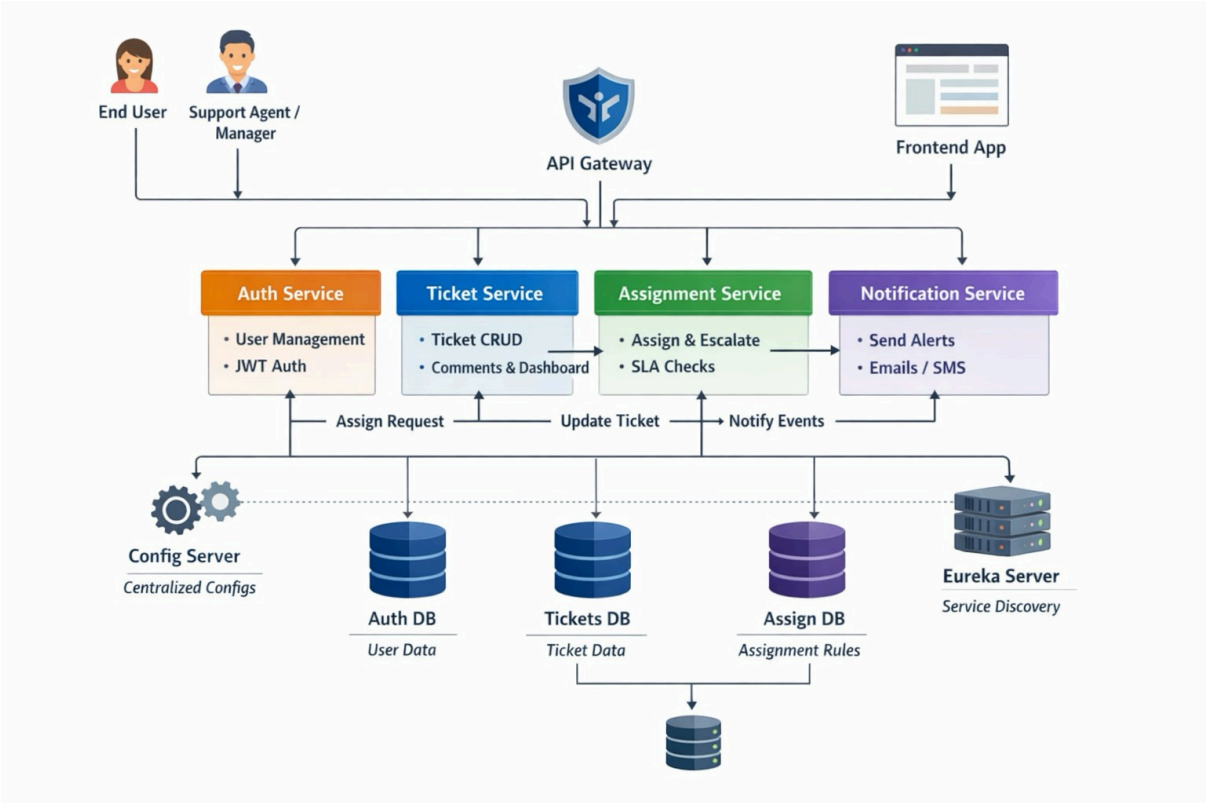
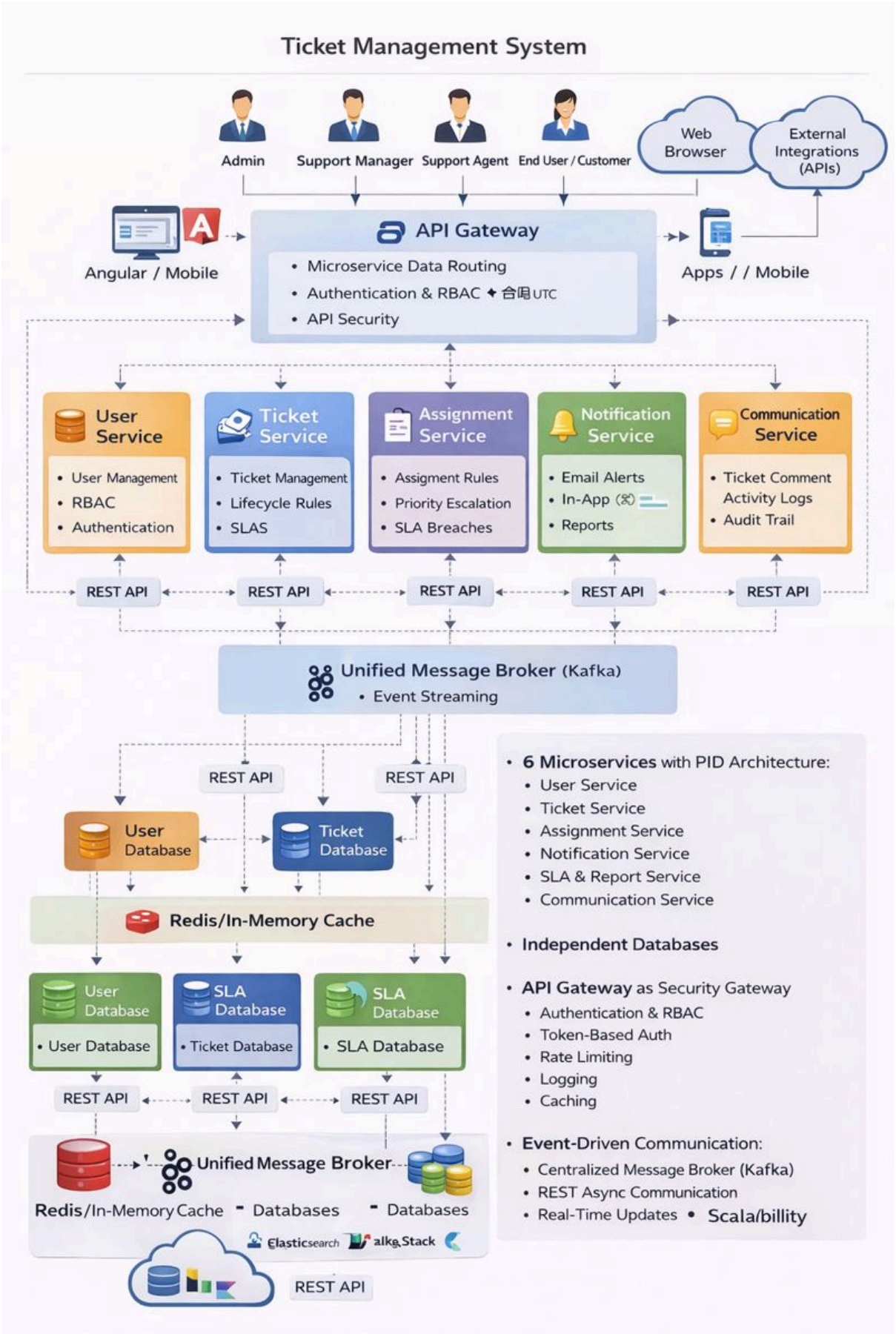
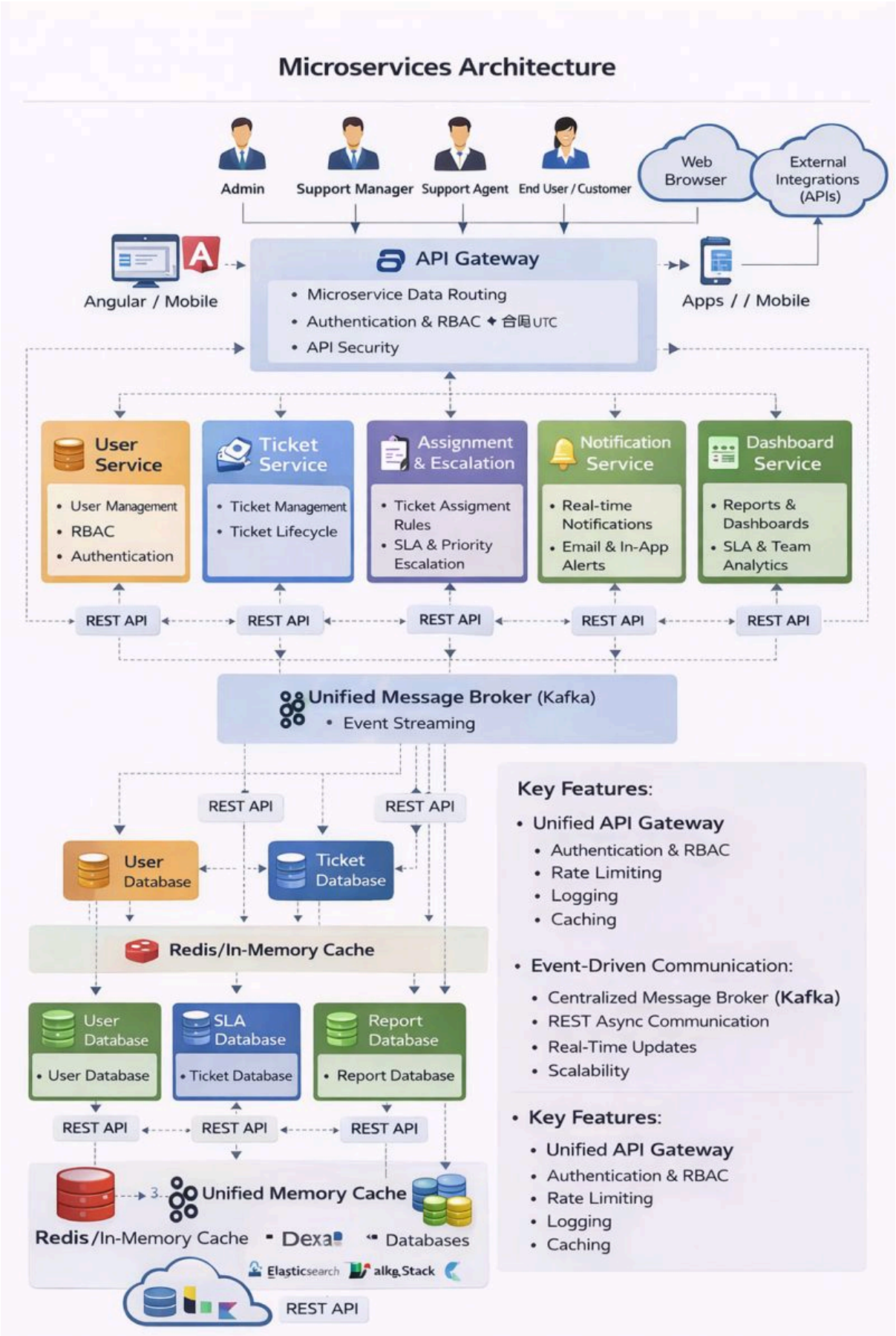
SOFTWARE DESIGN DOCUMENT(SDD)

