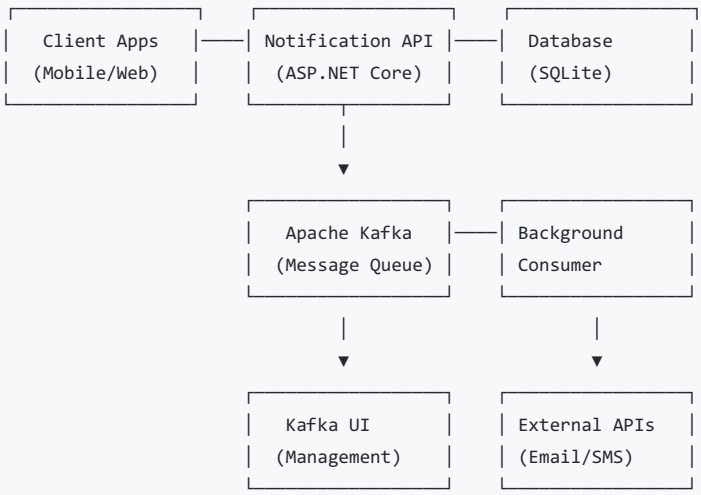
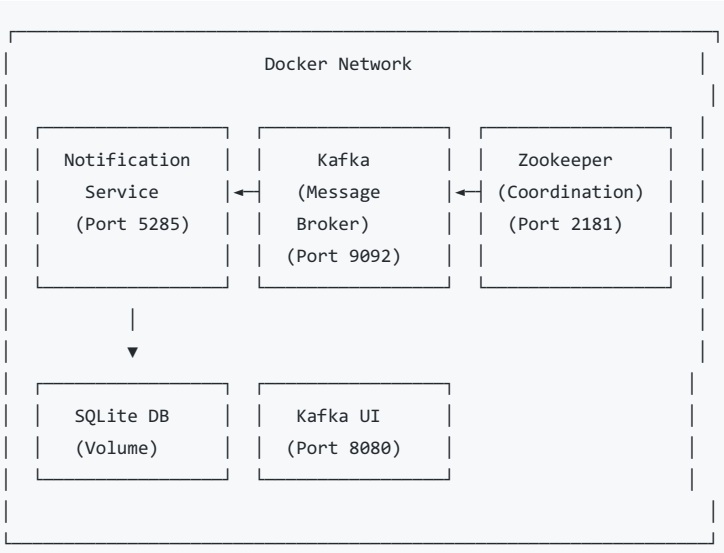
# Notification Service - System Design

Production-ready notification service with .NET 9, Kafka, and Docker providing performance improvement through async processing

## Architecture Overview




## Containerized Architecture




## Technology Stack

**Core**: ASP.NET Core 9, Apache Kafka **Data**: Entity Framework Core, SQLite, Repository Pattern
**Infrastructure**: Docker Compose, Kafka UI, Health Checks
**Production**: Structured Logging, Error Handling, Container Security

## Data Model

| Field | Type | Purpose |
|---|---|---|
| `Id` | GUID | Primary Key |
| `UserId` | String(100) | User identifier |
| `Channel` | Enum | Email/SMS/Push |
| `Template` | String(100) | Template ID |
| | | |

| Field | Type | Purpose |
|---|---|---|
| `Data` | JSON | Template variables |
| `Status` | Enum | Pending/Sent/Failed/Retrying |
| `IdempotencyKey` | String(100) | To prevent duplicates |
| `CreatedAt/SentAt` | DateTime | Timestamps |
| `ErrorMessage` | String | Failure details |
| `RetryCount` | Integer | Retry attempts |

**Indexes**: Primary (Id), Unique (IdempotencyKey), Composite (UserId, CreatedAt)

# Performance Architecture

## Sync vs Async Processing

**Synchronous Mode** (Local Development):

```
Request → Validation → DB Insert → Process → Response (20-50ms)
```

**Asynchronous Mode** (Docker/Production):

```
Request → Validation → DB Insert → Kafka → Response (2-5ms)
                                     ↓
Background Consumer → Process → DB Update
```

