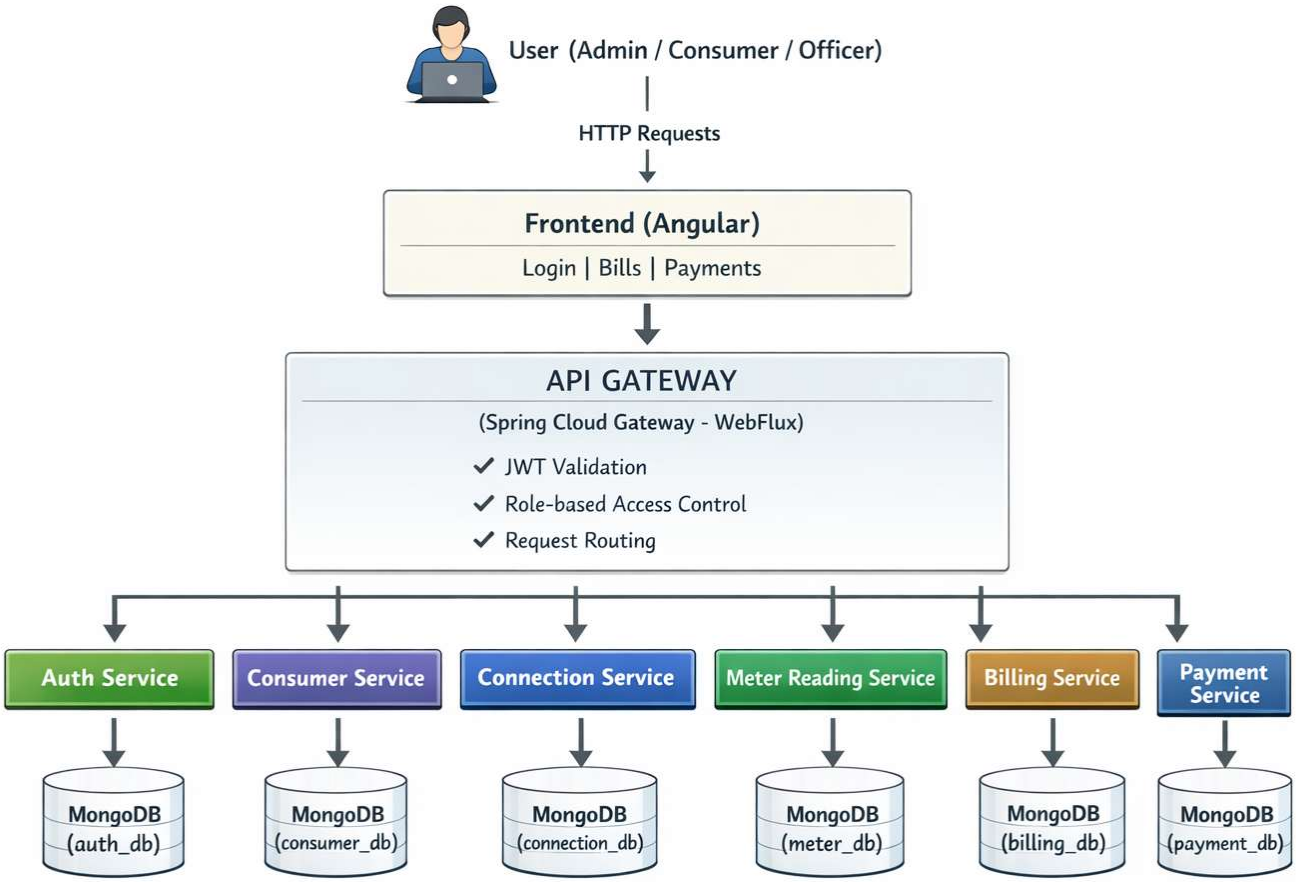
SOFTWARE DESIGN DOCUMENT (SDD)

Utility Billing System



---

1. DOCUMENT CONTROL

Item	Details
Project Name	Utility Billing System
Version	1.0
Author	Modala Poojitha
Date	27-12-25
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 Utility Billing 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:

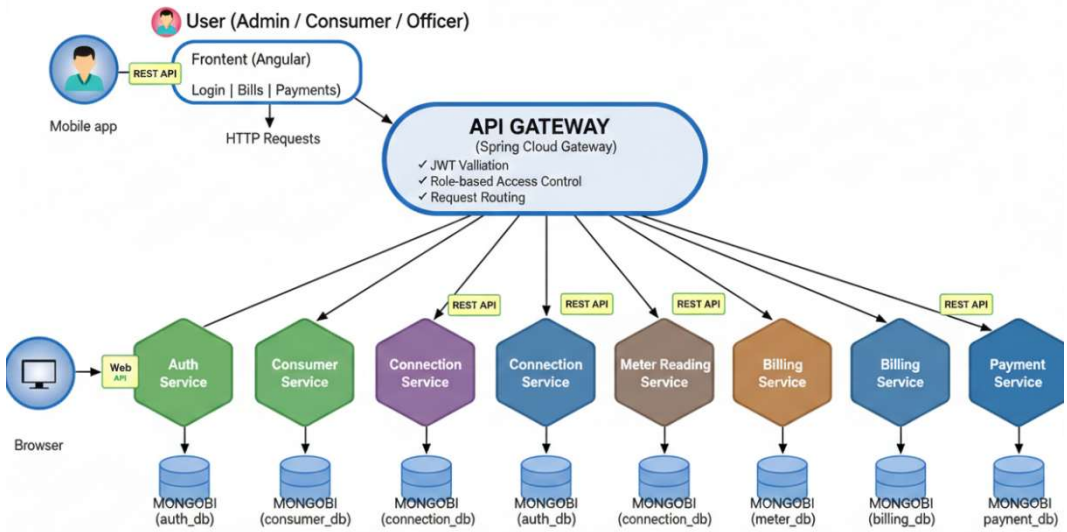
- Manage Consumers
- Automate Billing
- Track Payments
- Support Growth

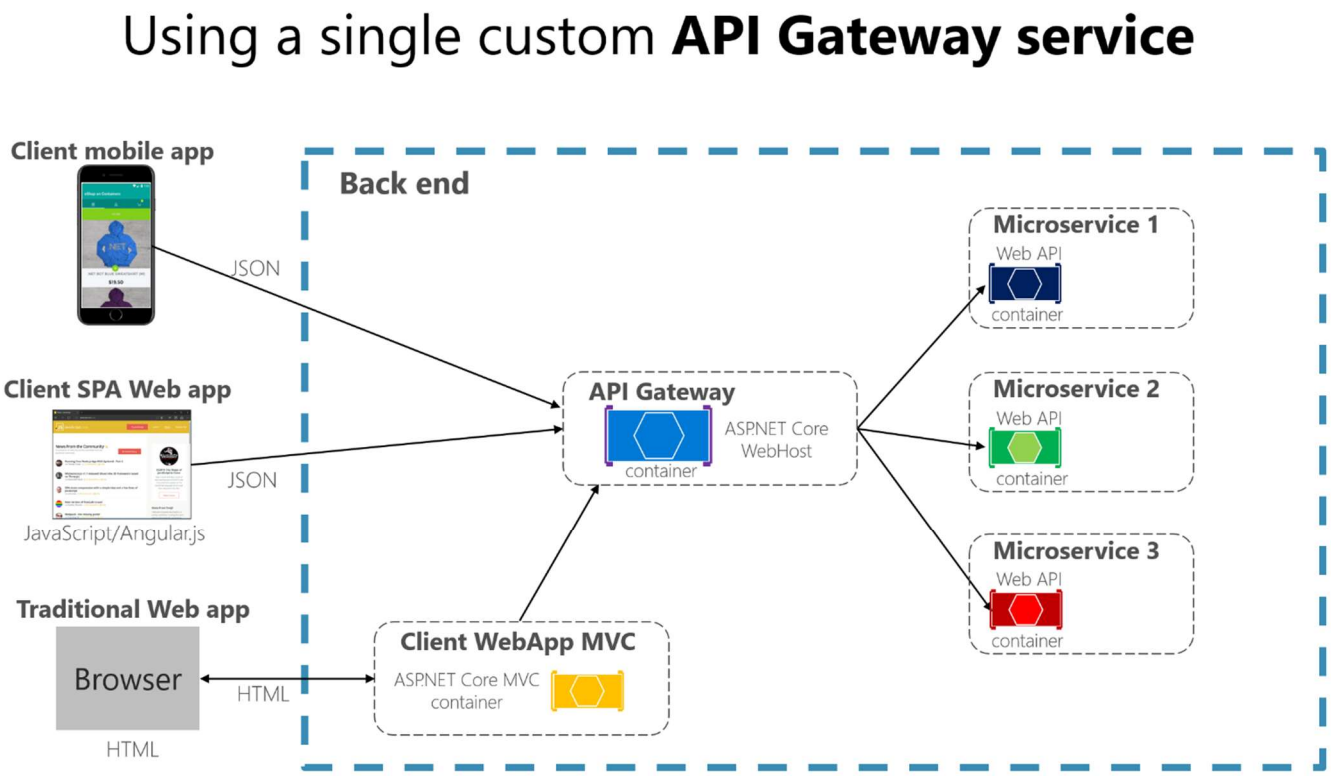
High-Level Features

- User Login & Security
- Consumer Management
- Meter Reading & Billing
- Payment Tracking
- Role-based access (ADMIN / CONSUMER / BILLING OFFICER / ACCOUNT OFFICER)

