gate checked
6. Docker images built
7. Docker Compose deploys services

**Participants**

* Developer
* Git
* Jenkins
* SonarQube
* Docker

**5. Error Handling — Sequence Diagram**

**Scenario**

Invalid request sent to backend.

**Flow**

1. Angular sends invalid request
2. Controller validation fails
3. Global exception handler triggered
4. Standard error response returned

**Key Point**

* Consistent error structure across services

**HOW TO EXPLAIN IN INTERVIEWS**

“We use sequence diagrams to show runtime behavior.  
Each diagram highlights service boundaries, validation points, and data ownership, ensuring clean microservices communication.”

**----- upto here ------**

**SEQUENCE → CODE CLASS MAPPING**

**1) User Registration**

**Sequence Steps**

1. User submits registration form
2. API receives request
3. Validate input & business rules
4. Encrypt password
5. Persist user
6. Return response

**Code Mapping (User Service)**

| **Step** | **Layer** | **Class** | **Responsibility** |
| --- | --- | --- | --- |
| 1 | Angular | RegisterComponent | Collect form data |
| 2 | Angular | AuthService | POST /users/register |
| 3 | API | UserController | Request mapping |
| 4 | DTO | UserRegisterRequest | Bean validation |
| 5 | Service | UserService | Uniqueness checks |
| 6 | Security | PasswordEncoderConfig | BCrypt encryption |
| 7 | Repo | UserRepository | Save user |
| 8 | API | UserController | Return response |

**Key Methods**

* UserController.register(UserRegisterRequest)
* UserService.registerUser(...)
* UserRepository.existsByEmail(...)

**2) User Login**

**Sequence Steps**

1. Submit credentials
2. Fetch user
3. Verify password
4. Generate token (optional)
5. Return auth response

**Code Mapping**

| **Step** | **Layer** | **Class** | **Responsibility** |
| --- | --- | --- | --- |
| 1 | Angular | LoginComponent | Capture credentials |
| 2 | Angular | AuthService | POST /users/login |
| 3 | API | AuthController | Handle login |
| 4 | Repo | UserRepository | Find by email |
| 5 | Service | AuthService | Password match |
| 6 | Security | JwtTokenProvider | Create JWT |
| 7 | API | AuthController | Return token |

**Key Methods**

* AuthController.login(LoginRequest)
* AuthService.authenticate(...)

**3) Product Listing**

**Sequence Steps**

1. UI requests products
2. Fetch from DB
3. Return list

**Code Mapping (Product Service)**

| **Step** | **Layer** | **Class** | **Responsibility** |
| --- | --- | --- | --- |
| 1 | Angular | ProductListComponent | Load products |
| 2 | Angular | ProductApiService | GET /products |
| 3 | API | ProductController | Handle request |
| 4 | Service | ProductService | Business logic |
| 5 | Repo | ProductRepository | Query MongoDB |

**Key Methods**

* ProductController.getAllProducts()
* ProductService.findAll()

**4) Order Placement (Critical Flow)**

**Sequence Steps**

1. Place order
2. Validate request
3. Check stock
4. Reduce inventory
5. Create order
6. Persist & respond

**Code Mapping (Order + Product Services)**

| **Step** | **Service** | **Class** | **Responsibility** |
| --- | --- | --- | --- |
| 1 | Angular | CheckoutComponent | Submit order |
| 2 | Angular | OrderApiService | POST /orders |
| 3 | Order API | OrderController | Receive order |
| 4 | DTO | OrderRequest | Validate payload |
| 5 | Order Svc | OrderService | Orchestrate flow |
| 6 | Order Svc | ProductClient | Call Product Service |
| 7 | Product API | ProductController | Validate stock |
| 8 | Product Svc | InventoryService | Deduct quantity |
| 9 | Order Repo | OrderRepository | Save order |
| 10 | Order API | OrderController | Return result |

**Key Methods**

* OrderService.placeOrder(OrderRequest)
* ProductClient.checkAndReserveStock(...)
* InventoryService.reduceStock(...)

**5) Order Status Update (Admin)**

**Sequence Steps**

1. Admin updates status
2. Validate role
3. Validate transition
4. Update order

**Code Mapping**

| **Step** | **Layer** | **Class** | **Responsibility** |
| --- | --- | --- | --- |
| 1 | Angular | AdminOrderComponent | Update status |
| 2 | Angular | OrderApiService | PUT /orders/{id}/status |
| 3 | Security | JwtAuthFilter | Role validation |
| 4 | API | OrderController | Accept request |
| 5 | Service | OrderService | Validate transition |
| 6 | Repo | OrderRepository | Persist status |

**Key Methods**

* OrderService.updateStatus(orderId, status)
* OrderStatusValidator.isValidTransition(...)

