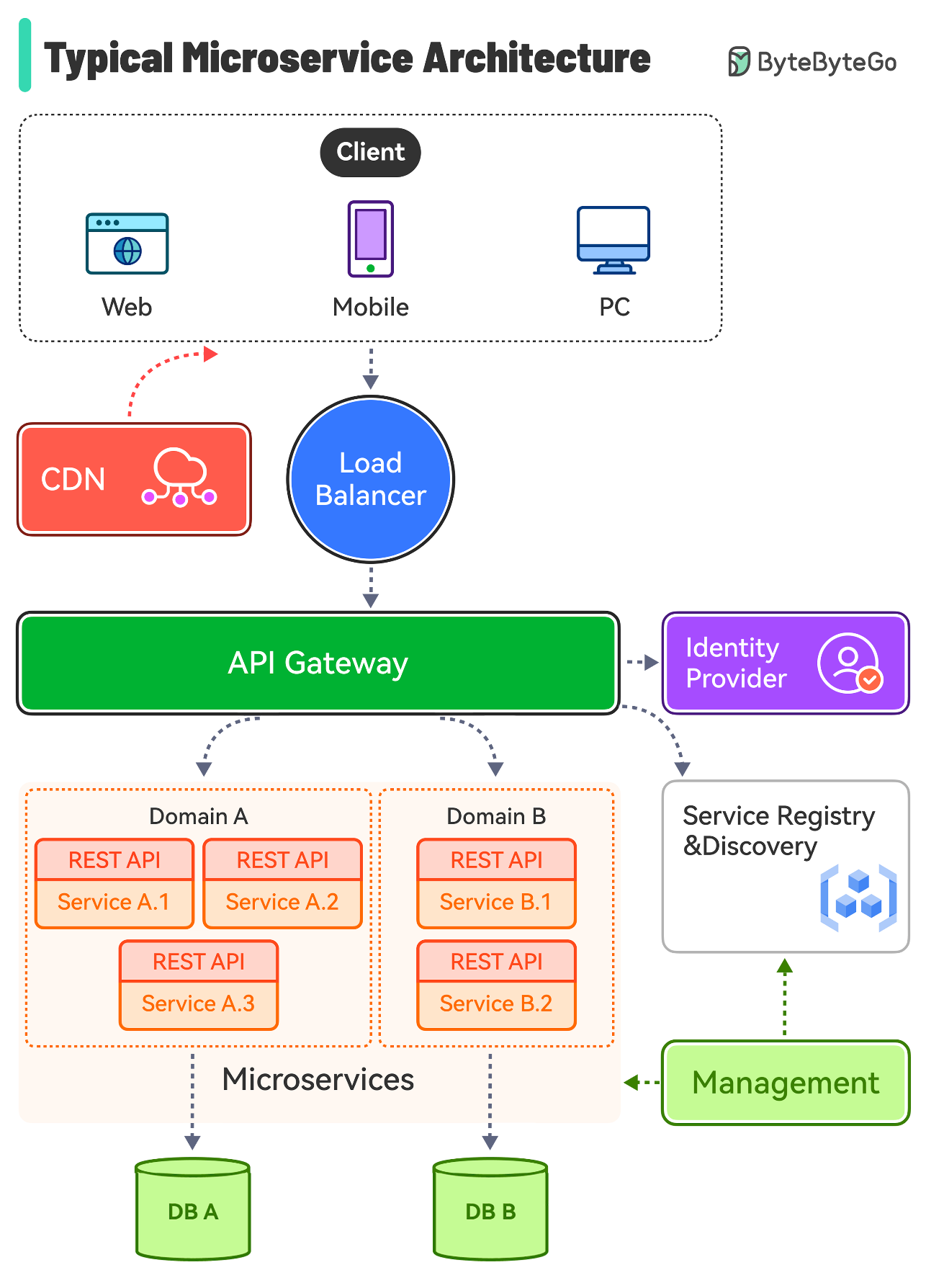
**SOFTWARE DESIGN DOCUMENT (SDD)**

**Utility Billing System**



**1. DOCUMENT CONTROL**

| **Item** | **Details** |
| --- | --- |
| Project Name | Utility Billing System |
| Version | 1.0 |
| Author | Bhavana Yeluri |
| Date | 29/12/2025 |
| 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 Smart Order Management System.

It is intended for:

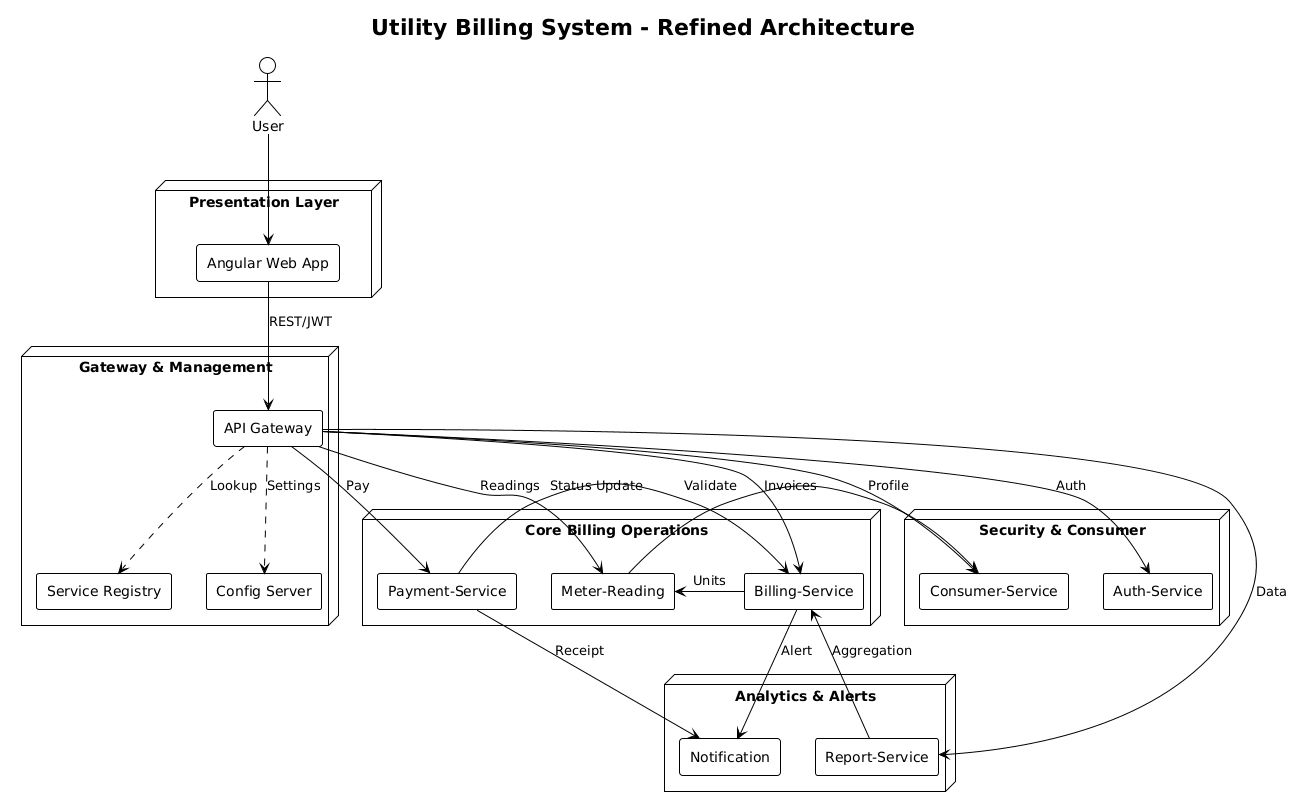
* Developers
* Reviewers
* Interview discussions
* Maintenance & enhancement planning

**3. SYSTEM OVERVIEW**

**Business Objective**

* **Centralized Utility Management** – Provide a single platform to manage multiple utilities, consumers, billing, and payments.
* **Automated and Accurate Billing** – Generate timely and error-free bills based on consumption data.
* **Secure Role-Based Access** – Enforce JWT-based authentication and controlled access for different user roles.
* **Flexible Payment Processing** – Support both online and offline payment methods for user convenience.
* **Real-Time Notifications and Transparency** – Notify users about bills, payments, and reminders to improve transparency.
* **Scalable and Fault-Tolerant Architecture** – Use microservices, centralized configuration, and resilience mechanisms to ensure scalability and availability.