---

4. ARCHITECTURE OVERVIEW (HLD)





Architecture Style

- Microservices Architecture
- REST-based communication
- Containerized deployment

Core Components

1. Angular Frontend
2. API Gateway
3. Auth Service
4. Consumer Service
5. Connection Service
6. Meter Reading Service
7. Billing Service
8. Payment Service
9. Notification Service
10. MongoDB (per service)
11. CI/CD Pipeline

---

5. TECHNOLOGY STACK

Backend

- Java 17
- Spring Boot

- Spring Reactive Web
- Spring Data Reactive MongoDB
- Spring Validation

Frontend

- Angular
- Angular Material
- RxJS

Database

- MongoDB (one DB per microservice)

DevOps

- Docker
- Docker Compose
- Jenkins
- SonarQube

Testing

- JUnit 5
- Mockito
- Spring Boot Test
- Jasmine / Karma

---

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
POST /auth/login
POST /auth/forgot-password
POST /auth/reset-password
PUT /auth/change-password
GET /auth/users/{userId}
DELETE /auth/users/{userId}
```

Database

- users collection

6.2 Consumer Service

Responsibilities

- Consumer registration and profile management
- Consumer personal details
- Link consumer to user account

APIs

```
POST /consumers
GET /consumers
GET /consumers/{consumerId}
PUT /consumers/{consumerId}
```

Database

- consumers collection

6.3 Connection Service

Responsibilities

- Manage utility types
- Manage tariff plans & slabs
- Manage billing cycles
- Create consumer connections

APIs

```
POST / utilities
GET / utilities
PUT /utilities/{id}

POST /tariffs
GET /tariffs
POST /tariffs/{id}/slabs
GET /tariffs/{id}/slabs

POST /billing-cycles
GET /billing-cycles

POST /connections
GET /connections/{id}
GET /connections /consumer/{consumerId}
PUT /connections/{id}
```

Database

- utility_types collection
- tariff_plans collection
- tariff_slabs collection
- billing_cycles collection
- connections collection

6.4 Meter Reading Service

Responsibilities

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

APIs

```
POST   /meter-readings
GET    /meter-readings/connection/{connectionId}
GET    /meter-readings/month/{yyyy-mm}
```

Database

- meter_readings collection

6.5 Billing Service

Responsibilities

- Generate bills based on meter readings and tariffs
- Apply fixed charges and taxes
- Track bill status

APIs

```
POST /bills/generate
GET /bills/{billId}
GET /bills/consumer/{consumerId}
GET /bills/status/{status}
PUT /bills/{id}/mark-paid
```

Database

- bills collection

6.6 Payment Service

Responsibilities

- Record payments
- Update bill status
- Track outstanding balance

APIs

```
POST /payments
GET /payments/bill/{billId}
GET /payments/consumer/{consumerId}
```

Database

- Payments collection

6.7 Notification Service

Responsibilities

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

APIs

```
POST   /notifications/bill-generated
POST   /notifications/payment-reminder
POST   /notifications/password-reset
```

Database

- notifications collection

6.8 Report Service

Responsibilities

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

APIs

```
GET /reports/monthly-revenue
GET /reports/outstanding-dues
GET /reports/consumption-by-utility
GET /reports/consumer-summary/{consumerId}
```

Database

- Aggregation Queries on bills, payments, meter_readings

---

7. DATA DESIGN (LLD)

User Document

```
{
  "id": "Long",
  "username": "String",
  "password": "String",
  "role": "ADMIN | BILLING_OFFICER | ACCOUNTS_OFFICER | CONSUMER",
  "enabled": "Boolean",
  "createdAt": "Timestamp"
}
```

Consumer Document

```
{
  "id": "Long",
  "userId": "Long",
  "name": "String",
  "address": "String",
  "phone": "String",
  "createdAt": "Timestamp"
}
```

Connection Document

Utility Type

```
{
  "id": "Long",
  "name": "String",
  "unit": "String"
}
```

TariffPlan

```
{
  "id": "Long",
  "utilityTypeId": "Long",
  "name": "String",
  "fixedCharge": "Decimal",
  "taxPercent": "Decimal",
  "active": "Boolean"
}
```

TariffSlab

```
{
  "id": "Long",
  "tariffPlanId": "Long",
  "minUnit": "Integer",
  "maxUnit": "Integer",
  "ratePerUnit": "Decimal"
}
```

BillingCycle

```
{
  "id": "Long",
  "cycleName": "String",
  "dueDays": "Integer"
}
```

Connection

```
{
  "id": "Long",
  "consumerId": "Long",
  "utilityTypeId": "Long",
  "tariffPlanId": "Long",
```

```
  "billingCycleId": "Long",
  "meterNumber": "String",
  "active": "Boolean",
  "createdAt": "Timestamp"
}
```

MeterReading Document

```
{
  "id": "Long",
  "connectionId": "Long",
  "readingMonth": "YYYY-MM",
  "previousReading": "Decimal",
  "currentReading": "Decimal",
  "unitsConsumed": "Decimal",
  "createdAt": "Timestamp"
}
```

Bill Document

```
{
  "id": "Long",
  "connectionId": "Long",
  "billMonth": "YYYY-MM",
  "unitsConsumed": "Decimal",
  "energyCharge": "Decimal",
  "fixedCharge": "Decimal",
  "taxAmount": "Decimal",
  "totalAmount": "Decimal",
  "status": "GENERATED | DUE | PAID | OVERDUE",
  "dueDate": "Date",
  "generatedAt": "Timestamp"
}
```

Payment Document

Payment

```
{
  "id": "Long",
  "billId": "Long",
  "amount": "Decimal",
  "paymentMode": "ONLINE | CASH | UPI",
  "paymentDate": "Timestamp"
}
```

PaymentRequest

```
{
  "billId": "Long",
  "amount": "Decimal",
  "paymentMode": "ONLINE | CASH | UPI"
}
```

Notification Document

```
{
  "id": "Long",
  "consumerId": "Long",
  "eventType": "BILL_GENERATED | PAYMENT_RECEIVED | PAYMENT_DUE",
  "message": "String",
  "sentAt": "Timestamp"
}
```

Report Document

MonthlyRevenueReport

```
{
  "month": "YYYY-MM",
  "totalRevenue": "Decimal",
  "paidAmount": "Decimal",
```

```
  "outstandingAmount": "Decimal"
}
```

OutstandingDue

```
{
  "billId": "Long",
  "consumerId": "Long",
  "totalAmount": "Decimal",
  "dueDate": "Date",
  "daysOverdue": "Integer"
}
```

ConsumptionByUtility

```
{
  "utilityType": "String",
  "totalUnitsConsumed": "Decimal"
}
```

ConsumerSummary