**6) Global Validation & Error Handling**

**Sequence Steps**

1. Invalid input
2. Validation fails
3. Standard error response

**Code Mapping**

| **Step** | **Layer** | **Class** | **Responsibility** |
| --- | --- | --- | --- |
| 1 | DTO | @Valid annotations | Input validation |
| 2 | Framework | MethodArgumentNotValidException | Triggered |
| 3 | API | GlobalExceptionHandler | Build error response |

**Key Classes**

* GlobalExceptionHandler
* ApiErrorResponse

**7) CI/CD with SonarQube**

**Sequence Steps**

1. Git push
2. Build & tests
3. Sonar scan
4. Quality gate
5. Docker build
6. Deploy

**Code/Config Mapping**

| **Step** | **Tool** | **File/Class** |
| --- | --- | --- |
| 1 | Git | Repository |
| 2 | Jenkins | Jenkinsfile |
| 3 | Maven | pom.xml |
| 4 | SonarQube | sonar-project.properties |
| 5 | Docker | Dockerfile |
| 6 | Compose | docker-compose.yml |

**TRACEABILITY (WHY THIS MATTERS)**

* **Sequence step** → **Controller** → **Service** → **Repository**
* Every **business rule** is enforced **either in DTO validation or service logic**
* Easy to explain **end-to-end flow** in interviews

**How to explain succinctly**

“Each sequence diagram step maps directly to a controller endpoint, a service orchestration method, and a repository call. Validation is enforced at DTO and service layers, and failures are handled centrally.”

**CODE SKELETON — PACKAGES & CLASSES**

**1. USER SERVICE (user-service)**

user-service

└─ src/main/java/com/example/user

├─ UserServiceApplication.java

│

├─ controller

│ └─ UserController.java

│

├─ service

│ ├─ UserService.java

│ └─ UserServiceImpl.java

│

├─ repository

│ └─ UserRepository.java

│

├─ model

│ └─ User.java

│

├─ dto

│ ├─ UserRegisterRequest.java

│ ├─ LoginRequest.java

│ └─ UserResponse.java

│

├─ exception

│ ├─ UserNotFoundException.java

│ ├─ DuplicateUserException.java

│ └─ GlobalExceptionHandler.java

│

└─ config

└─ SecurityConfig.java

**Key Responsibilities**

* UserController → API layer
* UserService → Business rules
* UserRepository → MongoDB access
* SecurityConfig → Password encoding / JWT (optional)

**2. PRODUCT SERVICE (product-service)**

product-service

└─ src/main/java/com/example/product

├─ ProductServiceApplication.java

│

├─ controller

│ └─ ProductController.java

│

├─ service

│ ├─ ProductService.java

│ ├─ ProductServiceImpl.java

│ └─ InventoryService.java

│

├─ repository

│ └─ ProductRepository.java

│

├─ model

│ └─ Product.java

│

├─ dto

│ ├─ ProductRequest.java

│ └─ ProductResponse.java

│

└─ exception

├─ ProductNotFoundException.java

└─ GlobalExceptionHandler.java

**Key Responsibilities**

* InventoryService → Stock validation & update
* ProductServiceImpl → Core business logic

**3. ORDER SERVICE (order-service)**

order-service

└─ src/main/java/com/example/order

├─ OrderServiceApplication.java

│

├─ controller

│ └─ OrderController.java

│

├─ service

│ ├─ OrderService.java

│ ├─ OrderServiceImpl.java

│ └─ OrderStatusValidator.java

│

├─ client

│ ├─ ProductClient.java

│ └─ UserClient.java

│

├─ repository

│ └─ OrderRepository.java

│

├─ model

│ ├─ Order.java

│ └─ OrderItem.java

│

├─ dto

│ ├─ OrderRequest.java

│ ├─ OrderItemRequest.java

│ └─ OrderResponse.java

│

└─ exception

├─ OrderNotFoundException.java

└─ GlobalExceptionHandler.java

**Key Responsibilities**

* ProductClient → Calls Product Service
* OrderStatusValidator → Valid status transitions
* OrderServiceImpl → Orchestrates order flow