**4. ARCHITECTURE OVERVIEW (HLD)**

****

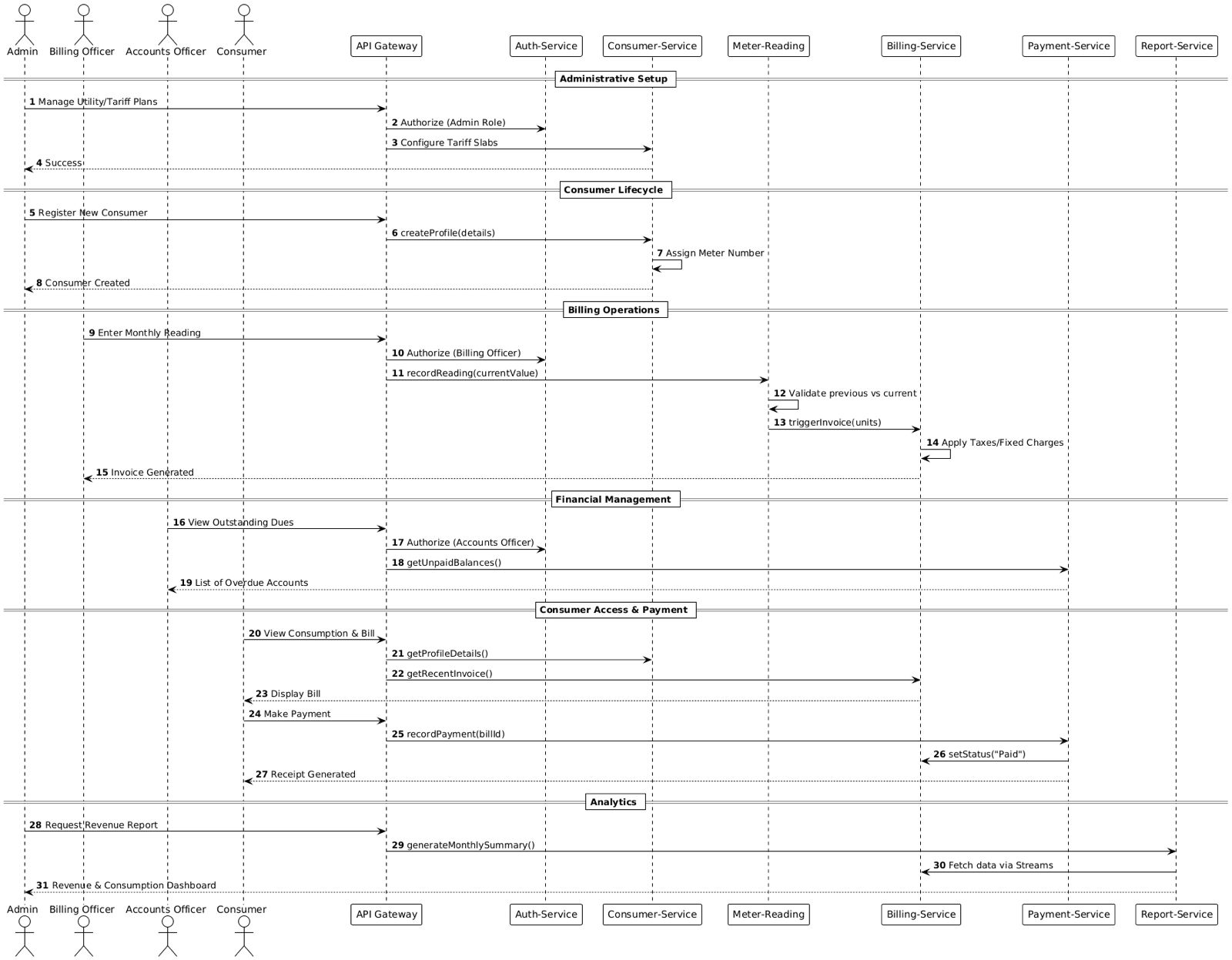
**Architecture Style**

* Microservices Architecture
* REST-based communication
* Containerized deployment

**Core Components**

* **Authentication and Authorization Service** – Manages users, roles, and JWT-based security across all microservices.
* **Consumer Management Service** – Handles consumer profiles, utility connections, and tariff plan mapping.
* **Meter Reading Service** – Stores and provides meter readings for accurate and historical bill calculations.
* **Billing Service** – Calculates tariffs, generates bills, and manages the bill lifecycle.
* **Payment Service** – Processes online and offline payments and updates billing status.
* **Notification Service** – Sends OTPs, bill alerts, and payment confirmations.
* **Dashboard Service** – Provides real-time operational and financial metrics for administrators.
* **Reporting Service** – Generates exportable financial and usage reports for auditing and analysis.
* **API Gateway** – Acts as a single entry point and routes requests to backend services.
* **Service Registry** – Enables service discovery, load balancing, and dynamic scaling.
* **Config Server** – Centralizes and manages configuration for all services.
* **Databases** – Maintains isolated MongoDB databases per service for scalability and data independence.
* **Resilience and Fault Tolerance Layer** – Prevents cascading failures and ensures system stability.

**USER FLOW**



**5. TECHNOLOGY STACK**

**Backend**

* Java 17
* Spring Boot
* Spring Web
* Spring Data MongoDB
* Spring Validation

**Frontend**

* Angular

**Database**

* MongoDB (one DB per microservice)

**DevOps**

* Docker
* Docker Compose
* Jenkins
* SonarQube

**Testing**

* JUnit 5
* Mockito
* Spring Boot Test

**6. MICROSERVICES DESIGN**

**6.1 Auth Service**

**Responsibilities**

* User registration and login
* JWT token generation and validation
* Role-based access control
* Forgot password and reset password flows

**APIs**

POST /auth/register

**Request Payload**

{

"username":"john\_doe",

"email":"john.doe@gmail.com",

"password":"Password@123",

"role":"CONSUMER"

}

**Response Payload**

{

"userId":"U101",

"message":"User registered successfully"

}

POST /auth/login

**Request Payload**

{

"username":"john\_doe",

"password":"Password@123"

}

**Response Payload**

{

"accessToken":"jwt-access-token",

"tokenType":"Bearer",

"role":"CONSUMER"

}

POST /auth/forgot-password

**Request Payload**

{

"email":"john.doe@gmail.com"

}

**Response Payload**

{

"message":"Password reset link sent to registered email"

}

POST /auth/reset-password

**Request Payload**

{

"resetToken":"reset-token-xyz",

"newPassword":"NewPassword@123"

}

**Response Payload**

{

"message":"Password reset successful"

}

PUT /auth/change-password

**Request Payload**

{

"oldPassword":"Password@123",

"newPassword":"UpdatedPassword@123"

}

**Response Payload**

{

"message":"Password updated successfully"

}

GET /auth/users

**Response Payload**

[

{

"userId":"U101",

"username":"john\_doe",

"email":"john.doe@gmail.com",

"role":"CONSUMER",

"status":"ACTIVE"

}

]