```
{
  "consumerId": "Long",
  "totalBills": "Integer",
  "totalPaid": "Decimal",
  "totalOutstanding": "Decimal"
}
```

---

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

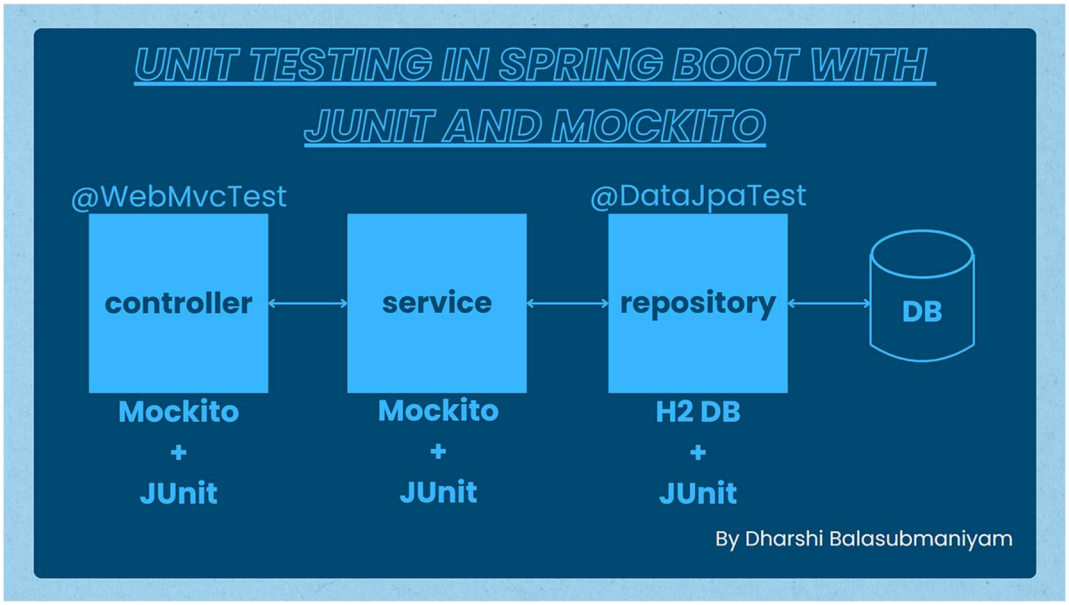
---

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

---

12. TESTING STRATEGY



Backend

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

Frontend

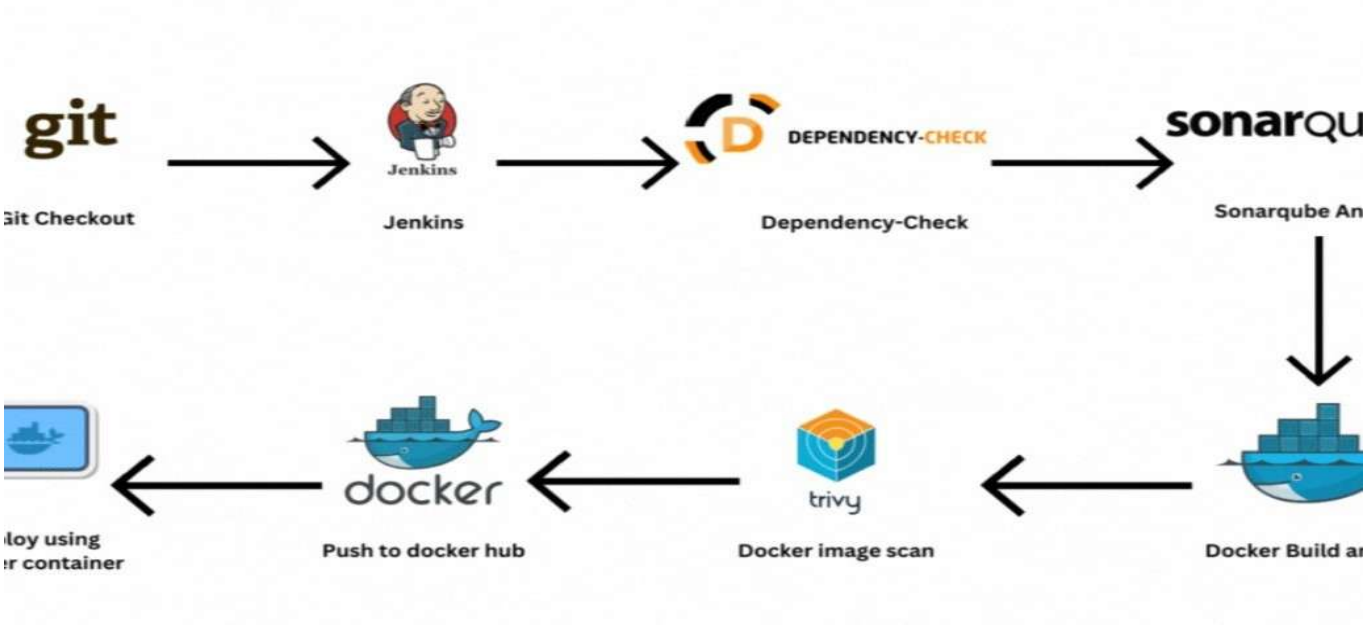
- Component tests
- Service tests

Quality Gates

- SonarQube enforced
- Build fails on violations

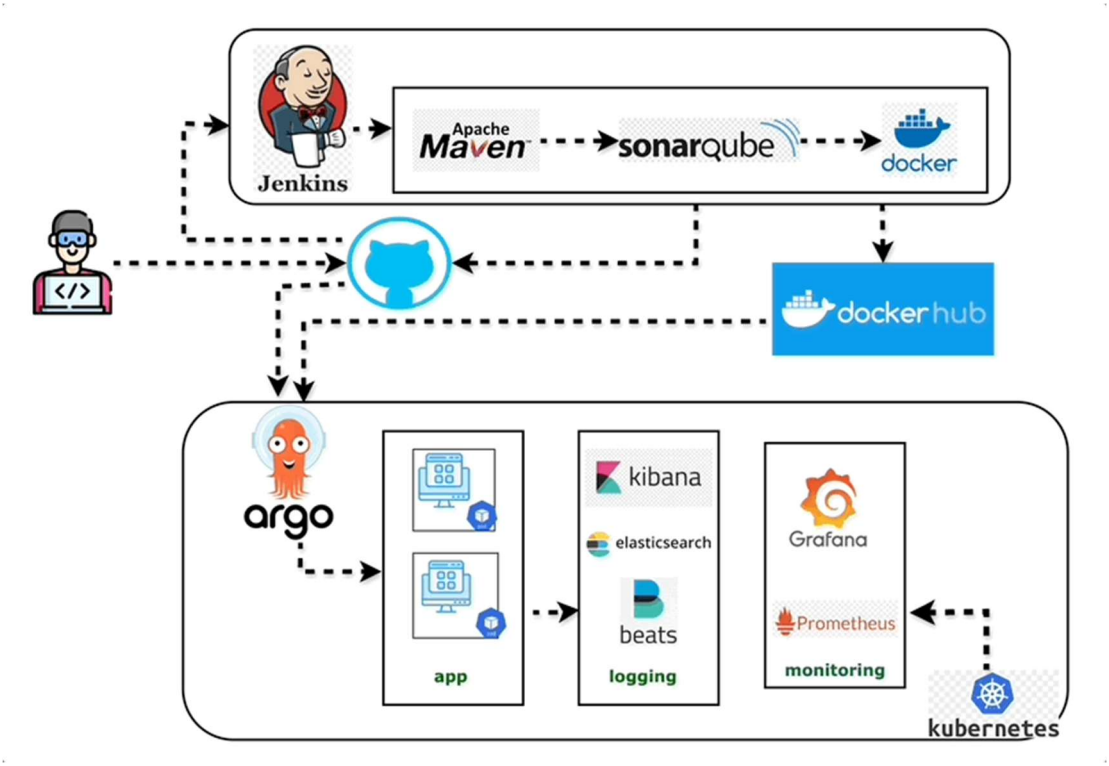
---

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

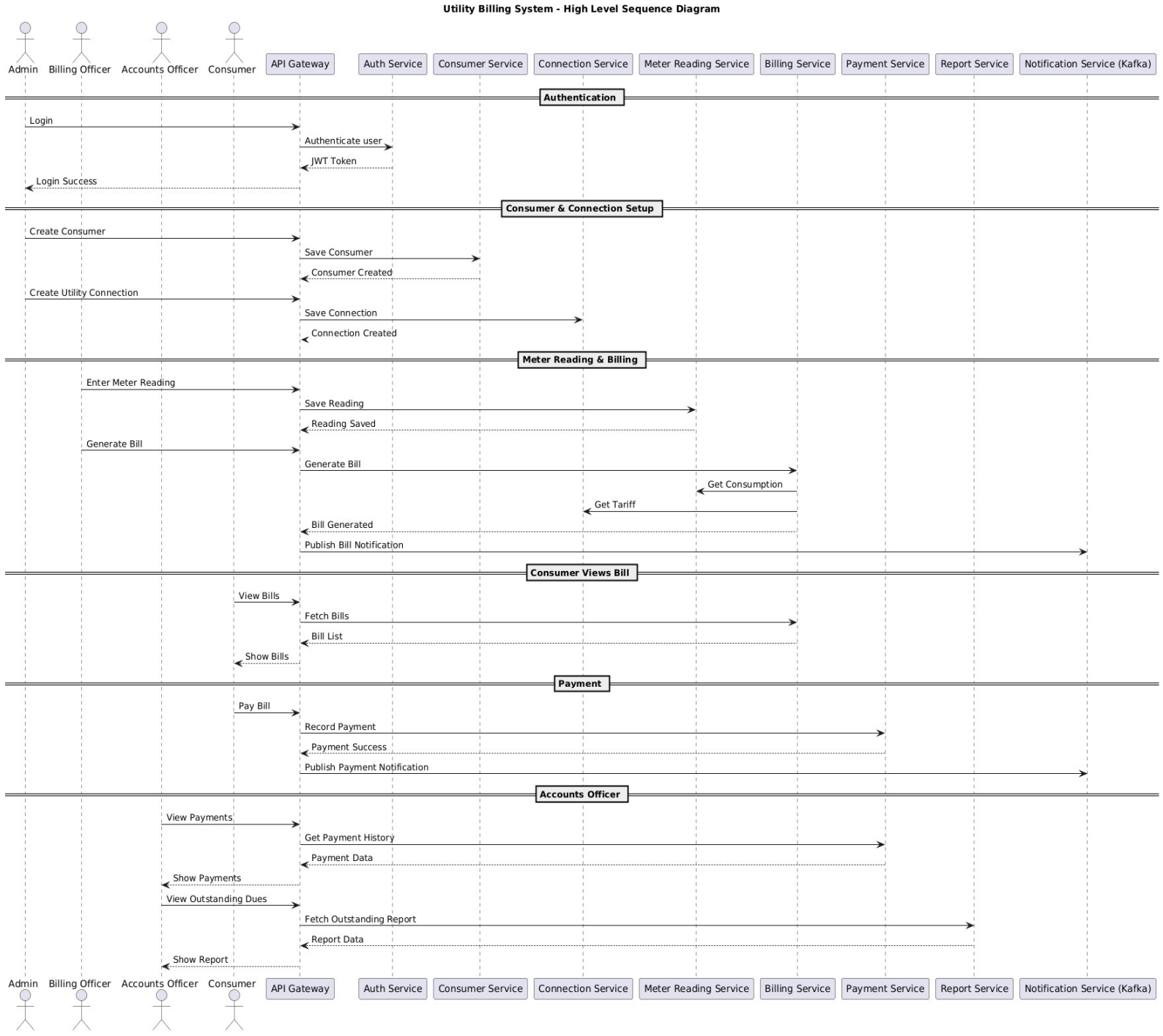
- Spring Cloud Gateway
- Kafka-based async communication

- 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

---

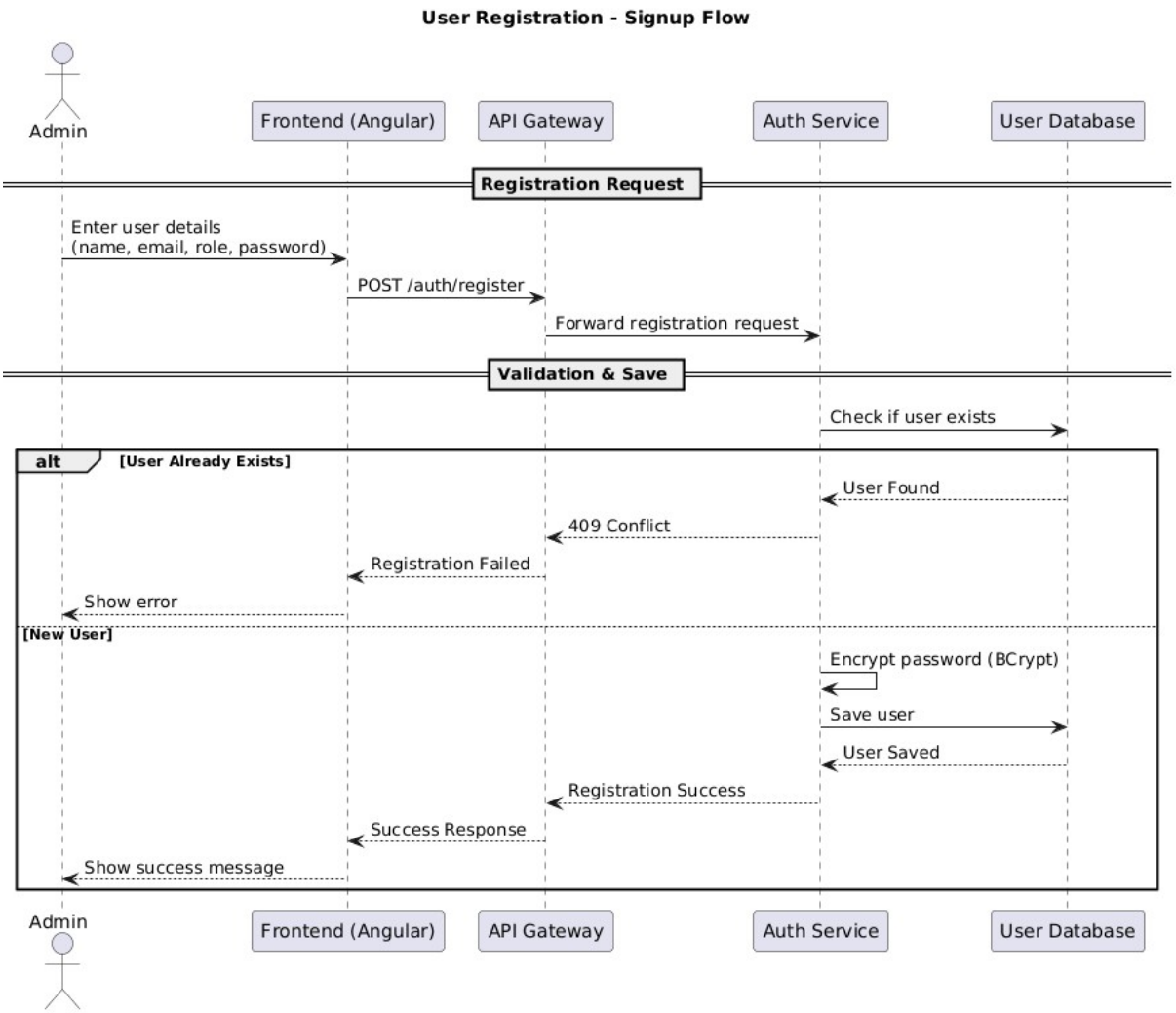
1. User Registration — Sequence Diagram

Scenario

A new user registers using the Angular UI.

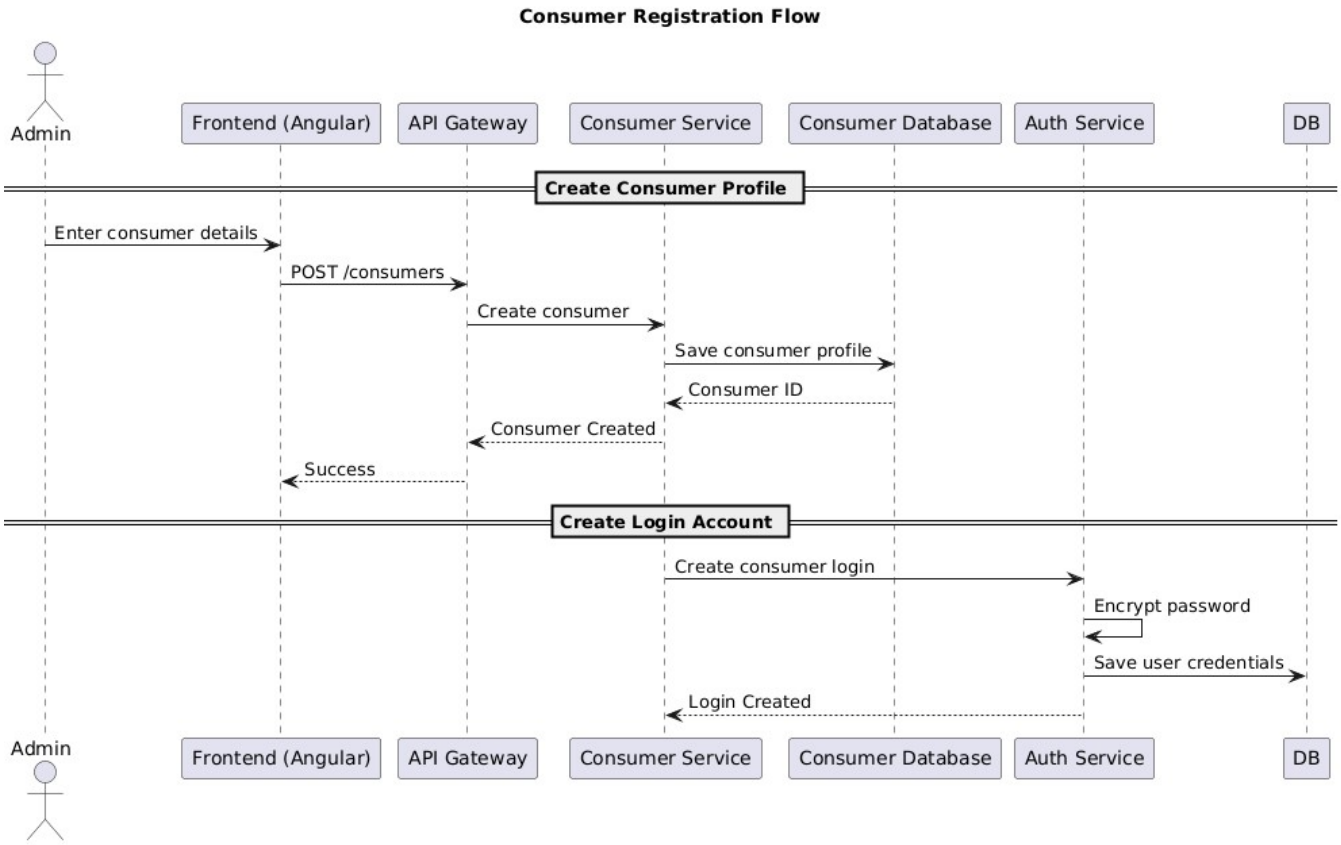
Admin Registration Flow

1. Admin logs into the system
2. Angular sends `POST /users/register` to Auth Service
3. Admin submits registration details for an officer
4. Request goes through API Gateway
5. Auth Service validates and encrypts password
6. User saved with role
7. Success response sent back to UI