Smart Ticket & Issue Management System




Ticket Management System
Admin
Support Manager
Support Agent
End User / Customer
Web Browser
External Integrations (APIs)
Angular / Mobile
API Gateway
• Microservice Data Routing
• Authentication & RBAC
• API Security
Apps / / Mobile
User Service
• User Management
• RBAC
• Authentication
Ticket Service
• Ticket Management
• Lifecycle Rules
• SLAS
Assignment Service
• Assigment Rules
• Priority Escalation
• SLA Breaches
Notification Service
• Email Alerts
• In-App
• Reports
Communication Service
• Ticket Comment Activity Logs
• Audit Trail
REST API
REST API
REST API
REST API
REST API
Unified Message Broker (Kafka)
• Event Streaming
REST API
REST API
User Database
Ticket Database
Redis/In-Memory Cache
User Database
• User Database
SLA Database
• Ticket Database
SLA Database
• SLA Database
REST API
REST API
REST API
Unified Message Broker
Redis/In-Memory Cache
Databases
Databases
Elasticsearch
alkg Stack
REST API
• 6 Microservices with PID Architecture:
• User Service
• Ticket Service
• Assignment Service
• Notification Service
• SLA & Report Service
• Communication Service
• Independent Databases
• API Gateway as Security Gateway
• Authentication & RBAC
• Token-Based Auth
• Rate Limiting
• Logging
• Caching
• Event-Driven Communication:
• Centralized Message Broker (Kafka)
• REST Async Communication
• Real-Time Updates • Scala/billity


Microservices Architecture
Admin
Support Manager
Support Agent
End User / Customer
Web Browser
External Integrations (APIs)
Angular / Mobile
API Gateway
• Microservice Data Routing
• Authentication & RBAC
• API Security
Apps / / Mobile
User Service
• User Management
• RBAC
• Authentication
Ticket Service
• Ticket Management
• Ticket Lifecycle
Assignment & Escalation
• Ticket Assigment Rules
• SLA & Priority Escalation
Notification Service
• Real-time Notifications
• Email & In-App Alerts
Dashboard Service
• Reports & Dashboards
• SLA & Team Analytics
REST API
REST API
REST API
REST API
REST API
Unified Message Broker (Kafka)
• Event Streaming
REST API
REST API
User Database
Ticket Database
Redis/In-Memory Cache
User Database
• User Database
SLA Database
• Ticket Database
Report Database
• Report Database
REST API
REST API
REST API
Unified Memory Cache
Redis/In-Memory Cache
Dexa
Databases
Elasticsearch
alkg Stack
REST API
Key Features:
• Unified API Gateway
• Authentication & RBAC
• Rate Limiting
• Logging
• Caching
• Event-Driven Communication:
• Centralized Message Broker (Kafka)
• REST Async Communication
• Real-Time Updates
• Scalability
• Key Features:
• Unified API Gateway
• Authentication & RBAC
• Rate Limiting
• Logging
• Caching

---

1. DOCUMENT CONTROL

Item	Details
Project Name	Smart Ticket & Issue Management System
Version	1.0
Author	Varakantham Pranitha
Date	27-12-2025
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 Ticket & Issue 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 tickets (creation, update, resolution, closure, escalation)
- Assign tickets manually or automatically to agents
- Enforce SLA rules and escalation policies
- Provide dashboards for Admins, Managers, Agents, and End-Users

High-Level Features

- User Registration & Secure Login
- Real-time Ticket creation & tracking
- Manual and auto assignment
- SLA enforcement & escalation
- Role-based access (USER / ADMIN / MANAGER / AGENT)

4. ARCHITECTURE OVERVIEW (HLD)

Image need to add

Architecture Style

- Microservices Architecture
- REST-based communication
- Containerized deployment

Core Components

1. Angular Frontend
2. API Gateway (optional)
3. Auth-User Service
4. Ticket Service
5. Assignment-Escalation Service
6. Notification Service (Kafka + SMTP)
7. Dashboard Service (aggregates stats via Feign)
8. MongoDB (per service)
9. CI/CD Pipeline

5. TECHNOLOGY STACK

Backend

- Java 17
- Spring Boot
- Spring WebFlux (reactive)
- Spring Data MongoDB
- Spring Validation

Frontend

- Angular
- Bootstrap / Angular Material
- RxJS
- Chart.js (via ng2-charts) for dashboard visualizations

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-User Service

Responsibilities

- User registration & authentication
- Role management (Admin, Manager, Agent, User)
- Password reset/change flows

APIs

```
POST /auth/register -> Register new user
POST /auth/login -> Authenticate User
POST /auth/change-password -> Change password with old/new
POST /auth/reset-password -> Reset password with token
POST /auth/request-reset -> Request Password Reset
PUT /users/{id} -> Update user details
GET /users -> List all users
GET /users/{id} -> Get user by ID
GET /auth/{id} -> Fetch user email by ID
DELETE /auth/{id} -> Delete user by ID (Soft delete)
```

Database

- users collection

6.2 Ticket Service

Responsibilities

- Ticket CRUD operations
- Lifecycle Management, Status transitions
  (OPEN → IN_PROGRESS → RESOLVED → CLOSED → REOPENED)
- Reopen/Cancel
- Category management
- Ticket activity timeline (comments, actions, SLA reports)

APIs

```
POST  /tickets/create -> Create Ticket
PUT  /tickets/{id} -> Update Ticket
PUT  /tickets/{id}/close -> Close Ticket
PUT  /tickets/{id}/resolve -> Resolve Ticket
PUT  /tickets/{id}/reopen -> Reopen Ticket
GET  /tickets/{id} -> Get Tickets by ID
GET  /tickets/user/{userId} -> Get Tickets by User ID
GET  /tickets/recent -> Get recent tickets
GET  /tickets/recent/{userId} -> Get recent tickets by user ID
DELETE  /tickets/{id} -> Delete ticket by ID
```