GET /auth/users/{userId}

**Response Payload**

{

"userId":"U101",

"username":"john\_doe",

"email":"john.doe@gmail.com",

"role":"CONSUMER",

"status":"ACTIVE"

}

DELETE /auth/users/{userId}

**Response Payload**

{

"message":"User deleted successfully"

}

**Database**

* users collection
* password\_reset\_tokens collection

**6.2 Consumer Service**

**Responsibilities**

* Consumer registration and profile management
* Linking consumers to utility connections
* Managing connection status

**APIs**

POST /consumers

**Request Payload**

{

"name":"Alice Johnson",

"phone":"9876543210",

"email":"alice@gmail.com",

"address":"Hyderabad"

}

**Response Payload**

{

"consumerId":"C201",

"message":"Consumer registered successfully"

}

GET /consumers/{consumerId}

**Response Payload**

{

"consumerId":"C201",

"name":"Alice Johnson",

"phone":"9876543210",

"email":"alice@gmail.com",

"address":"Hyderabad",

"status":"ACTIVE"

}

GET /consumers

**Response Payload**

[

{

"consumerId":"C201",

"name":"Alice Johnson",

"status":"ACTIVE"

}

]

PUT /consumers/{consumerId}

**Request Payload**

{

"phone":"9999999999",

"address":"Bangalore"

}

**Response Payload**

{

"message":"Consumer updated successfully"

}

DELETE /consumers/{consumerId}

**Response Payload**

{

"message":"Consumer deactivated successfully"

}

POST /connections

**Request Payload**

{

"consumerId":"C201",

"utilityId":"U1",

"meterNumber":"MTR1001",

"tariffId":"T101"

}

**Response Payload**

{

"connectionId":"CN301",

"status":"ACTIVE"

}

GET /connections/{consumerId}

**Response Payload**

[

{

"connectionId":"CN301",

"utilityId":"U1",

"meterNumber":"MTR1001",

"status":"ACTIVE"

}

]

PUT /connections/{connectionId}

**Request Payload**

{

"status":"DISCONNECTED"

}

**Response Payload**

{

"message":"Connection status updated"

}

**Database**

* consumers collection
* connections collection

**6.3 Utility Service**

**Responsibilities**

* Manage utility types (Electricity, Water, Gas, etc.)
* Define billing units and billing cycles
* Manage tariff slabs, taxes, and fixed charges

**APIs**

POST /utilities

**Request Payload**

{

"name":"ELECTRICITY",

"unit":"kWh",

"billingCycle":"MONTHLY"

}

**Response Payload**

{

"utilityId":"U1",

"message":"Utility created successfully"

}

GET /utilities

**Response Payload**

[

{

"utilityId":"U1",

"name":"ELECTRICITY",

"unit":"kWh",

"billingCycle":"MONTHLY"

}

]

GET /utilities/{utilityId}

**Response Payload**

{

"utilityId":"U1",

"name":"ELECTRICITY",

"unit":"kWh",

"billingCycle":"MONTHLY"

}

PUT /utilities/{utilityId}

**Request Payload**

{

"unit":"Units",

"billingCycle":"MONTHLY"

}

**Response Payload**

{

"message":"Utility updated successfully"

}

POST /tariffs

**Request Payload**

{

"utilityId":"U1",

"slabs":[

{"min":0,"max":100,"rate":3.5},

{"min":101,"max":300,"rate":5.0}

],

"fixedCharge":120,

"taxPercent":5

}

**Response Payload**

{

"tariffId":"T101",

"message":"Tariff created successfully"

}

GET /tariffs

**Response Payload**

[

{

"tariffId":"T101",

"utilityId":"U1",

"fixedCharge":120,

"taxPercent":5

}

]

GET /tariffs/{tariffId}

**Response Payload**

{

"tariffId":"T101",

"utilityId":"U1",

"slabs":[

{"min":0,"max":100,"rate":3.5},

{"min":101,"max":300,"rate":5.0}

]

}

GET /tariffs/utility/{utilityId}

**Response Payload**

{

"tariffId":"T101",

"utilityId":"U1",

"fixedCharge":120,

"taxPercent":5

}

PUT /tariffs/{tariffId}

**Request Payload**

{

"fixedCharge":150,

"taxPercent":6

}

**Response Payload**

{

"message":"Tariff updated successfully"

}

**Database**

* utilities collection
* tarrif\_plans collection

**6.4 Meter Service**

**Responsibilities**

* Record monthly meter readings
* Validate readings against previous values
* Calculate units consumed per billing cycle

**APIs**

POST /meter-readings

**Request Payload**

{

"consumerId":"C201",

"utilityId":"U1",

"previousReading":330,

"currentReading":450,

"readingDate":"2025-03-01"

}

**Response Payload**

{

"meterReadingId":"MR401",

"unitsConsumed":120,

"message":"Meter reading recorded successfully"

}

GET /meter-readings/{consumerId}

**Response Payload**

[

{

"meterReadingId":"MR401",

"utilityId":"U1",

"previousReading":330,

"currentReading":450,

"unitsConsumed":120,

"readingDate":"2025-03-01"

}

]

GET /meter-readings/latest/{consumerId}

**Response Payload**

{

"meterReadingId":"MR401",

"utilityId":"U1",

"currentReading":450,

"unitsConsumed":120,

"readingDate":"2025-03-01"

}

**Database**

* meter\_readings collection

**6.5 Billing Service**

**Responsibilities**

* Generate bills based on meter readings and tariffs
* Apply fixed charges and taxes
* Maintain bill status transitions

**APIs**

POST /bills/generate

**Request Payload**

{

"consumerId":"C201",

"utilityId":"U1",

"billingMonth":"MARCH",

"billingYear":2025

}

**Response Payload**

{

"billId":"B501",

"totalUnits":120,

"energyCharge":950,

"fixedCharge":120,

"taxAmount":45,

"totalAmount":1115,

"status":"GENERATED"

}

GET /bills/{billId}

**Response Payload**

{

"billId":"B501",

"consumerId":"C201",

"utilityId":"U1",

"billingPeriod":"MARCH-2025",

"totalAmount":1115,

"status":"DUE"

}

GET /bills/consumer/{consumerId}

**Response Payload**

[

{

"billId":"B501",

"billingPeriod":"MARCH-2025",

"totalAmount":1115,

"status":"DUE"

}

]

GET /bills

**Response Payload**

[

{

"billId":"B501",

"consumerId":"C201",

"totalAmount":1115,

"status":"DUE"

}

]

PUT /bills/{billId}/status

**Request Payload**

{

"status":"DUE"

}

**Response Payload**

{

"message":"Bill marked as PAID"

}

**Database**

* bills collection

**6.6 Notification Service**

**Responsibilities**

* Send bill generation notifications
* Send payment reminders
* Send password reset notifications

**APIs**

POST /notifications/bill-generated

**Request Payload**

{

"consumerId":"C201",

"billId":"B501",

"message":"Your electricity bill for March 2025 has been generated."