Consumer Registration Flow

1. Admin creates consumer profile
2. Consumer Service saves consumer details
3. Consumer Service requests Auth Service to create login
4. Auth Service creates consumer credentials
5. Consumer receives access to login



Participants

- Admin
- Angular UI
- API Gateway
- Auth Service
- Consumer Service
- MongoDB

Key Design Points

- Only **Admin** can register system users
- Roles assigned at registration time
- Passwords encrypted using **BCrypt**
- Stored only in **Auth Service**
- Used for **RBAC (Role-Based Access Control)**
- Consumers are **business entities**, not system users
- Consumer profile stored in **Consumer Service**
- Login credentials stored in **Auth Service**
- Communication between services via **Feign**

---

2. User Login — Sequence Diagram

Scenario

Registered user logs into the system.

Flow

1. User enters username & password
2. Frontend sends login request
3. API Gateway forwards request
4. Auth Service validates credentials
5. JWT token generated
6. Token returned to frontend

Participants

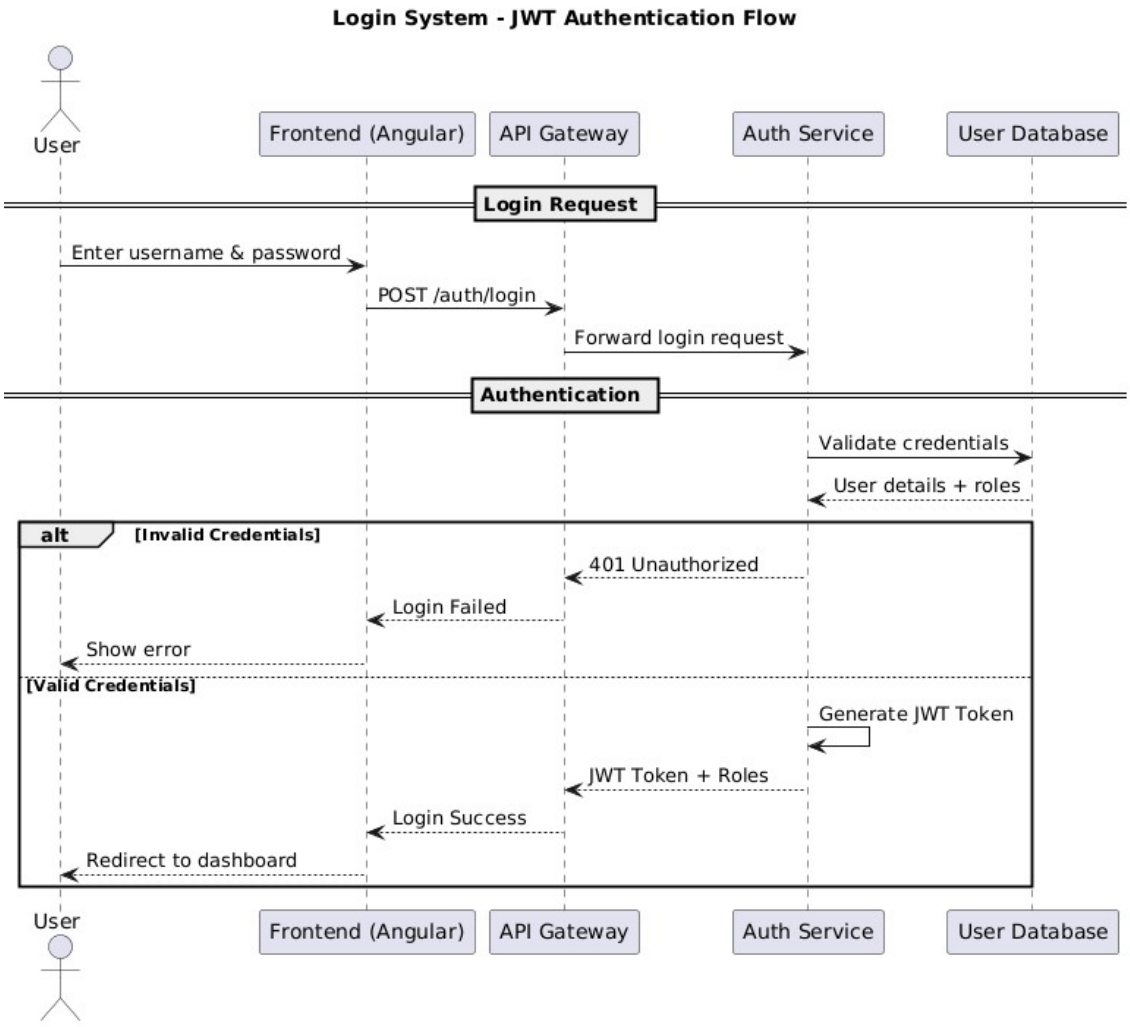
- User (Admin / Officer / Consumer)
- Angular UI
- API Gateway
- Auth Service
- MongoDB

Failure Cases

- Invalid credentials → 401
- Inactive user → 403