Database

- tickets collection
- ticket_activities collection
- categories collection

6.3 Assignment-Escalation Service

Responsibilities

- Manual assignment of Tickets
- Auto assignment (least workload → hybrid skill+workload)
- SLA Rules enforcement & escalation checks
- Reassignment logic

APIs

```
POST  /assignments/manual -> Manual assignment
POST  /assignments/{ticketId}/auto -> Auto assignment
PUT  /assignments/{ticketId}/complete -> Complete assignment
PUT  /assignments/{ticketId}/check-escalation -> Manual assignment
```

Database

- assignment collection
- sla_rules collection

6.4 Notification Service

Responsibilities

- Send email notifications (assignment, escalation, resolution etc)
- Kafka event publishing for async communication
- Notification history per user

APIs

```
POST /notifications/send -> Send Notification
GET /notifications/status/{id} -> Get notification status
GET /notifications/history/{userEmail} -> Get notification history
```

Database

- notifications collection

6.5 Dashboard Service

Responsibilities

- Aggregate stats from Ticket & Assignment services via Feign clients
- Provide unified dashboard endpoints for Admin/Manager views

---

6.5 Api Gateway

Responsibilities

- Forwarding api points

---

7. DATA DESIGN (LLD)

User Document

```
{
  "id": "u101",
  "displayId": "USR-001",
  "username": "agent_john",
  "password": "encrypted",
  "email": "john.doe@test.com",
  "roles": ["AGENT", "USER"],
  "enabled": true,
  "passwordLastChanged": "2026-01-01T09:00:00",
  "resetToken": "abc123token",
  "resetTokenExpiry": "2026-01-01T10:00:00Z",
  "agentLevel": "L1"
}
```

Purpose: Stores user identity, roles, and security metadata.

Ticket Document

```
{
  "id": "t201",
  "displayId": "TCK-001",
  "title": "Login issue",
  "description": "Unable to login with correct credentials",
  "categoryId": "c101",
  "status": "OPEN",
  "priority": "HIGH",
  "createdBy": "u101",
  "assignedTo": "u202",
  "createdAt": "2026-01-01T10:00:00",
  "updatedAt": "2026-01-01T10:05:00"
}
```

Purpose: Represents ticket lifecycle and metadata.

TicketActivity Document

```
{
  "id": "a301",
  "ticketId": "t201",
  "actorId": "u202",
  "actionType": "COMMENT",
  "details": "Investigating login issue",
  "timestamp": "2026-01-01T10:15:00Z"
}
```

Purpose: Logs timeline of actions on a ticket.

Category Document

```
{
  "id": "c101",
  "name": "Authentication",
  "description": "Issues related to login and password",
  "linkedSlaId": "sla001",
  "active": true
}
```

Purpose: Defines ticket categories and links to SLA rules.

Assignment Document

```
{
  "id": "as501",
  "ticketId": "t201",
  "agentId": "u202",
  "assignedAt": "2026-01-01T10:05:00Z",
  "dueAt": "2026-01-01T12:05:00Z",
  "unassignedAt": null,
  "breached": false,
  "breachedAt": null,
  "status": "ASSIGNED",
  "type": "AUTO",
  "escalationLevel": 0
}
```

Purpose: Tracks assignment lifecycle and SLA compliance.

SlaRule Document

```
{
  "id": "sla001",
  "priority": "HIGH",
  "responseMinutes": 30,
  "resolutionMinutes": 120
}
```

Purpose: Defines SLA rules per priority.

Notification Document

```
{
  "id": "n701",
  "recipient": "john.doe@test.com",
  "subject": "Ticket Assigned",
  "body": "Ticket TCK-001 has been assigned to you.",
  "eventType": "ASSIGNMENT",
  "createdAt": "2026-01-01T10:06:00Z"
}
```

Purpose: Stores notifications sent to users.

Dashboard DTOs

StatusSummaryDto

```
{
  "status": "OPEN",
  "count": 15
}
```

PrioritySummaryDto

```
{
  "priority": "HIGH",
  "count": 8
}
```

CategorySummaryDto

```
{
  "categoryId": "c101",
  "count": 12
}
```

AgentSummaryDto

```
{
  "agentId": "u202",
  "assignedCount": 10,
  "resolvedCount": 7,
  "overdueCount": 2,
  "escalationLevel": 1,
  "averageResolutionTimeMinutes": 45.5
}
```

Purpose: Lightweight aggregation objects for dashboard visualizations.

---

8. API DESIGN & VALIDATION

- **RESTful APIs**: All services expose endpoints following REST conventions (`GET`, `POST`, `PUT`, `DELETE`).
- **JSON Payloads**: Requests and responses use JSON format for interoperability.
- **Reactive Programming**: APIs return `Mono` or `Flux` for non-blocking I/O.
- Service-level business rule enforcement
- Standard HTTP status codes

  `200 OK` → Successful read/update
  `201 CREATED` → Resource created
  `400 BAD REQUEST` → Validation or business rule violation
  `401 UNAUTHORIZED` → Invalid credentials
  `403 FORBIDDEN` → Role/permission denied
  `404 NOT FOUND` → Resource not found
  `409 CONFLICT` → Duplicate resource (e.g., user already exists)
  `500 INTERNAL SERVER ERROR` → Unexpected system error

---

9. ERROR HANDLING STRATEGY

Global Exception Handling

- Centralized using `@ControllerAdvice`
- Converts exceptions into standard JSON error responses.

```
{
  "timestamp": "2026-01-01T10:00:00",
  "status": 400,
  "error": "Validation Error",
  "message": "Priority must be one of [LOW, MEDIUM, HIGH, CRITICAL]"
}
```