**Result**: 10x performance improvement (2-5ms vs 20-50ms)

# Notification Flow: Creation to Delivery

## Asynchronous Flow (Production with Kafka)

```
1. Client Request
   ↓
2. API Validation (Input validation, channel verification)
   ↓
3. Idempotency Check (Check existing notification by key)
   ↓
4. Database Insert (Status: Pending, CreatedAt timestamp)
   ↓
5. Kafka Event Publish (NotificationCreated event)
   ↓
6. Immediate Response (2-5ms, Status: Pending)

   --- Background Processing ---
   ↓
7. Kafka Consumer Receives Event
   ↓
8. Channel Processing (Email/SMS/Push simulation)
   ↓
9. Status Update (Status: Sent/Failed, SentAt timestamp)
   ↓
10. Error Handling (Retry logic if failed)
```

## Synchronous Flow (Local Development)

```
1. Client Request
   ↓
2. API Validation
   ↓
3. Idempotency Check
   ↓
4. Database Insert (Status: Pending)
   ↓
5. Immediate Processing (Channel simulation)
   ↓
6. Status Update (Status: Sent/Failed)
   ↓
7. Response (20-50ms, Final status)
```

### Flow Details

**Step 1-2: Request & Validation**

- JSON payload validation using Data Annotations
- Channel verification (Email/SMS/Push)
- Template and user ID validation
- Data structure validation

**Step 3: Idempotency Handling**

- Check for existing notification with same IdempotencyKey
- Return existing notification if found (prevents duplicates)
- Race condition safe with database unique constraints

**Step 4: Database Persistence**

- Entity Framework Core transaction
- Notification record created with Pending status
- Automatic CreatedAt timestamp
- Indexed storage for fast retrieval

**Step 5-6: Async Processing (Kafka Mode)**

- Kafka event published to 'notification-events' topic
- Immediate API response (2-5ms)
- Client receives notification ID for tracking

**Step 7-10: Background Processing**

- Dedicated consumer processes events
- Channel-specific processing simulation
- Database status updates (Sent/Failed)
- Retry logic for failed notifications
- Error logging and monitoring

## Key Design Decisions

### 1. Kafka Async Processing

- **Why**: 10x faster responses, horizontal scalability
- **Implementation**: Dual-mode (sync/async) based on Kafka availability
- **Benefit**: Immediate API response, background processing

### 2. Database Level Idempotency

- **Why**: Race condition safe, no additional infrastructure
- **Implementation**: Unique constraint on IdempotencyKey in the database
- **Benefit**: Guaranteed duplicate prevention under concurrency

### 3. Container First Architecture

- **Why**: Consistent dev/prod environments, easy scaling
- **Implementation**: Docker Compose with health checks
- **Benefit**: Complete orchestration, service isolation, can run anywhere

### 4. Modular Monolith

- **Why**: Balance simplicity and scalability
- **Implementation**: Clear module boundaries, loose coupling
- **Benefit**: Easy development, can evolve to microservices

## Security & Production Features

**Current Security**:

- Input validation with size limits
- SQL injection protection (EF parameterized queries)
- Container security
- Secure error handling

**Production Enhancements**:

- Authentication (JWT/API keys)
- Rate limiting (per-user/global)
- Audit logging

## Scalability Strategy

**Current Capacity**:

- Single instance: 1,000+ notifications/second
- Memory: ~100MB base
- Database: SQLite 10,000+ ops/second

**Horizontal Scaling**:

```
Load Balancer → Multiple App Instances → Shared Kafka + Database
```

**Production Scaling**:

```
API Gateway → Auto Scaling → Kafka Cluster → External APIs
                                ↓
                        Consumer Groups → PostgreSQL
```

## Adding new Channel

Easly adding new channels from. Shared folder -> Enums -> NotificationChannel Class -> add new channel, the modify the Switch in line 160 in NotificationService class

## Deployment

**Local**: `dotnet run` (sync mode) or `docker-compose up -d` (async mode)
**Production**: `docker-compose -f docker-compose.prod.yml up -d`
**CI/CD**: GitHub Actions → Build → Test → Docker → Deploy

## Conclusion

The Notification Service demonstrates **production-ready architecture** with:

**10x performance** improvement via Kafka async processing
**Rock-solid idempotency** with database constraints
**Complete containerization** with Docker orchestration
**Multi-channel support** (Email, SMS, Push)
**Horizontal scalability** with stateless design
**Operational excellence** with monitoring and logging