}

**Response Payload**

{

"notificationId":"N601",

"status":"SENT"

}

POST /notifications/payment-reminder

**Request Payload**

{

"consumerId":"C201",

"billId":"B501",

"dueDate":"2025-03-15"

}

**Response Payload**

{

"notificationId":"N602",

"status":"SENT"

}

POST /notifications/password-reset

**Request Payload**

{

"email":"john.doe@gmail.com",

"resetLink":"https://app/reset-password?token=abc123"

}

**Response Payload**

{

"email":"john.doe@gmail.com",

"resetLink":"https://app/reset-password?token=abc123"

}

GET /notifications/{consumerId}

**Response Payload**

[

{

"notificationId":"N601",

"type":"BILL\_GENERATED",

"message":"Your electricity bill has been generated.",

"status":"SENT"

}

]

**Database**

* notifications collection

**6.7 Report Service**

**Responsibilities**

* Generate revenue reports
* Identify outstanding dues
* Analyze consumption patterns

**APIs**

GET /reports/monthly-revenue

**Response Payload**

[

{

"month":"MARCH-2025",

"totalRevenue":350000

}

]

GET /reports/outstanding-dues

**Response Payload**

{

"totalOutstandingAmount":125000

}

GET /reports/consumption-by-utility

**Response Payload**

[

{

"utility":"ELECTRICITY",

"totalUnitsConsumed":95000

}

]

GET /reports/consumer-summary/{consumerId}

**Response Payload**

{

"consumerId":"C201",

"totalBills":12,

"totalPaid":9800,

"totalDue":1115

}

**Database**

* Aggregation Queries

**7. DATA DESIGN (LLD)**

**User Document**

{

"\_id": ObjectId,

"username": "string",

"password": "string (encrypted)",

"role": "ADMIN | USER | ACCOUNTS\_OFFICER | BILLING\_OFFICER",

"active": true,

"createdAt": ISODate,

"updatedAt": ISODate

}

**Password\_reset\_token Document**

{

"userId":"U101",

"resetToken":"abc123",

"expiresAt":"2025-01-11T10:00",

"used":false

}

**Consumer Document**

{

"\_id": ObjectId,

"name": "string",

"email": "string",

"phone": "string",

"address": "string",

"status": "ACTIVE | INACTIVE",

"createdAt": ISODate,

"updatedAt": ISODate

}

**Connection Document**

{

"\_id": ObjectId,

"consumerId": "ObjectId",

"utilityType": "ELECTRICITY | WATER | GAS | INTERNET",

"meterNumber": "string",

"tariffPlan": "DOMESTIC | COMMERCIAL | INDUSTRIAL",

"status": "ACTIVE | DISCONNECTED",

"createdAt": ISODate

}

**Meter Reading Document**

{

"\_id": ObjectId,

"connectionId": "ObjectId",

"readingValue": Number,

"readingDate": ISODate,

"createdAt": ISODate

}

**Billing document**

{

"\_id": ObjectId,

"consumerId": "ObjectId",

"connectionId": "ObjectId",

"billingPeriod": "YYYY-MM",

"unitsConsumed": Number,

"amount": Number,

"dueDate": ISODate,

"status": "GENERATED | PAID | OVERDUE",

"createdAt": ISODate

}

**Payment document**

{

"\_id": ObjectId,

"billId": "ObjectId",

"consumerId": "ObjectId",

"amount": Number,

"mode": "ONLINE | OFFLINE",

"status": "INITIATED | SUCCESS | FAILED",

"otp": "string (hashed)",

"otpExpiry": ISODate,

"createdAt": ISODate

}

**Notification document**

{

"\_id": ObjectId,

"userId": "ObjectId",

"type": "EMAIL | SMS",

"message": "string",

"status": "SENT | FAILED",

"sentAt": ISODate

}

**Report document**

{

"\_id": ObjectId,

"reportType": "PAYMENT | BILLING | CONSUMER",

"generatedBy": "ADMIN",

"data": "JSON",

"generatedAt": ISODate

}

**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. Business Rules**

1. Authentication & Authorization Service

* Users must authenticate using valid credentials.
* JWT tokens are issued after successful authentication.
* Tokens contain user identity and role information.
* Role-based access control is enforced across all services.
* Only active users are allowed to access the system.

2. Consumer Service

* Each consumer must be linked to a valid authenticated user.
* A consumer can have multiple utility connections.
* Consumer profile data can be modified only by authorized users.
* Consumer records are soft-deleted to preserve historical data.

3. Utility Connection Service

* Each utility connection must have a defined utility type and tariff plan.
* A connection belongs to exactly one consumer.
* Tariff changes do not affect previously generated bills.

Only active connections are eligible for billing and meter readings.

4. Meter Reading Service

* Meter readings can be recorded only for active connections.
* New readings must not be less than previous readings.
* Only authorized roles can submit meter readings.
* Meter readings are immutable once recorded.

5. Billing Service

* Bills are generated based on meter readings and tariff plans.
* Each bill has a defined lifecycle: Generated, Paid, or Overdue.
* Bills marked as paid cannot be modified.
* Overdue status is determined based on due date.

6. Payment Service

* Online payments require OTP verification.
* Payment is confirmed only after successful OTP validation.
* Offline payments can be recorded only by authorized staff.
* A bill is marked as paid only after successful payment confirmation.

7. Notification Service

* Notifications are sent for bill generation and payment events.
* Notification failures do not impact core business flow.
* Each notification is associated with a specific user and event.

8. Dashboard & Reporting Service

* Dashboard data is read-only and aggregated from multiple services.
* Financial reports are accessible only to authorized roles.
* Reports reflect consistent and finalized transactional data.

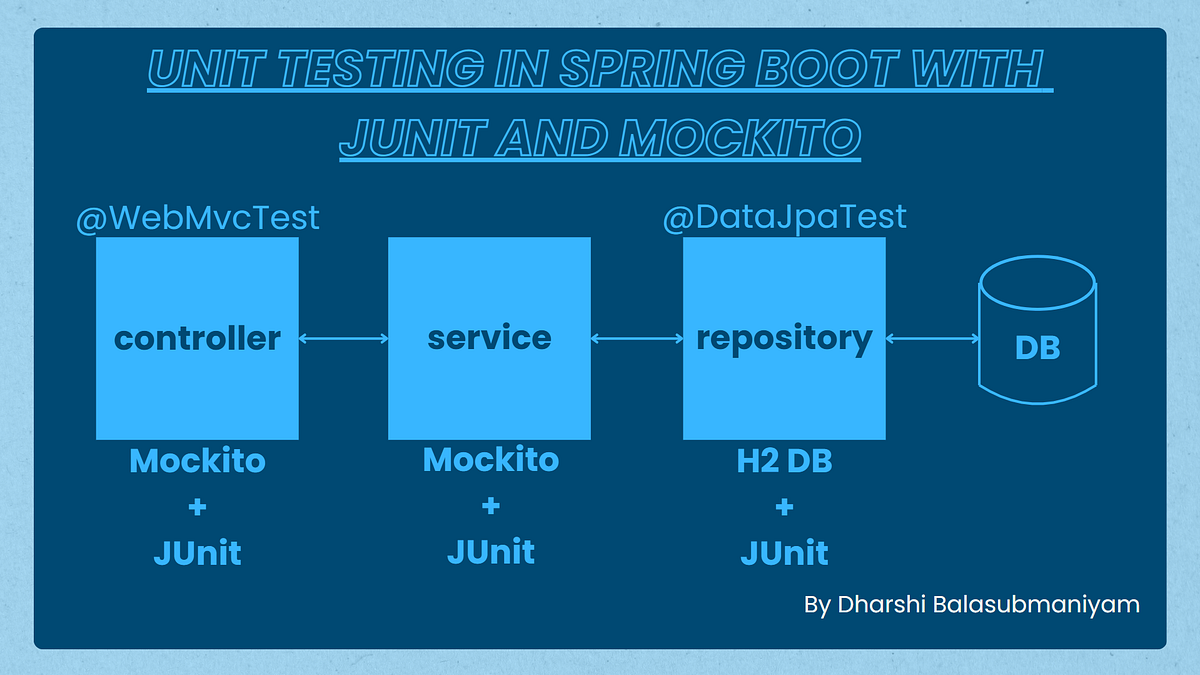
9. System-Wide Rules

* All inter-service communication is secured using JWT.
* Services are loosely coupled and independently deployable.
* Circuit breakers handle service failures gracefully.
* All critical actions are logged for auditing purposes.