---

10. SECURITY DESIGN

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

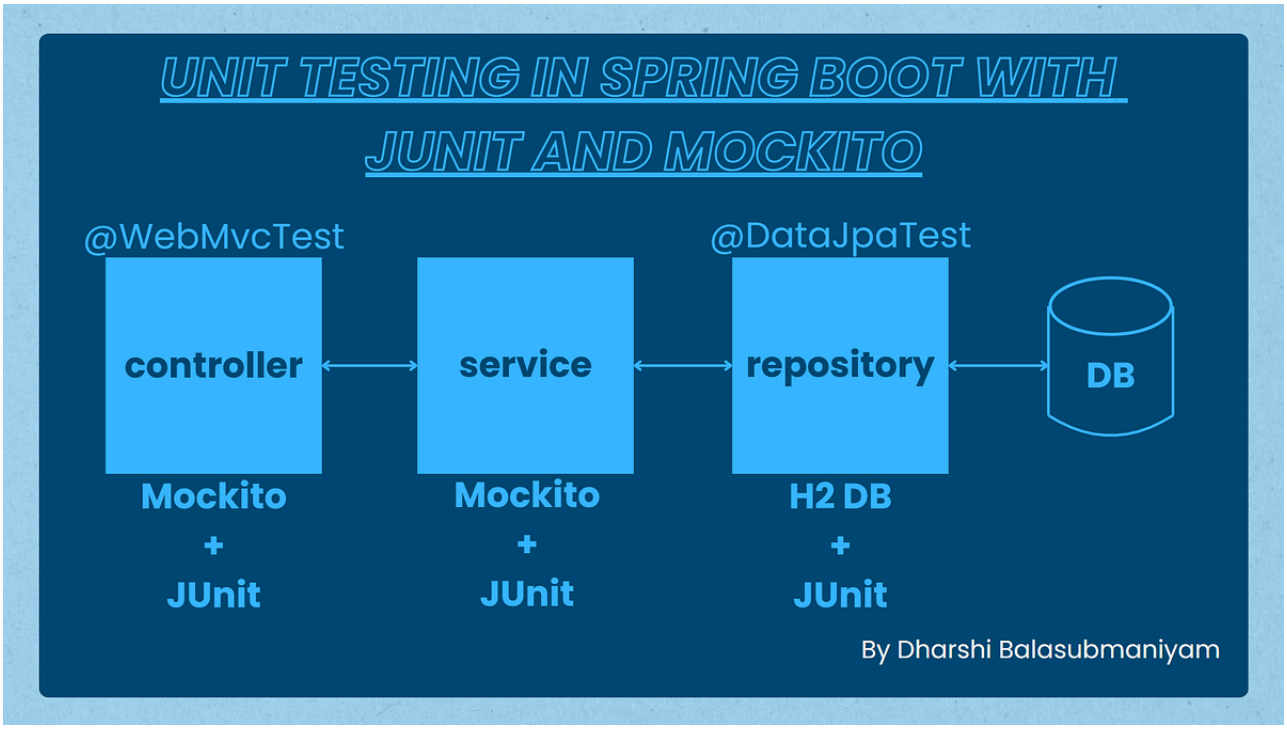
---

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

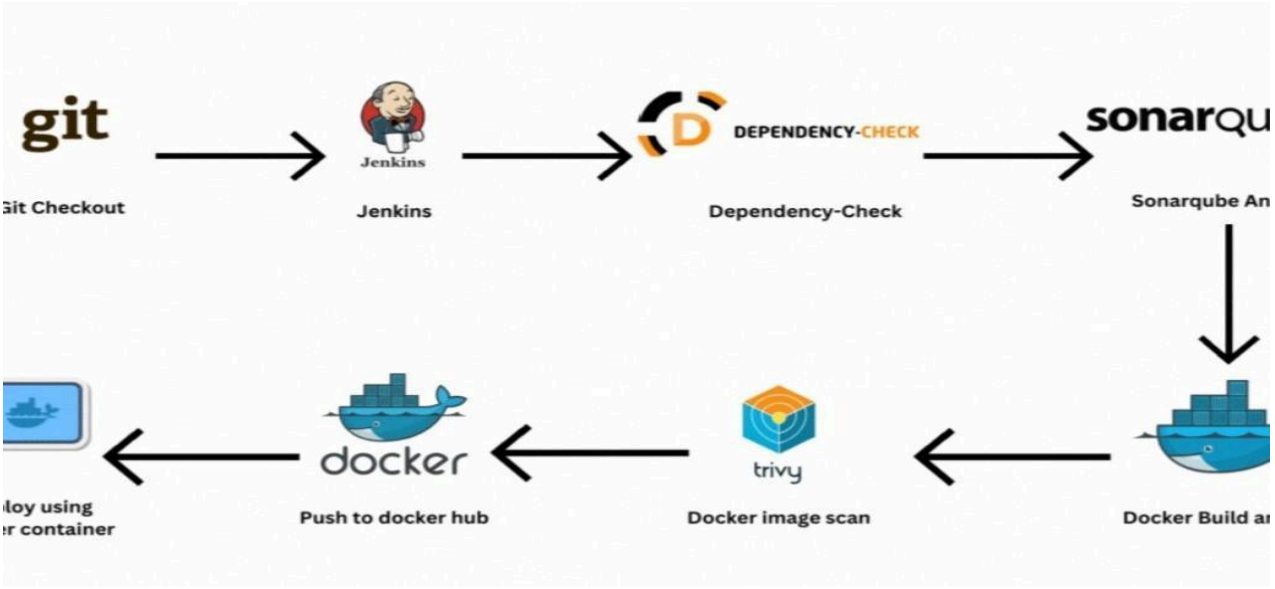
- Component tests
- Service tests

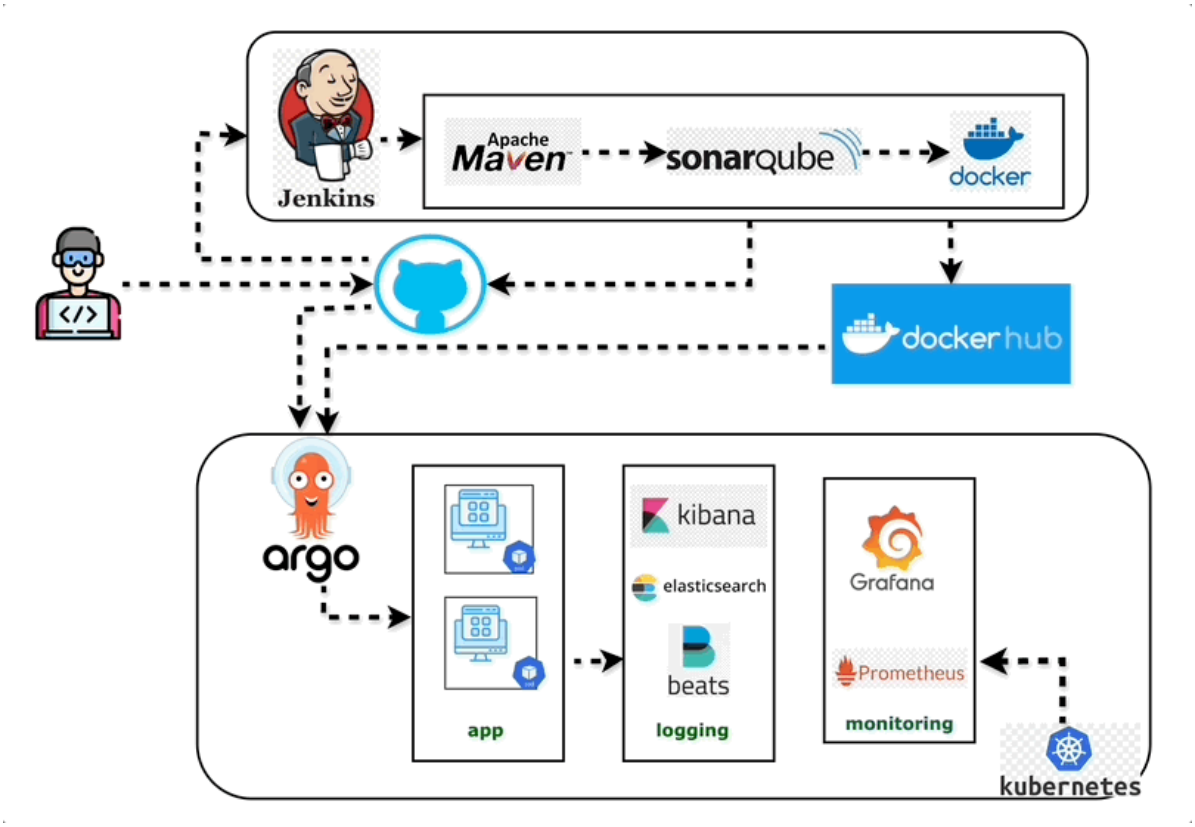
Quality Gates

- 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

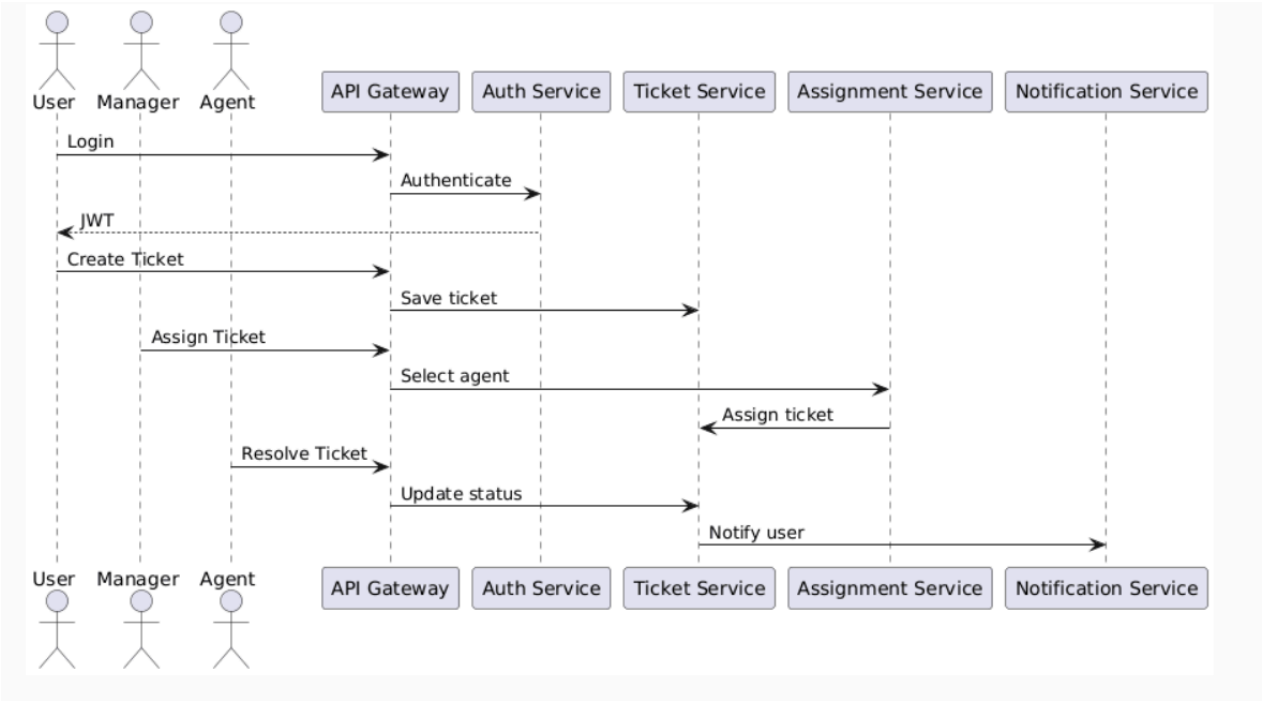
- Spring Cloud Gateway
- Kafka-based async communication
- Kubernetes deployment
- Centralized logging (ELK)