**4. COMMON / SHARED CONCEPTS (Optional)**

common

└─ src/main/java/com/example/common

├─ exception

│ └─ ApiErrorResponse.java

│

├─ util

│ └─ Constants.java

│

└─ config

└─ SwaggerConfig.java

**5. ANGULAR FRONTEND (angular-ui)**

angular-ui

└─ src/app

├─ core

│ ├─ services

│ │ ├─ auth.service.ts

│ │ ├─ product.service.ts

│ │ └─ order.service.ts

│ │

│ └─ guards

│ └─ auth.guard.ts

│

├─ modules

│ ├─ auth

│ │ ├─ login.component.ts

│ │ └─ register.component.ts

│ │

│ ├─ product

│ │ └─ product-list.component.ts

│ │

│ └─ order

│ ├─ checkout.component.ts

│ └─ order-history.component.ts

│

├─ shared

│ └─ models

│ ├─ user.model.ts

│ ├─ product.model.ts

│ └─ order.model.ts

│

└─ app.module.ts

**6. TEST STRUCTURE (IMPORTANT)**

src/test/java

└─ com/example

├─ controller

│ └─ UserControllerTest.java

├─ service

│ ├─ UserServiceTest.java

│ └─ OrderServiceTest.java

└─ repository

└─ ProductRepositoryTest.java

**7. DEVOPS & CI/CD FILES**

capstone-project

├─ Jenkinsfile

├─ docker-compose.yml

├─ user-service/Dockerfile

├─ product-service/Dockerfile

├─ order-service/Dockerfile

├─ angular-ui/Dockerfile

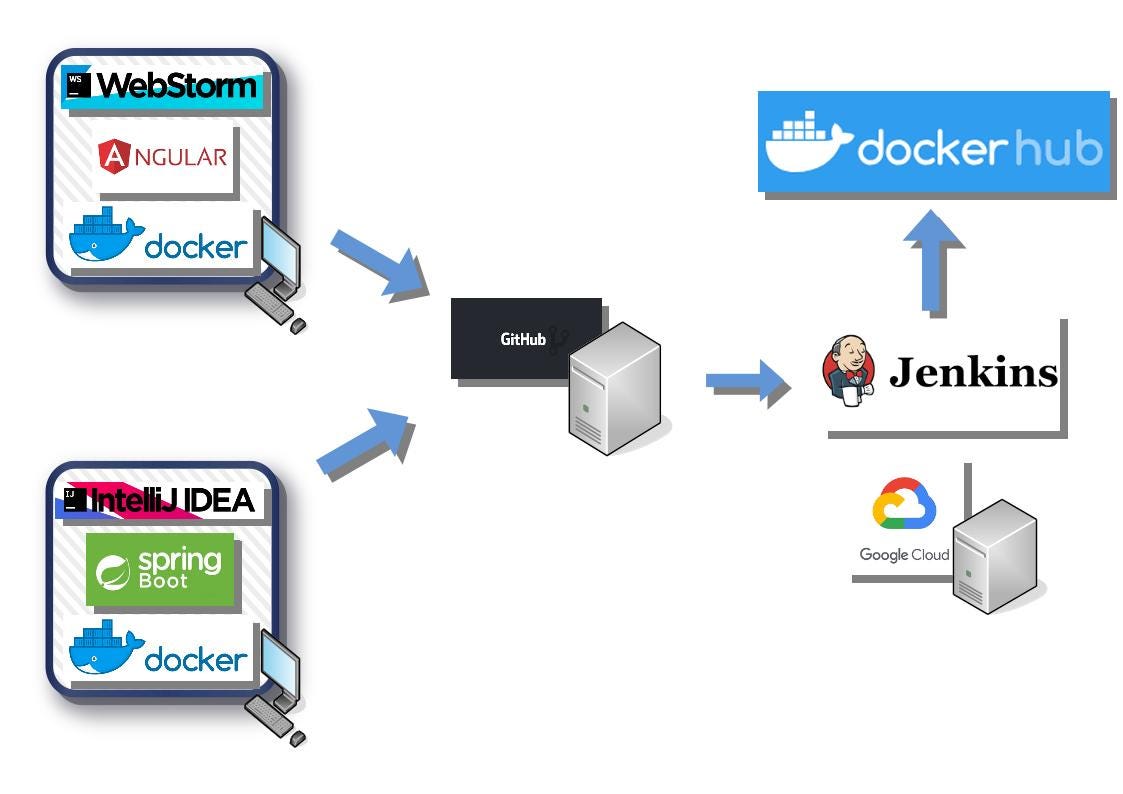
└─ README.md

**HOW TO EXPLAIN THIS IN INTERVIEWS**

“Each microservice follows a layered architecture: Controller → Service → Repository.  
Clients handle inter-service communication.  
DTOs isolate API contracts.  
Business rules live in the service layer.  
This makes the system scalable, testable, and maintainable.”

**STARTER REPO**





**1. REPOSITORY STRUCTURE (ROOT)**

smart-order-management/

│

├─ user-service/

├─ product-service/

├─ order-service/

├─ angular-ui/

│

├─ common/

│

├─ docker-compose.yml

├─ Jenkinsfile

├─ .gitignore

├─ README.md

You can create this repo **as-is** in GitHub.