**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 |

**12. TESTING STRATEGY**



**Backend**

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

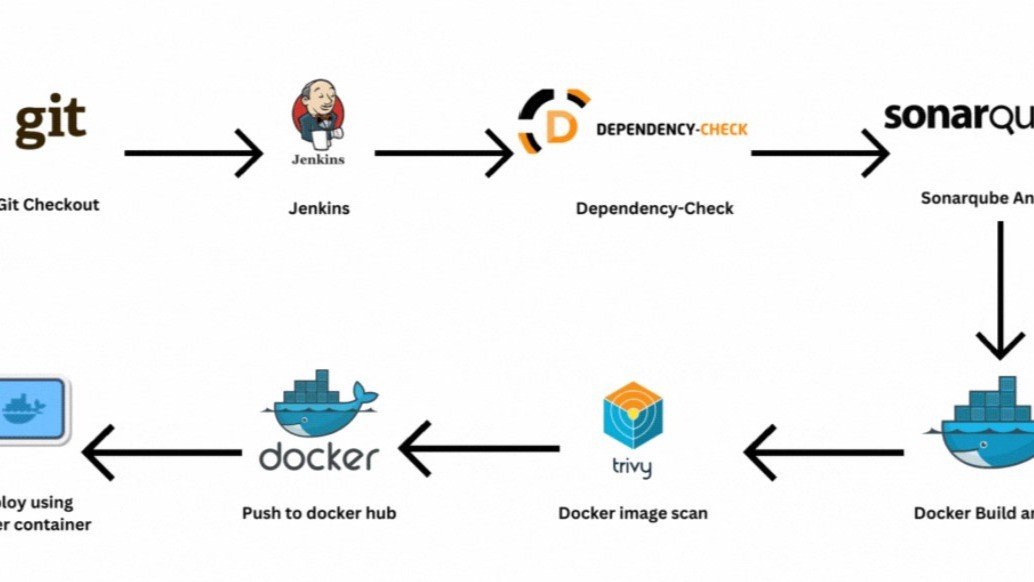
**Frontend**

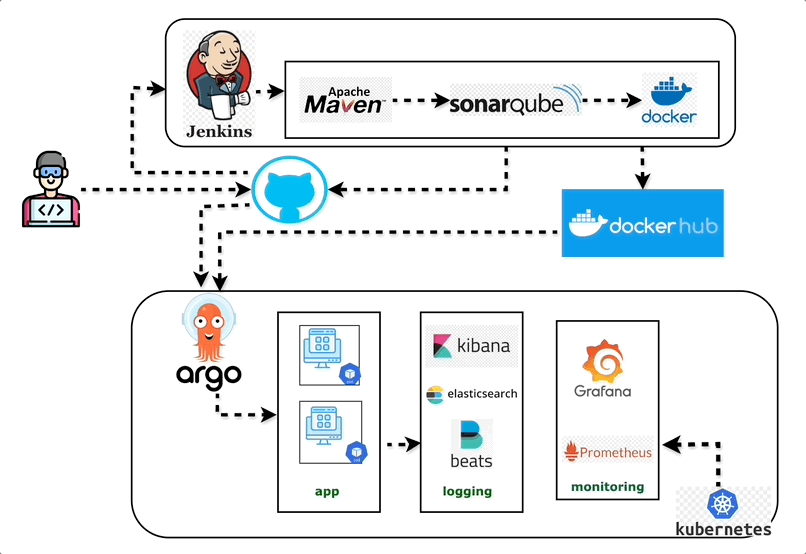
* Component tests
* Service tests

**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

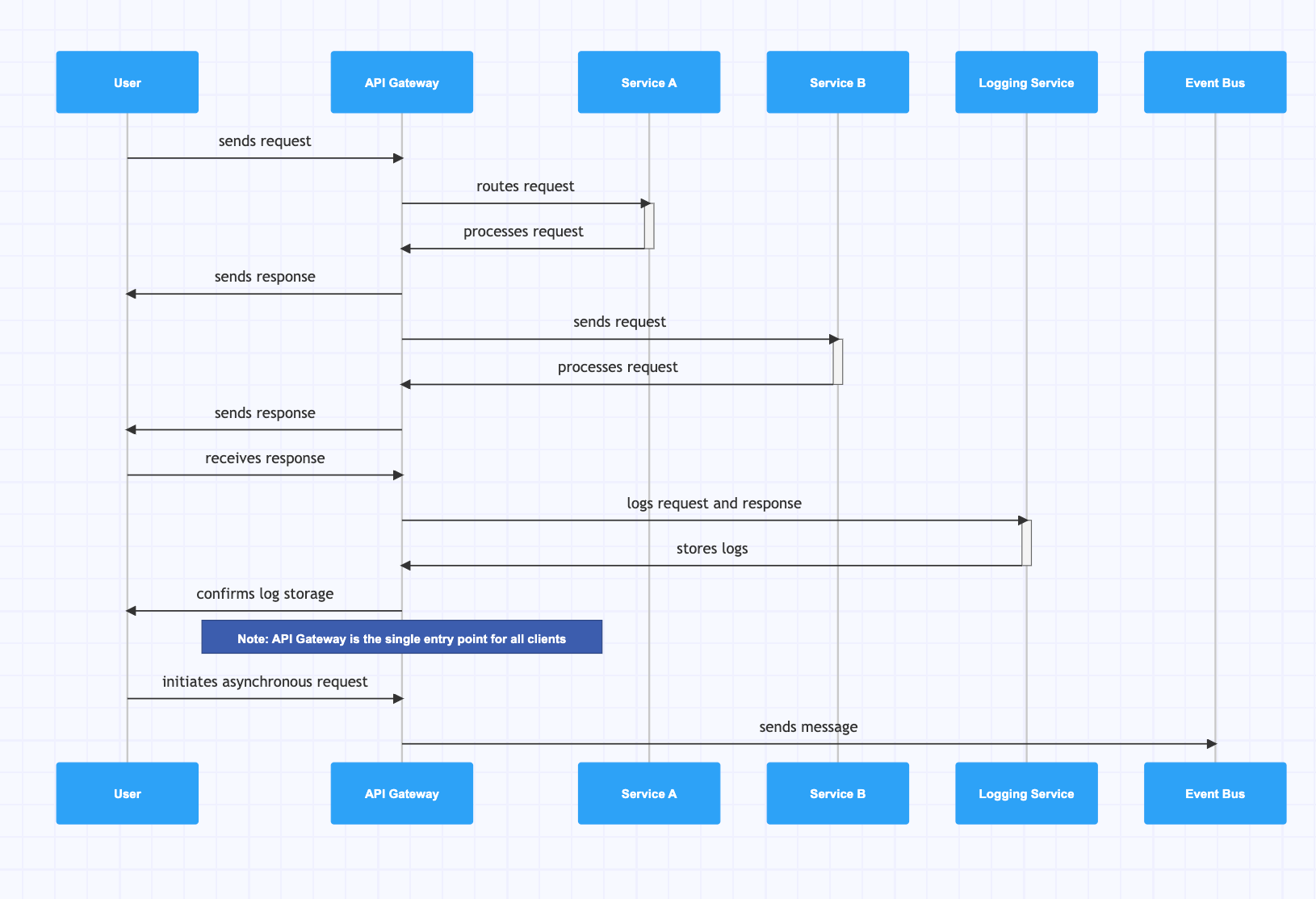
**16. FUTURE ENHANCEMENTS**

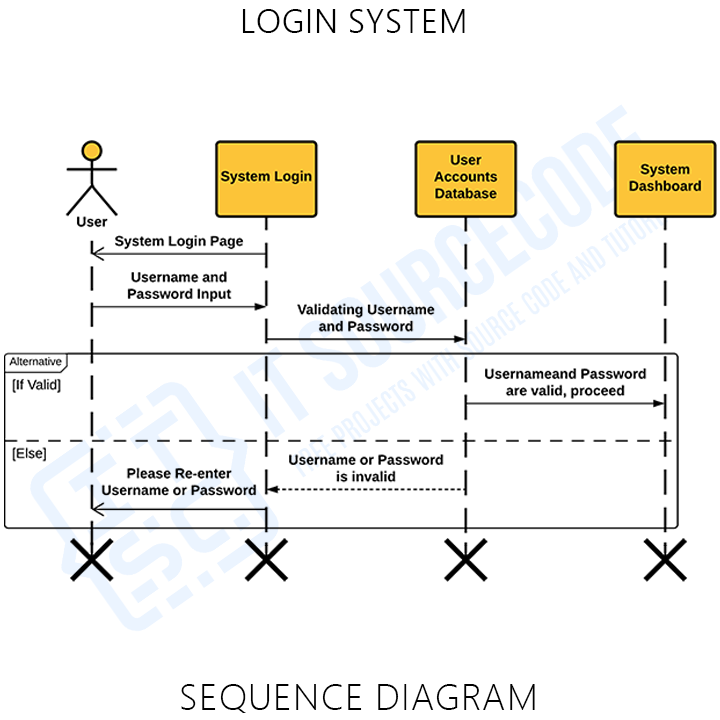
* Spring Cloud Gateway
* Kafka-based async communication
* Kubernetes deployment
* Centralized logging (ELK)

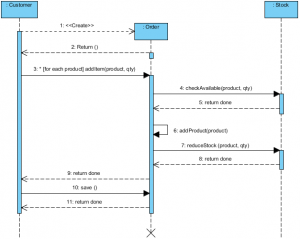
**17. CONCLUSION**

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

**SEQUENCE DIAGRAMS — SMART ORDER MANAGEMENT SYSTEM**







**1. User Registration — 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

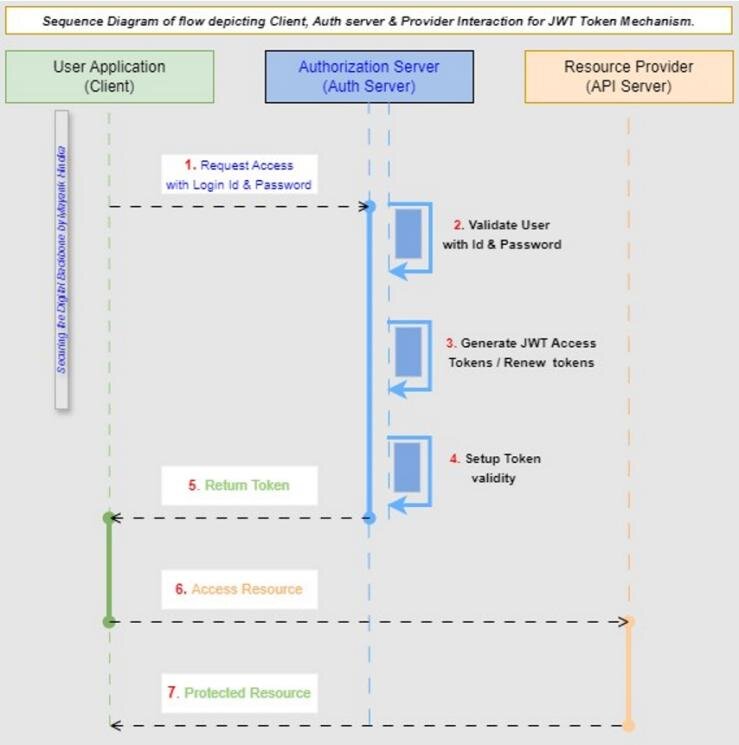