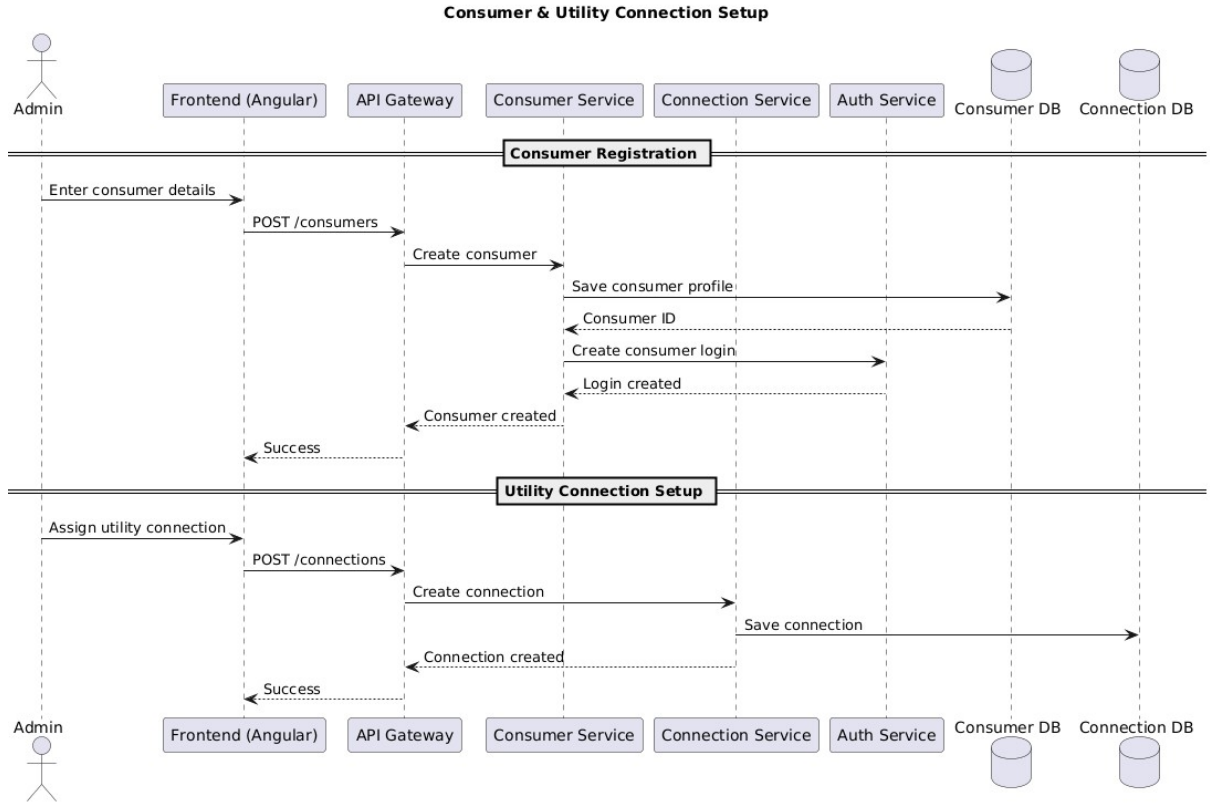
Participants

- Single login endpoint for all roles
- JWT contains:
  1. User ID
  2. Role
  3. Expiry time
- Stateless authentication
- Token used in all secured APIs
- Gateway validates token on every request



3.

3.Consumer & Connection - Sequence Diagram



Scenario

To register a **consumer (customer)** and assign a **utility connection** with tariff and meter details.

Flow

- **Consumer Registration**

1. Admin enters consumer details
2. Request goes through API Gateway
3. Consumer Service saves consumer profile
4. Consumer ID is generated
5. Consumer login is created in Auth Service

- **Utility Connection Creation**

6. Admin assigns utility connection to consumer
7. Connection Service validates tariff & utility type
8. Connection details saved
9. Connection linked to consumer

Participants

- Admin
- Frontend (Angular)
- API Gateway
- Consumer Service
- Connection Service
- Auth Service (for consumer login)
- Consumer Database
- Connection Database

---

4. Meter Reading - Sequence Diagram

Scenario

Billing Officer enters monthly meter readings for consumers.
The system validates readings, calculates consumption, and stores the data.

Flow

1. Billing Officer opens meter reading entry page
2. Inputs consumer’s meter reading