**2. USER SERVICE (Spring Boot)**

user-service/

├─ src/main/java/com/example/user

│ ├─ UserServiceApplication.java

│ │

│ ├─ controller

│ │ └─ UserController.java

│ │

│ ├─ service

│ │ ├─ UserService.java

│ │ └─ UserServiceImpl.java

│ │

│ ├─ repository

│ │ └─ UserRepository.java

│ │

│ ├─ model

│ │ └─ User.java

│ │

│ ├─ dto

│ │ ├─ UserRegisterRequest.java

│ │ ├─ LoginRequest.java

│ │ └─ UserResponse.java

│ │

│ ├─ exception

│ │ ├─ DuplicateUserException.java

│ │ └─ GlobalExceptionHandler.java

│ │

│ └─ config

│ └─ SecurityConfig.java

│

├─ src/test/java/com/example/user

│ ├─ controller/UserControllerTest.java

│ └─ service/UserServiceTest.java

│

├─ Dockerfile

├─ pom.xml

└─ application.yml

**3. PRODUCT SERVICE**

product-service/

├─ src/main/java/com/example/product

│ ├─ ProductServiceApplication.java

│ ├─ controller/ProductController.java

│ ├─ service

│ │ ├─ ProductService.java

│ │ ├─ ProductServiceImpl.java

│ │ └─ InventoryService.java

│ ├─ repository/ProductRepository.java

│ ├─ model/Product.java

│ ├─ dto/ProductRequest.java

│ └─ exception/GlobalExceptionHandler.java

│

├─ src/test/java/com/example/product

│ └─ service/ProductServiceTest.java

│

├─ Dockerfile

├─ pom.xml

└─ application.yml

**4. ORDER SERVICE**

order-service/

├─ src/main/java/com/example/order

│ ├─ OrderServiceApplication.java

│ ├─ controller/OrderController.java

│ ├─ service

│ │ ├─ OrderService.java

│ │ ├─ OrderServiceImpl.java

│ │ └─ OrderStatusValidator.java

│ ├─ client

│ │ ├─ ProductClient.java

│ │ └─ UserClient.java

│ ├─ repository/OrderRepository.java

│ ├─ model

│ │ ├─ Order.java

│ │ └─ OrderItem.java

│ └─ dto/OrderRequest.java

│

├─ src/test/java/com/example/order

│ └─ service/OrderServiceTest.java

│

├─ Dockerfile

├─ pom.xml

└─ application.yml

**5. ANGULAR FRONTEND**

angular-ui/

├─ src/app

│ ├─ core

│ │ ├─ services

│ │ │ ├─ auth.service.ts

│ │ │ ├─ product.service.ts

│ │ │ └─ order.service.ts

│ │ └─ guards/auth.guard.ts

│ │

│ ├─ modules

│ │ ├─ auth

│ │ │ ├─ login.component.ts

│ │ │ └─ register.component.ts

│ │ ├─ product/product-list.component.ts

│ │ └─ order/checkout.component.ts

│ │

│ └─ shared/models

│ ├─ user.model.ts

│ ├─ product.model.ts

│ └─ order.model.ts

│

├─ Dockerfile

└─ angular.json

**6. DOCKER COMPOSE**

version: '3.8'

services:

mongodb:

image: mongo

ports:

- "27017:27017"

user-service:

build: ./user-service

ports:

- "8081:8081"

depends\_on:

- mongodb

product-service:

build: ./product-service

ports:

- "8082:8082"

depends\_on:

- mongodb

order-service:

build: ./order-service

ports:

- "8083:8083"

depends\_on:

- mongodb

angular-ui:

build: ./angular-ui

ports:

- "4200:80"

**7. JENKINSFILE (CI/CD + SONARQUBE)**

pipeline {

agent any

stages {

stage('Checkout') {

steps { git 'https://github.com/your-org/smart-order-management.git' }

}

stage('Build & Test') {

steps { sh 'mvn clean test' }

}

stage('SonarQube Scan') {

steps {

withSonarQubeEnv('SonarQube') {

sh 'mvn sonar:sonar'

}

}

}

stage('Quality Gate') {

steps {

timeout(time: 2, unit: 'MINUTES') {

waitForQualityGate abortPipeline: true

}

}

}

stage('Docker Build & Deploy') {

steps {

sh 'docker-compose up -d --build'

}

}

}

}

**8. README.md (MINIMAL TEMPLATE)**

# Smart Order Management System

## Tech Stack

- Spring Boot Microservices

- Angular

- MongoDB

- Docker

- Jenkins + SonarQube

## Run Locally

docker-compose up -d

## Services

- User Service: 8081

- Product Service: 8082

- Order Service: 8083

- UI: http://localhost:4200

Map **design → code structure**

Tell me what you want to generate next.