**Participants**

* User
* Angular UI
* User Service
* MongoDB

**Key Design Points**

* Validation at API boundary
* Encryption at service layer
* No direct DB access from UI

**2. User Login — Sequence Diagram**



**Scenario**

Registered user logs into the system.

**Flow**

1. User submits login form
2. Angular calls POST /users/login
3. User Service fetches user from MongoDB
4. Password is verified
5. (Optional) JWT token generated
6. Response sent to Angular
7. Angular stores token and navigates user

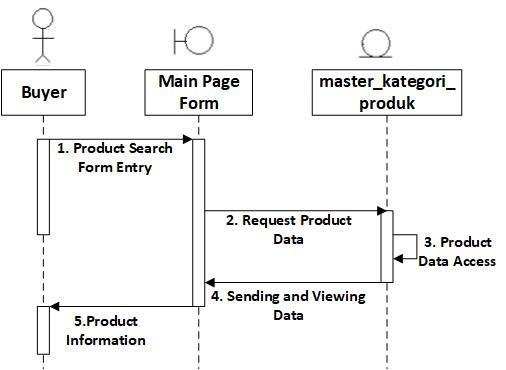
**Participants**

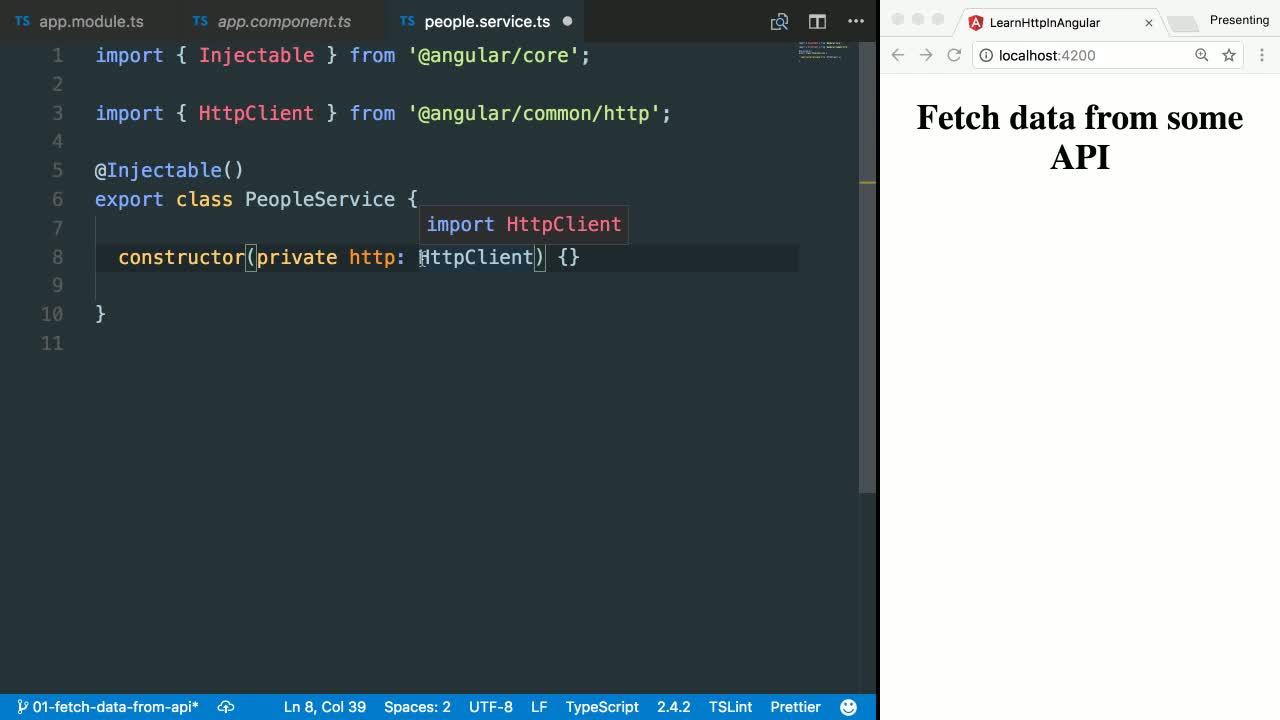
* User
* Angular UI
* User Service
* MongoDB

**Failure Cases**

* Invalid credentials → 401
* Inactive user → 403

**3. Product Listing — Sequence Diagram**





**Scenario**

User views available products.