---

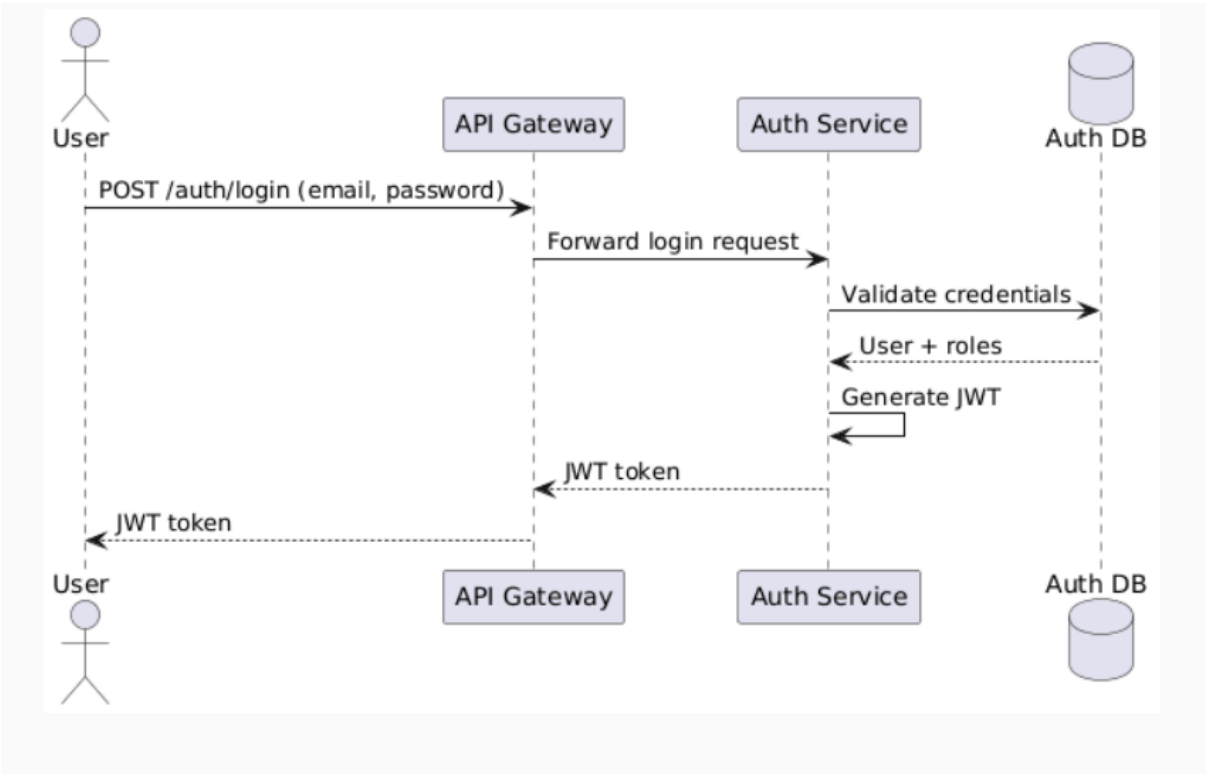
17. CONCLUSION

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

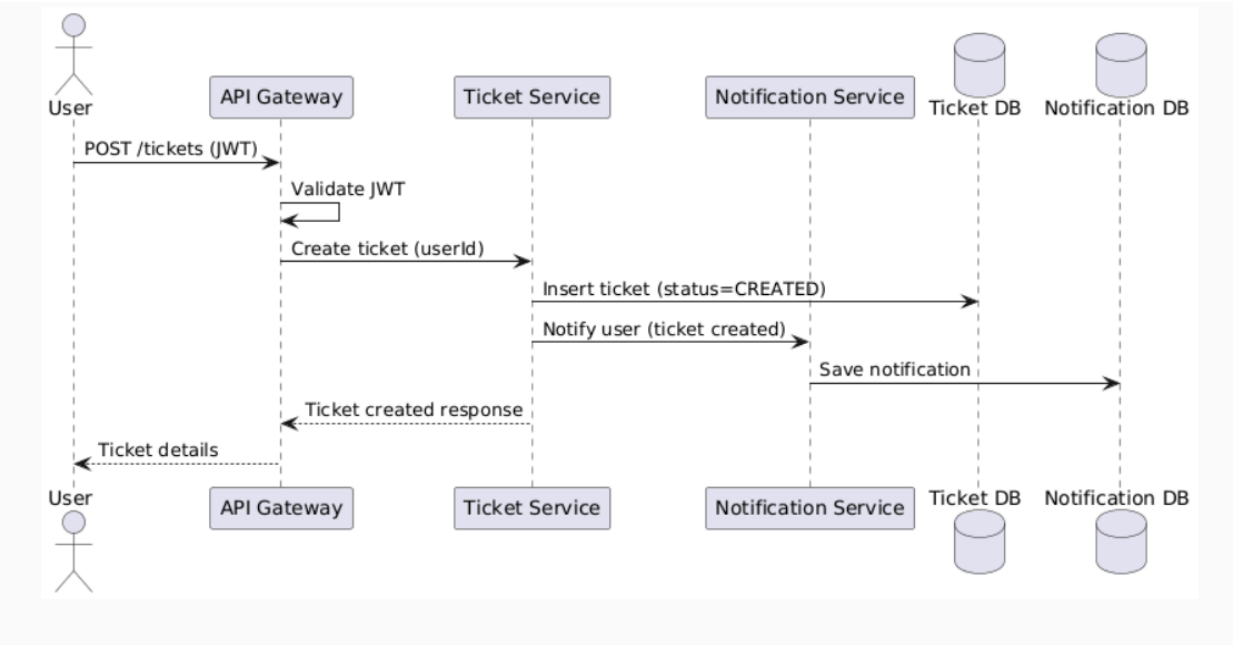
SEQUENCE DIAGRAMS



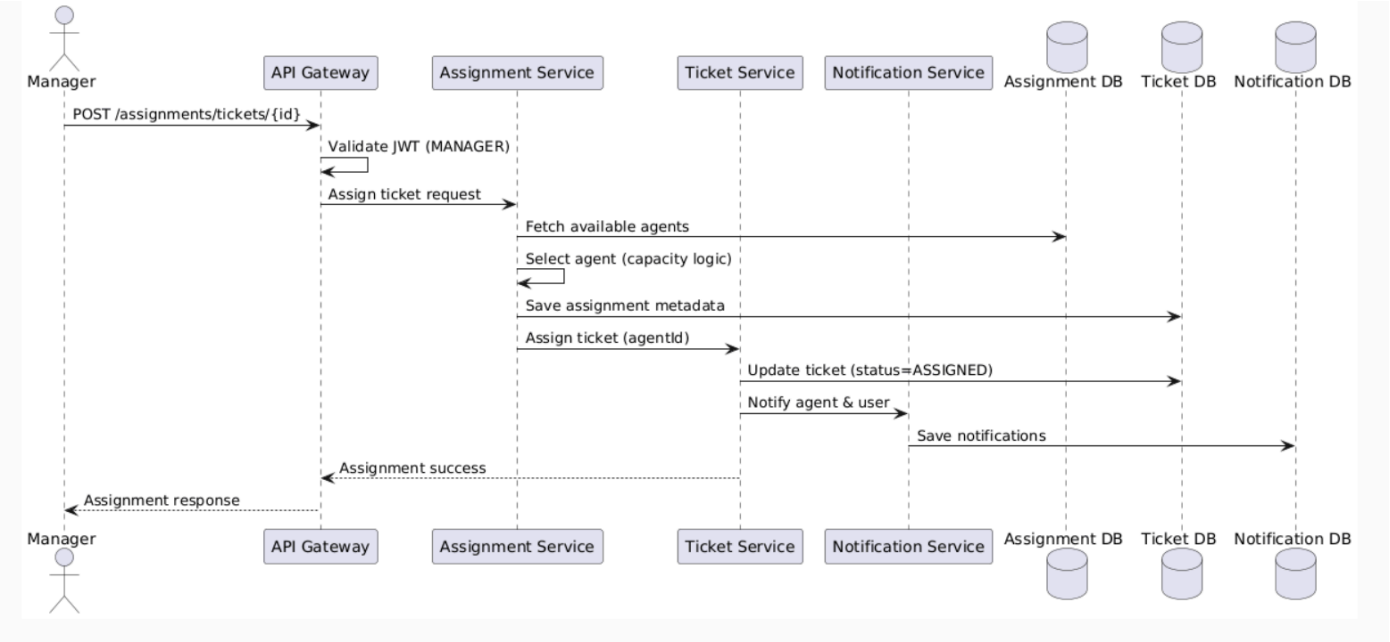
User Login Flow -



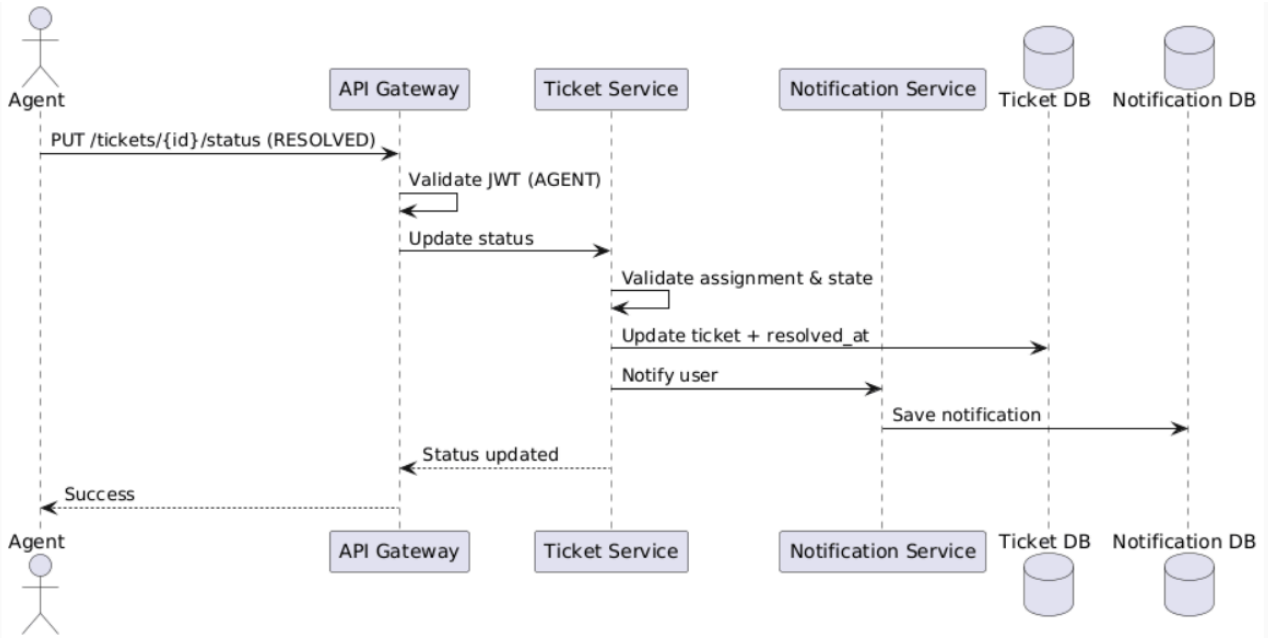
Ticket Creation -

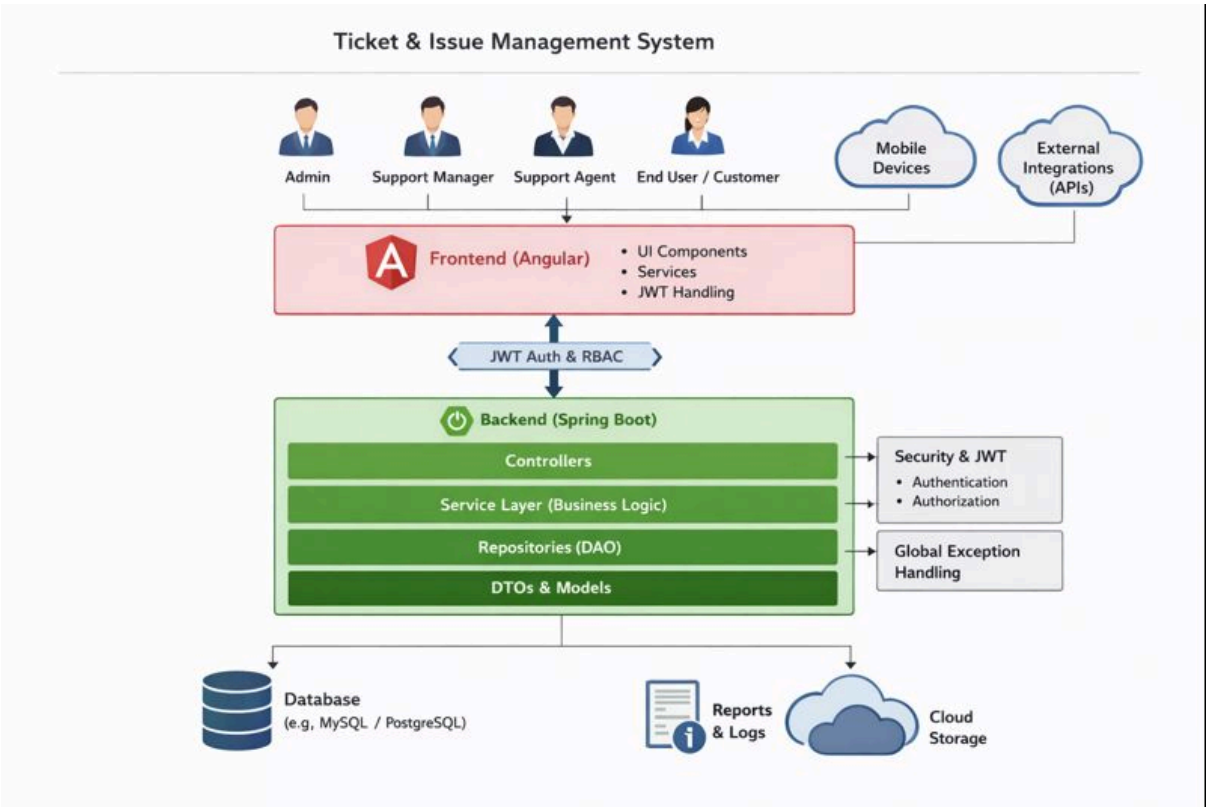


Manager Assigns Ticket to Agent -



Agent Resolves Ticket -


Agent
API Gateway
Ticket Service
Notification Service
Ticket DB
Notification DB
PUT /tickets/{id}/status (RESOLVED)
Validate JWT (AGENT)
Update status
Validate assignment & state
Update ticket + resolved_at
Notify user
Save notification
Status updated
Success