3. Frontend sends request via API Gateway
4. Gateway forwards request to Meter Reading Service
5. Meter Reading Service validates:
6. Connection exists

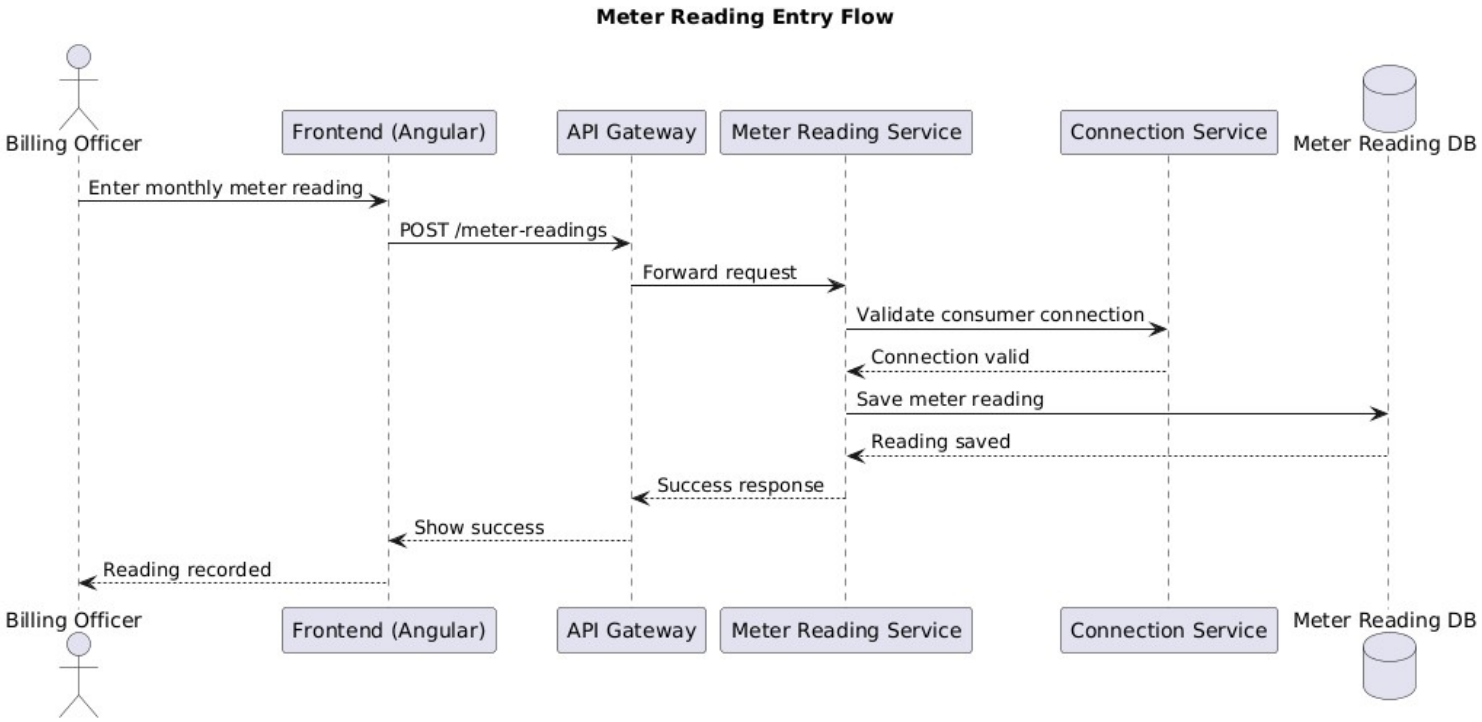
7. Reading is higher than previous month
8. If valid, save reading in DB
9. Response sent back to frontend (success/failure)

Participants

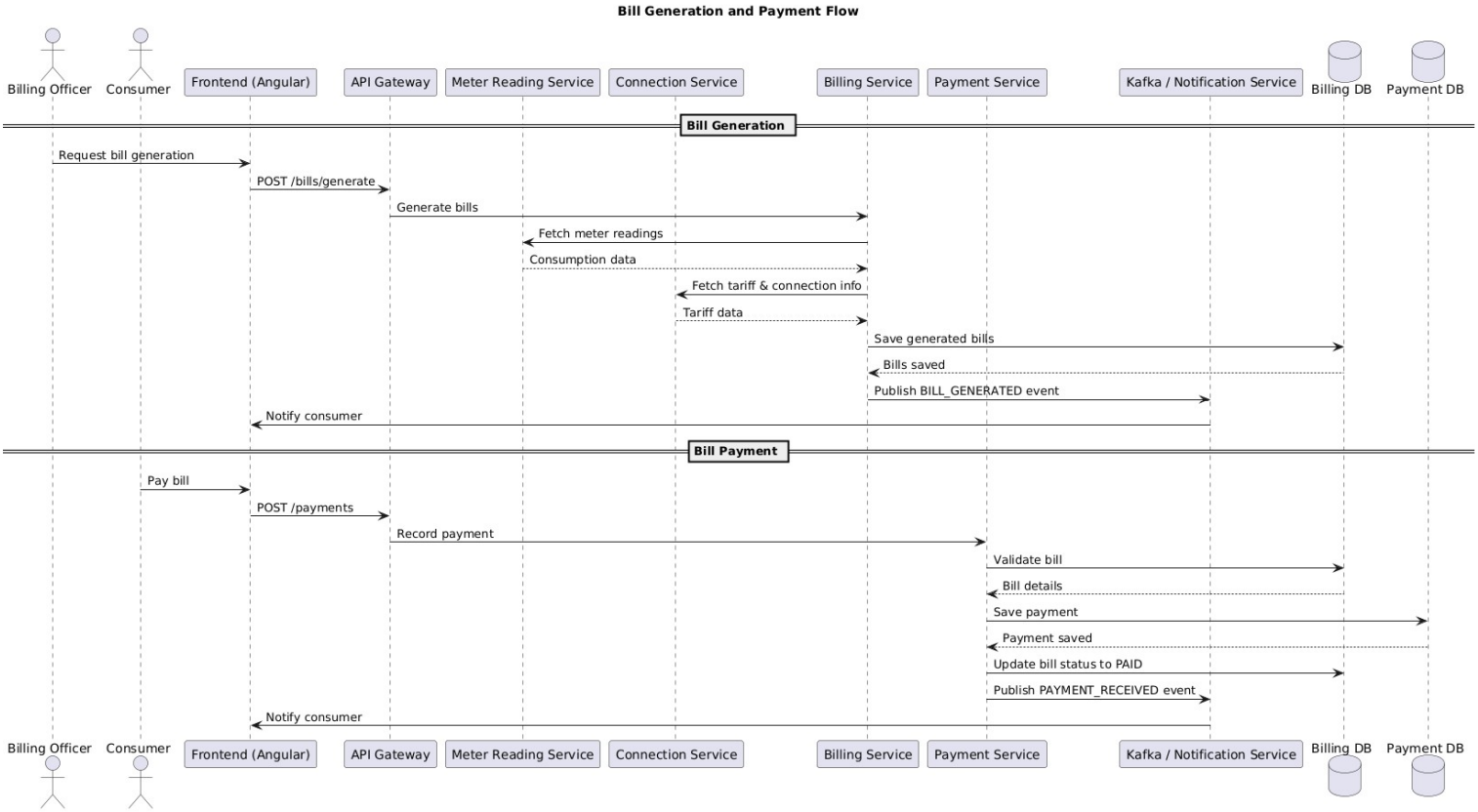
- Billing Officer (actor)
- Frontend (Angular)
- API Gateway
- Meter Reading Service
- Connection Service (for validation)
- Meter Reading Database

Key Design Decisions

- Only **Billing Officer** can enter readings
- Readings **validated against previous month** to avoid errors
- **Separate service** for meter readings → clean microservice design
- Connection Service used to **validate active connection**
- Stored in **Meter Reading DB**
- Gateway ensures **JWT authentication**
- Later used for **bill generatio**



5. Bill Generation & Payment - Sequence Diagram



Scenario

- Generate monthly bills based on meter readings, tariffs, and fixed charges
- Record payments, update bill status, and trigger notifications

Flow

Bill Generation (Billing Officer)

1. Billing Officer requests bill generation for a month
2. API Gateway forwards to Billing Service