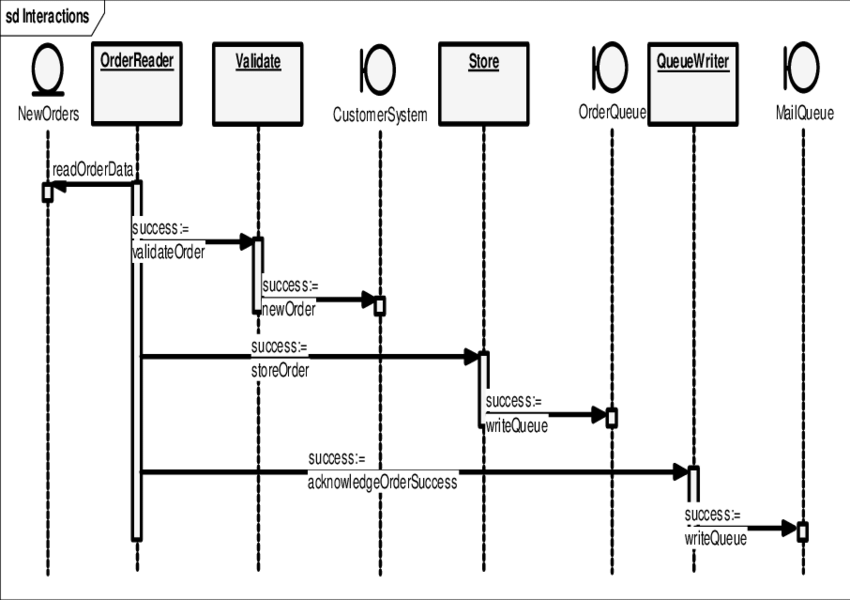
**Flow**

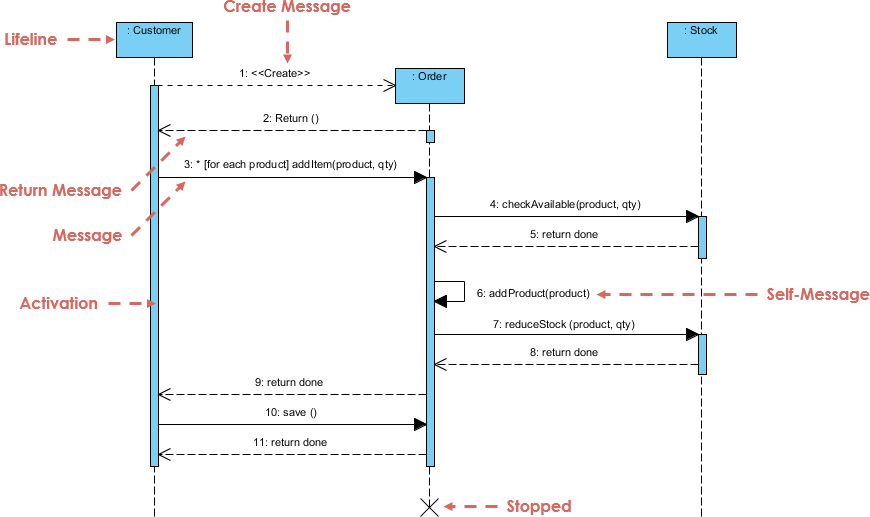
1. Angular loads product page
2. GET /products sent to Product Service
3. Product Service queries MongoDB
4. Product list returned to Angular

**Participants**

* Angular UI
* Product Service
* MongoDB

**4. Order Placement — Sequence Diagram (MOST IMPORTANT)**





**Scenario**

User places an order for one or more products.

**Flow**

1. User clicks **Place Order** in Angular
2. Angular sends POST /orders to Order Service
3. Order Service validates request
4. Order Service calls Product Service to check stock
5. Product Service validates availability
6. Order Service creates order
7. Product Service reduces inventory
8. Order saved in MongoDB
9. Success response returned to Angular

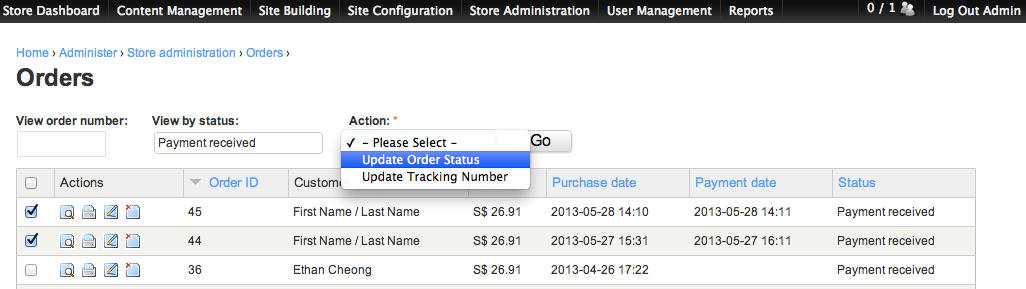
**Participants**

* User
* Angular UI
* Order Service
* Product Service
* MongoDB

**Key Design Decisions**

* Inventory validation before order creation
* Atomic stock update logic
* Failure stops order creation

**5. Order Status Update (Admin) — Sequence Diagram**



**Scenario**

Admin updates order status.