Agent
API Gateway
Ticket Service
Notification Service
Ticket DB
Notification DB


Ticket & Issue Management System
Admin
Support Manager
Support Agent
End User / Customer
Mobile Devices
External Integrations (APIs)
Frontend (Angular)
• UI Components
• Services
• JWT Handling
JWT Auth & RBAC
Backend (Spring Boot)
Controllers
Service Layer (Business Logic)
Repositories (DAO)
DTOs & Models
Security & JWT
• Authentication
• Authorization
Global Exception Handling
Database
(e.g, MySQL / PostgreSQL)
Reports & Logs
Cloud Storage

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

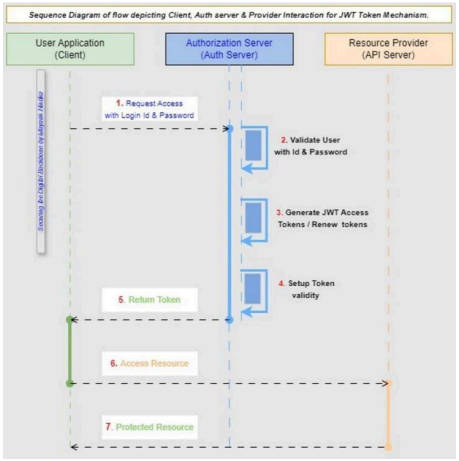
- User
- Angular UI
- User Service
- MongoDB