3. Billing Service fetches:
    - Meter readings from Meter Reading Service
    - Connection & tariff info from Connection Service
4. Billing Service calculates bills (consumption × tariff + fixed charges + tax)
5. Bills saved in Billing DB
6. Billing Service publishes **BILL_GENERATED** event to Kafka
7. Notification Service consumes event → sends bill notification to consumer

Bill Payment (Consumer)

1. Consumer pays bill via frontend
2. API Gateway forwards request to Payment Service

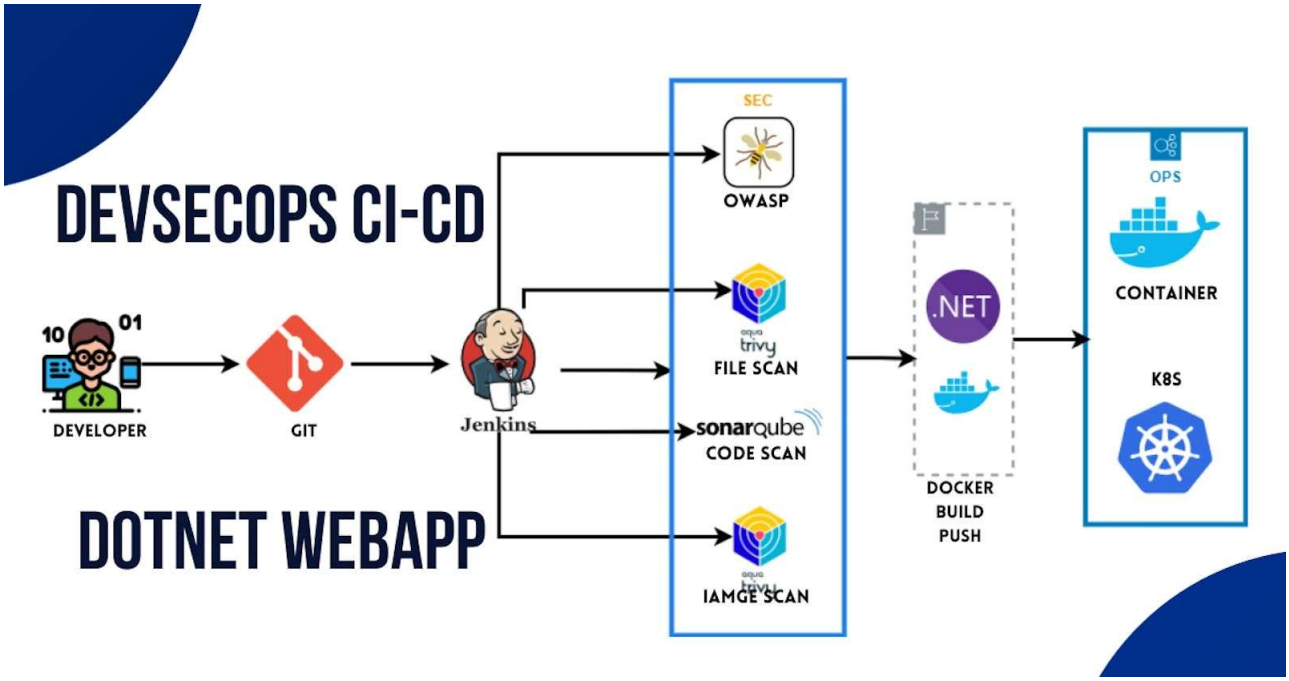
3. Payment Service validates bill, records payment, updates status to PAID
4. Payment Service publishes **PAYMENT_RECEIVED** event to Kafka
5. Notification Service sends payment receipt to consumer
6. Accounts Officer can view updated payments and outstanding dues

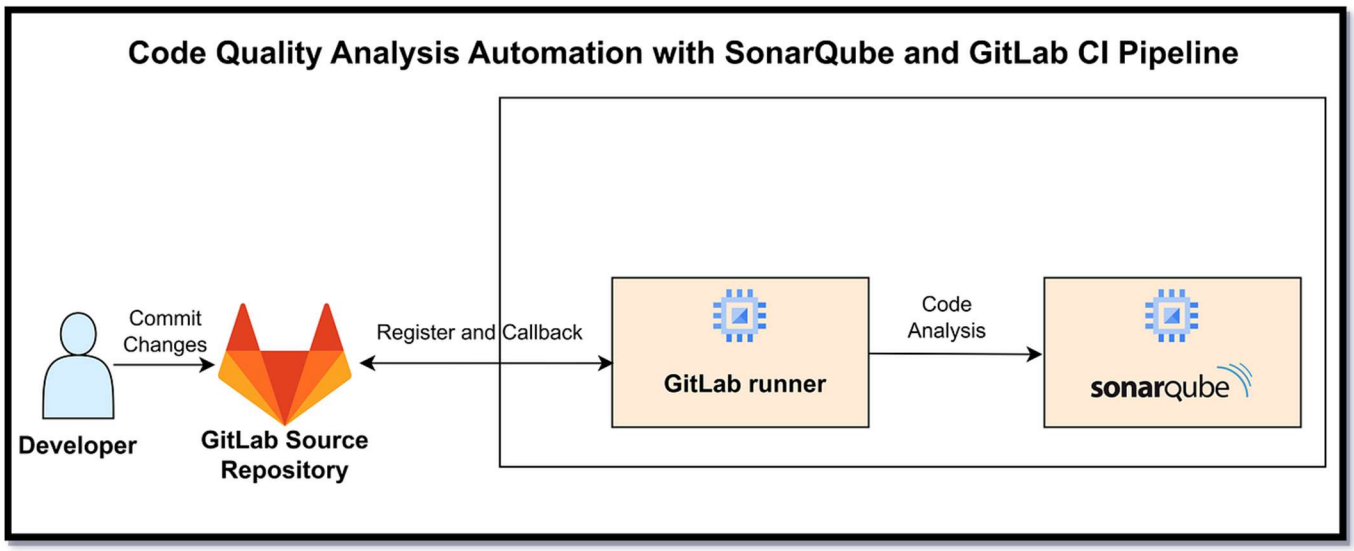
Participants

- Billing Officer – generates bills
- Consumer – pays bills
- Frontend (Angular)
- API Gateway
- Meter Reading Service – provides consumption
- Connection Service – provides tariff & cycle
- Billing Service – calculates bills
- Payment Service – records payments
- Kafka / Notification Service – sends bill/payment notifications
- Databases – for bills, payments

---

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
```