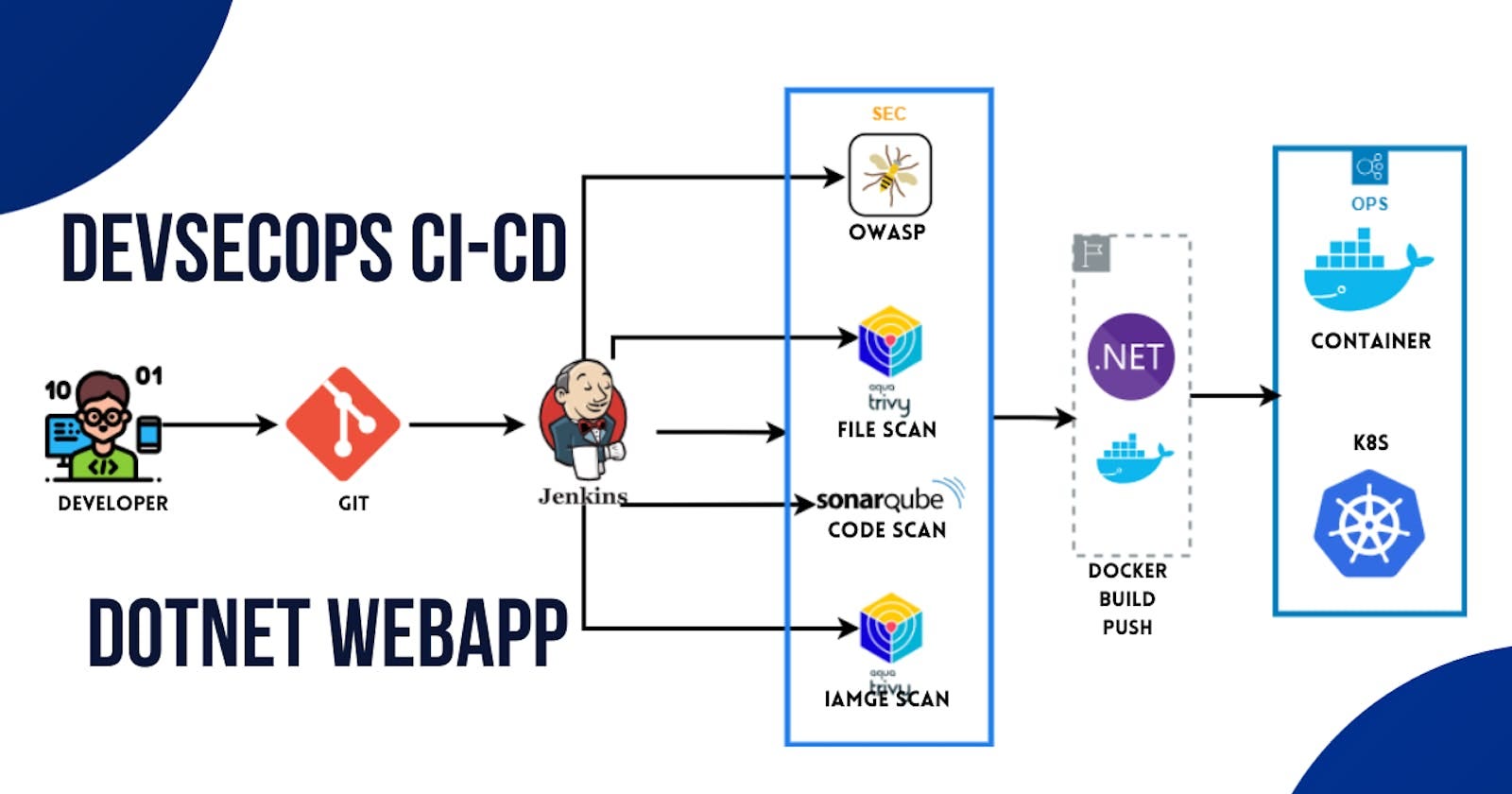
**Flow**

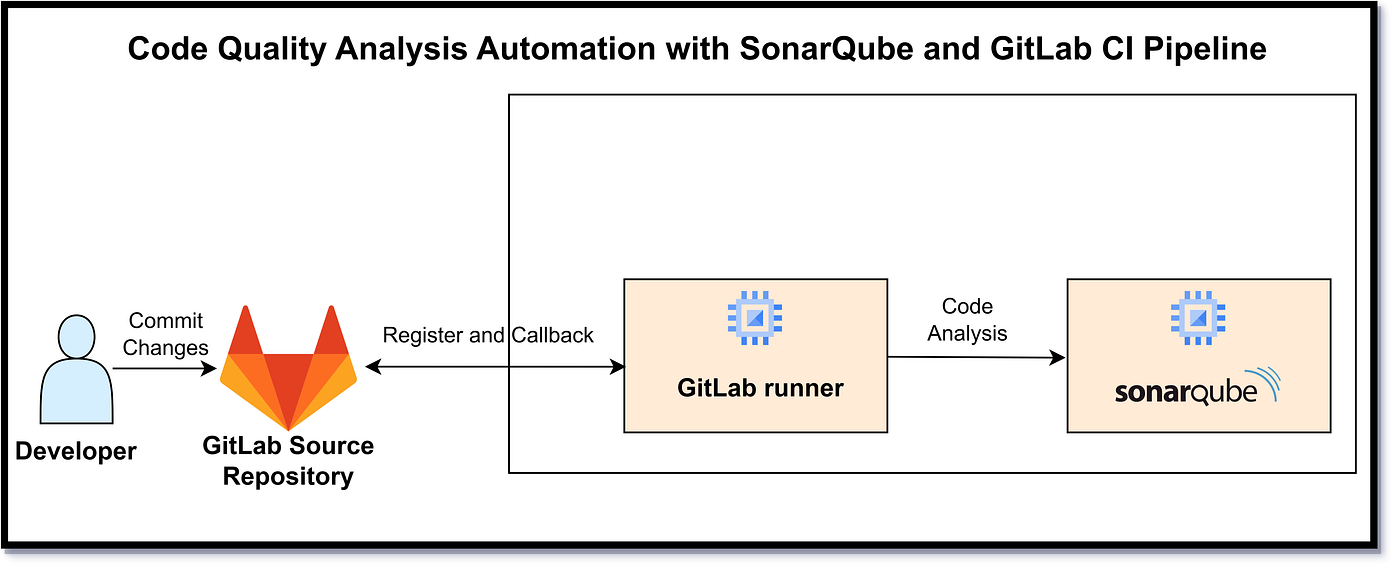
1. Admin sends update request from Angular
2. PUT /orders/{id}/status sent to Order Service
3. Role validation (ADMIN only)
4. Order status transition validated
5. Order updated in MongoDB
6. Updated status returned

**Valid Transitions**

* CREATED → PLACED
* PLACED → COMPLETED
* PLACED → CANCELLED

**6. 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

**7. 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

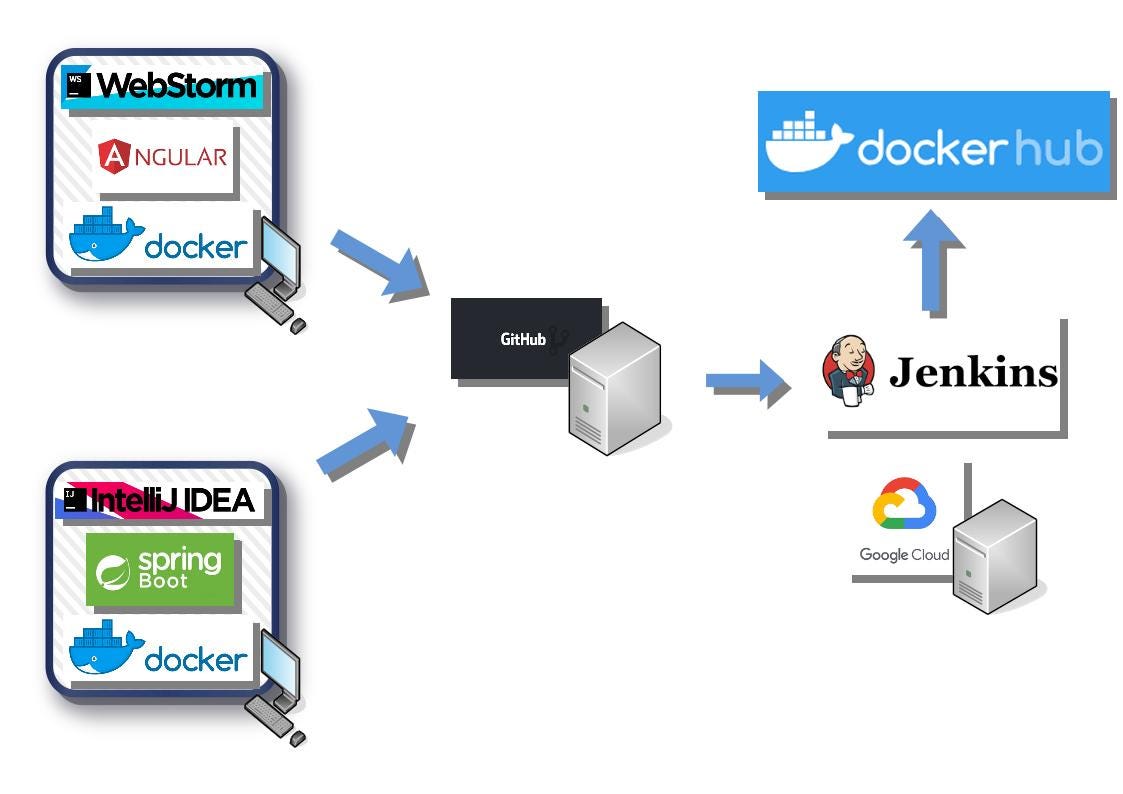
└─ README.md

**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.