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. Ticket 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

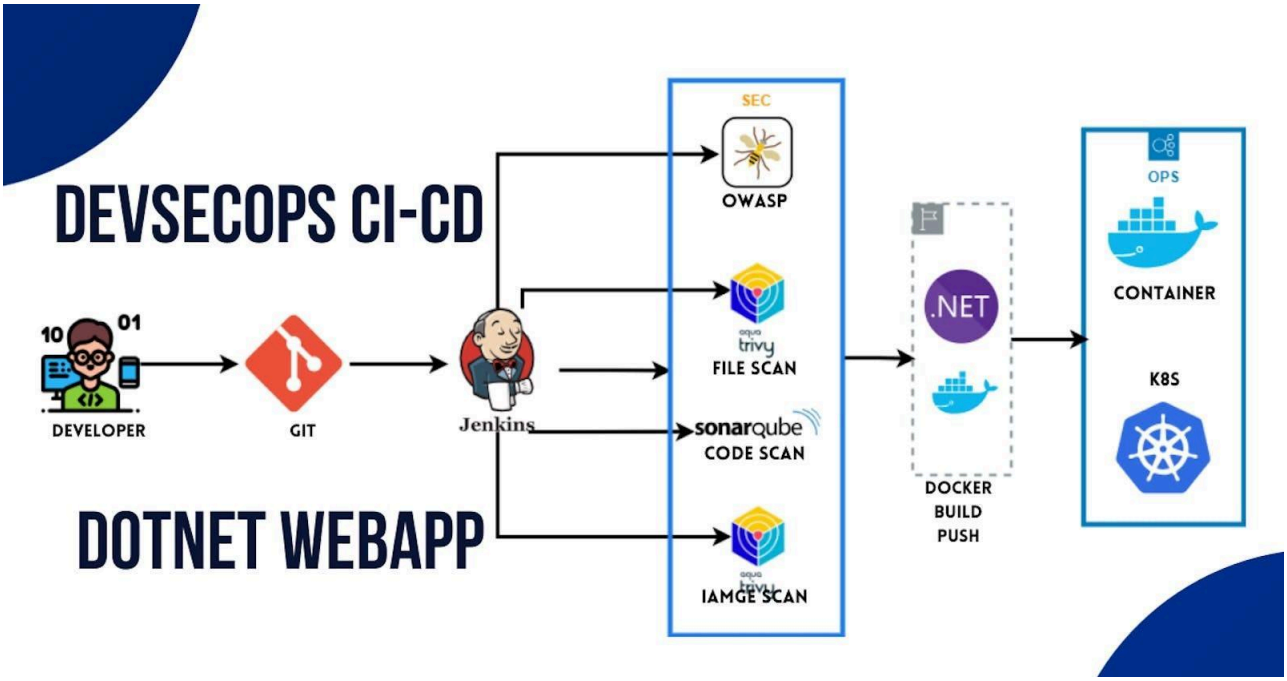
1. Admin sends update request from Angular
2. `PUT /orders/{id}/status` sent to Order Service
3. Role validation (ADMIN only)
4. Order status transition validated
5. Order updated in MongoDB
6. Updated status returned