---

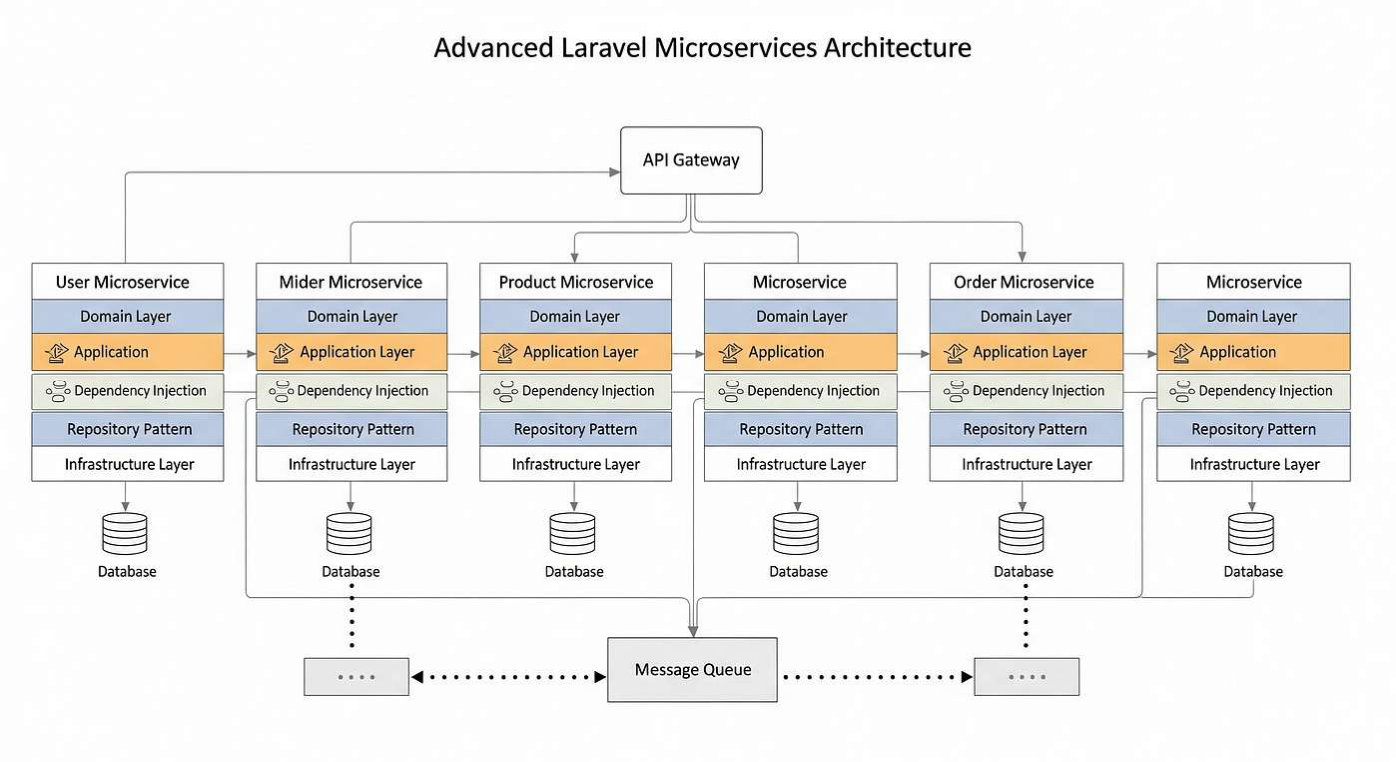
7. DEVOPS & CI/CD FILES

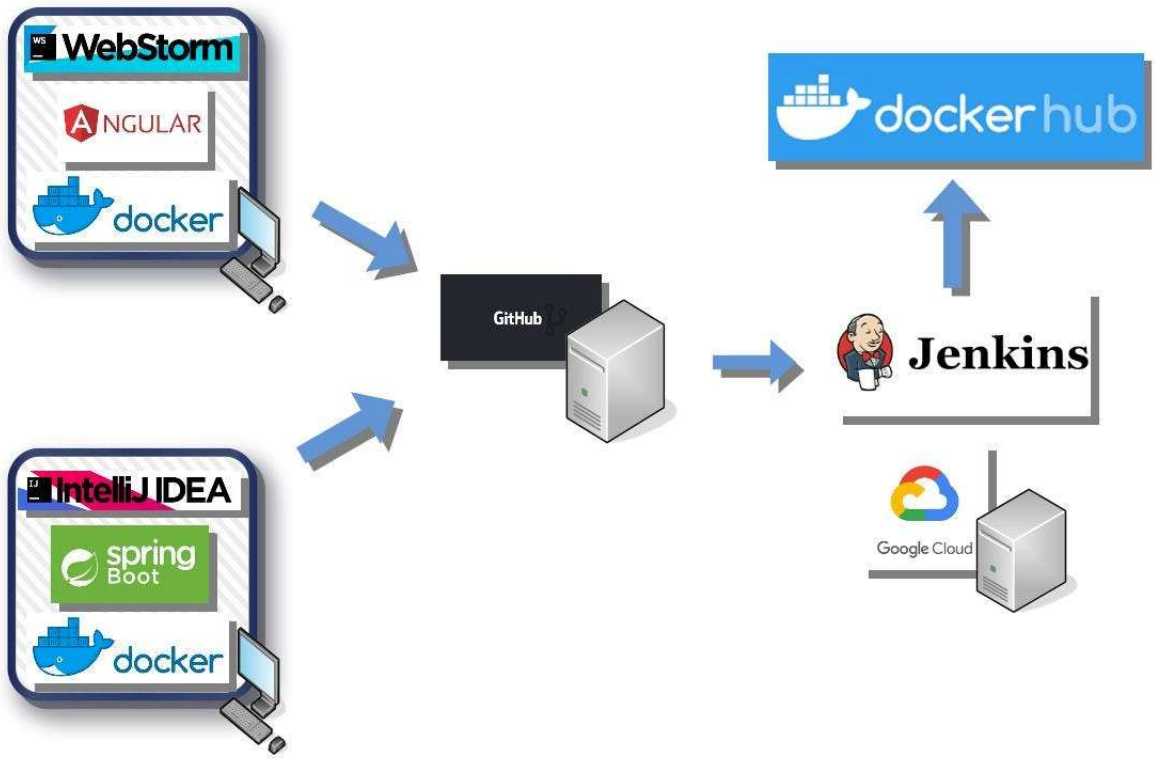
```
capstone-project
├─ Jenkinsfile
├─ docker-compose.yml
├─ user-service/Dockerfile
├─ product-service/Dockerfile
├─ order-service/Dockerfile
├─ angular-ui/Dockerfile
└─ README.md
```

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.