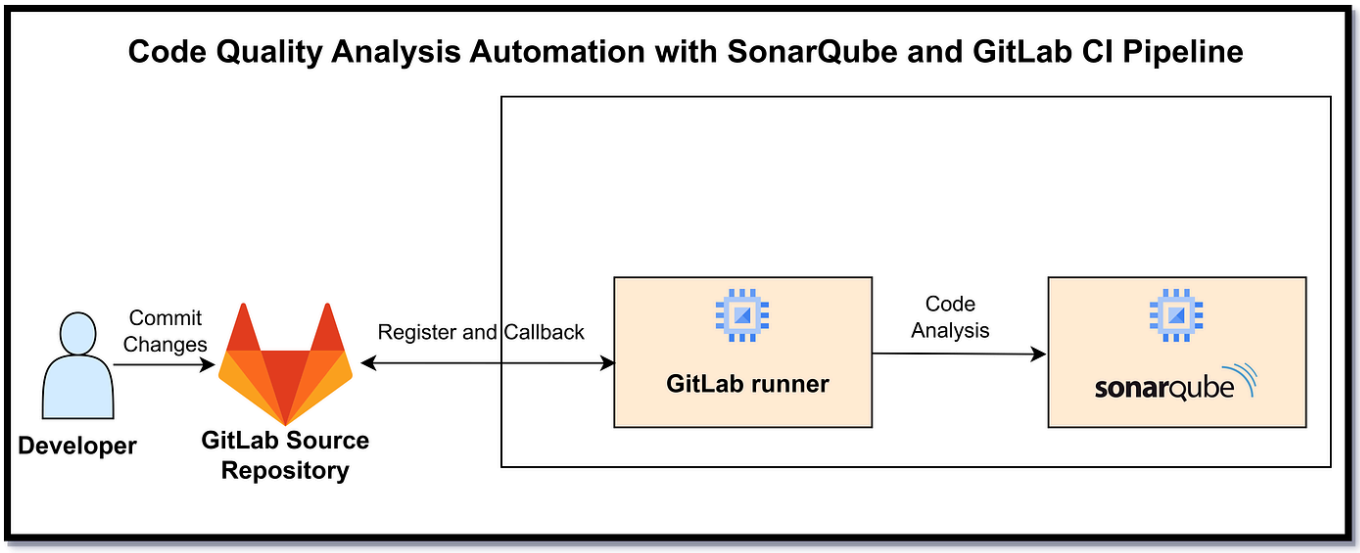
Valid Transitions

- 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

Business Rules

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

Step	Service	Class	Responsibility
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
```

---

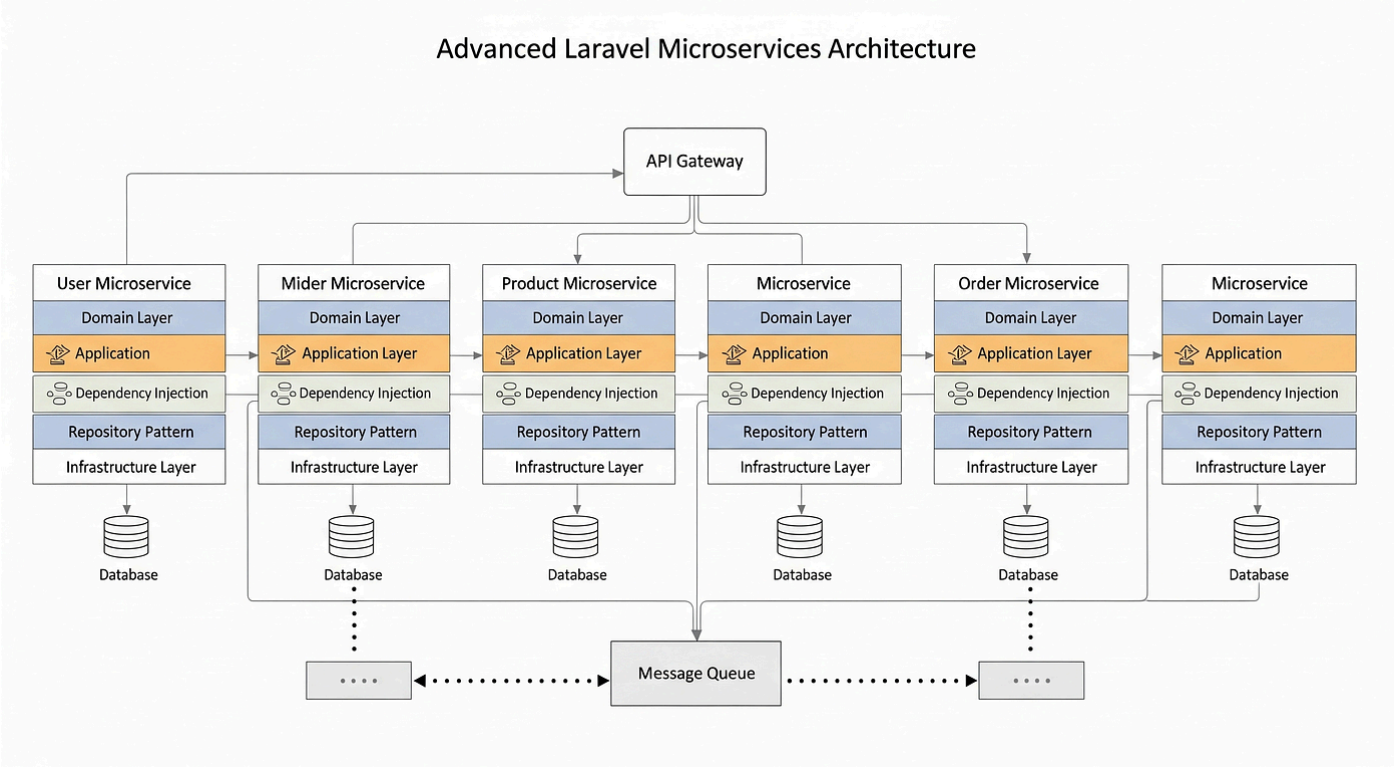
7. DEVOPS & CI/CD FILES

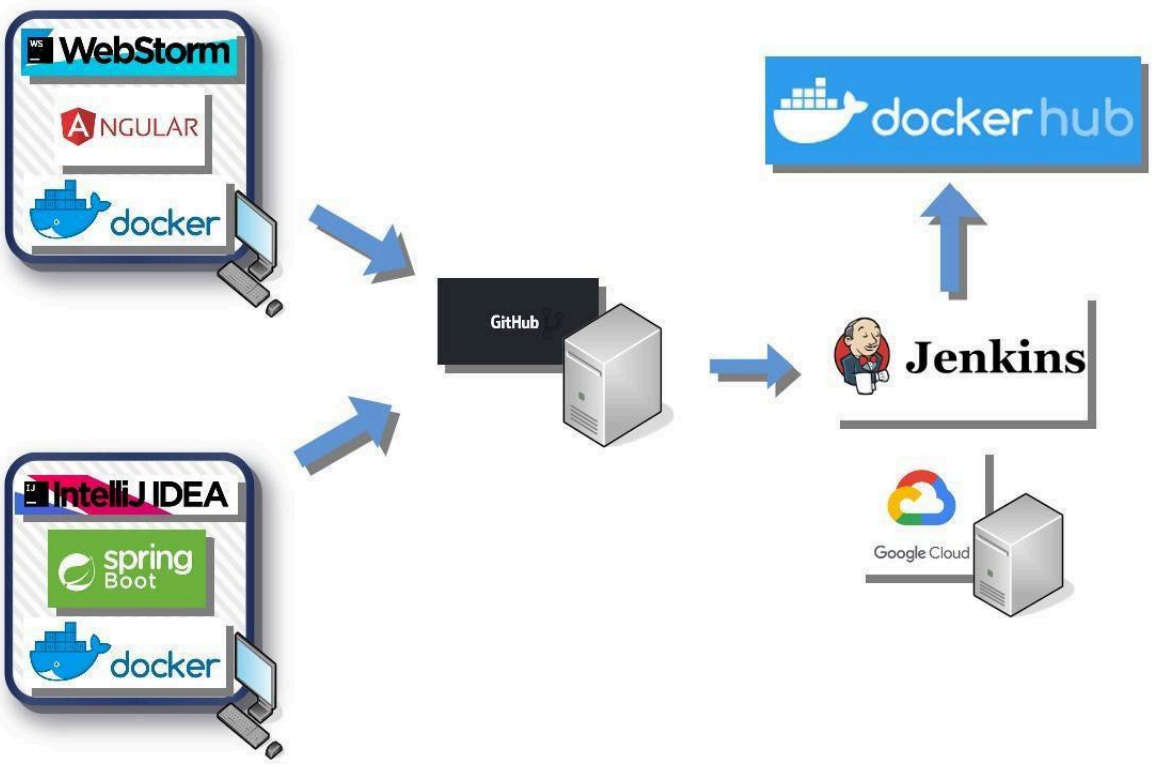
```
capstone-project
├── Jenkinsfile
├── docker-compose.yml
├── user-service/Dockerfile
├── product-service/Dockerfile
├── order-service/Dockerfile
├── angular-ui/Dockerfile
└── README.md
```

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.

Business Rules for Smart Ticket & Issue Management System

🔹 **Ticket Rules**

- **Ticket lifecycle: A ticket must progress through states `OPEN` → `IN_PROGRESS` → `RESOLVED` → `CLOSED`.**
- **Mandatory fields: Each ticket must have a title, description, priority, requester details, and creation timestamp.**
- **Priority assignment: Tickets must be classified as `HIGH`, `MEDIUM`, or `LOW`.**
- **Reopen policy: Closed tickets can only be reopened by managers or the original requester.**
- **Audit trail: Every ticket update must be logged with timestamp and user ID.**

🔹 **Assignment Rules**

- **Manual assignment: Managers can assign tickets to agents directly.**
- **Auto assignment: System must select agents based on role (senior/junior) and workload balancing.**
- **Workload limit: No agent may exceed a configurable maximum number of active tickets.**
- **Completion: Agents must mark assignments complete before tickets can be resolved.**
- **Activity tracking: All assignment actions (create, update, complete, escalate) must be logged.**

🔹 **Escalation & SLA Rules**

- **SLA deadlines:**
  - **HIGH priority → 4 hours**
  - **MEDIUM priority → 24 hours**
  - **LOW priority → 72 hours**
- **Escalation levels:**
  - **L1 → assigned agent breach**
  - **L2 → team lead breach**
  - **L3 → manager breach**
- **Automatic escalation: System must escalate when SLA is breached, updating ticket and assignment status.**
- **Manual escalation: Managers can escalate tickets at any time.**
- **Escalation history: Each escalation must be recorded with level, timestamp, and reason.**

🔹 **Notification Rules**

- **Event triggers: Notifications must be sent on ticket creation, assignment, escalation, and resolution.**
- **Channels: Notifications must support email, SMS, or Slack based on user preferences.**
- **Escalation alerts: Must include ticket ID, escalation level, and breach time.**
- **Delivery guarantee: Failed notifications must be retried at least 3 times before logging as failed.**

🔹 **Dashboard & Reporting Rules**

- **Summary view: Must show counts of total tickets, active assignments, completed, escalated, and breached.**
- **Agent workload view: Must show per-agent ticket counts and SLA breaches.**
- **Escalation tracking: Must list escalated tickets with escalation level and timestamps.**
- **Priority distribution: Must aggregate tickets by HIGH, MEDIUM, LOW.**
- **SLA compliance: Must report % of tickets resolved within SLA vs breached.**
- **Real-time updates: Dashboard must refresh data at configurable intervals (e.g., every 30 seconds).**

🔹 **User/Auth Rules**

- **Role-based access:**
  - **Agents → can view and update assigned tickets.**
  - **Managers → can assign, escalate, and reopen tickets.**
  - **Admins → full system access.**
- **Authentication: All endpoints must be secured with JWT tokens.**
- **Authorization: Role checks must be enforced at the gateway level.**
- **Audit logging: All user actions must be logged for compliance.**

Data Model

User

- `id` **(String, PK)**
- `username` **(String, required)**
- `password` **(String, required)**
- `email` **(String, required, unique)**
- `roles` **(Set<ROLE>)(enum)**
- `enabled` **(boolean)**
- `passwordLastChanged` **(LocalDateTime)**
- `resetToken` **(String)**
- `resetTokenExpiry` **(Instant)**

Category

- `id` **(String, PK)**
- `name` **(String)**
- `description` **(String)**
- `linkedSlaId` **(String → FK to SLA Rule)**
- `active` **(boolean)**

Ticket

- `id` **(String, PK)**
- `title` **(String, required)**
- `description` **(String, required)**
- `categoryId` **(String → FK to Category)**

- `status` (STATUS enum: OPEN, IN_PROGRESS, CLOSED, ESCALATED, etc.)
- `priority` (PRIORITY enum: HIGH, MEDIUM, LOW)
- `createdBy` (String → FK to User)
- `createdAt` (LocalDateTime)
- `updatedAt` (LocalDateTime)

TicketActivity

- `id` (String, PK)
- `ticketId` (String → FK to Ticket)
- `actorId` (String → FK to User/Agent)
- `actionType` (ACTION_TYPE enum: COMMENT, ASSIGNMENT, ESCALATION, STATUS_CHANGE)
- `details` (String)
- `timestamp` (Instant)

SlaRule

- `id` (String, PK)
- `priority` (PRIORITY enum)
- `responseMinutes` (int)
- `resolutionMinutes` (int)

Assignment