# PepSales Notification Service Documentation

## Table of Contents

1. [Project Overview](#project-overview)
2. [Technical Architecture](#technical-architecture)
3. [Code Structure](#code-structure)
4. [Implementation Details](#implementation-details)
5. [Interview Questions and Answers](#interview-questions-and-answers)
6. [Technical Concepts](#technical-concepts)

## Project Overview

PepSales is a robust notification service that supports multiple notification channels including email, SMS, and in-app notifications. The service is built using Node.js and TypeScript, implementing a queue-based processing system with automatic retry mechanisms.

### Key Features

* Multiple notification types (Email, SMS, In-app)
* Queue-based processing using RabbitMQ
* Automatic retry mechanism for failed notifications
* RESTful API endpoints
* TypeScript implementation

### Live Demo

The service is deployed at: <https://pepsales-8vmw.onrender.com>

## Technical Architecture

### Project Structure

src/  
├── index.ts # Application entry point  
├── controllers/ # Request handlers  
│ └── notification.controller.ts  
├── services/ # Business logic  
│ ├── notification.service.ts  
│ └── queue.service.ts  
├── routes/ # API routes  
│ └── notification.routes.ts  
└── types/ # TypeScript type definitions  
 └── notification.ts

### Design Patterns Used

1. **Strategy Pattern**
   * Different notification types (Email, SMS, In-app) are handled by separate strategies
   * Each notification type has its own implementation method
   * Easy to add new notification types
2. **Observer Pattern**
   * Queue service observes and processes notifications
   * Decouples notification creation from processing
3. **Singleton Pattern**
   * Services are instantiated once and reused
   * Maintains single instances of connections (RabbitMQ, SMTP, Twilio)

## Code Structure

### 1. Entry Point (index.ts)

const app = express();  
app.use(cors());  
app.use(express.json());  
  
// Service initialization  
const notificationService = new NotificationService();  
const queueService = new QueueService(notificationService);  
  
// Route registration  
app.use('/notifications', notificationRoutes);

### 2. Routes Layer (routes/notification.routes.ts)

const router = Router();  
router.post('/', notificationController.sendNotification);  
router.get('/users/:id', notificationController.getUserNotifications);

### 3. Controllers Layer (controllers/notification.controller.ts)

export class NotificationController {  
 private notificationService: NotificationService;  
  
 sendNotification = async (req: Request, res: Response): Promise<void> => {  
 // Request handling logic  
 };  
  
 getUserNotifications = async (req: Request, res: Response): Promise<void> => {  
 // Request handling logic  
 };  
}

### 4. Services Layer

#### Notification Service

export class NotificationService {  
 private emailTransporter: nodemailer.Transporter;  
 private twilioClient: twilio.Twilio;  
 private queueService: QueueService;  
}

#### Queue Service

export class QueueService {  
 private connection: any;  
 private channel: amqp.Channel;  
 private readonly mainQueue = 'notifications';  
 private readonly retryQueue = 'notifications-retry';  
}

### 5. Types Layer

export interface Notification {  
 id: string;  
 userId: string;  
 type: NotificationType;  
 title: string;  
 message: string;  
 metadata: NotificationMetadata;  
 status: NotificationStatus;  
 createdAt: string;  
 updatedAt: string;  
}

## Implementation Details

### Queue System Implementation

* Uses RabbitMQ for message queuing
* Implements retry mechanism with 1-minute delay
* Maximum of 3 retries for failed notifications
* Dead letter queue for failed messages

### Notification Processing

1. **Email Notifications**
   * Uses Nodemailer with SMTP
   * Supports HTML and plain text
   * Configurable SMTP settings
2. **SMS Notifications**
   * Uses Twilio API
   * Supports international phone numbers
   * Requires verified numbers in trial mode
3. **In-app Notifications**
   * Currently logs to console
   * Ready for WebSocket integration
   * Supports metadata for customization

## Interview Questions and Answers

### 1. How do you handle message persistence in RabbitMQ?

**Answer**: “We use durable queues and persistent messages:

await this.channel.assertQueue(this.mainQueue, { durable: true });  
await this.channel.sendToQueue(  
 this.mainQueue,  
 Buffer.from(JSON.stringify(notification)),  
 { persistent: true }  
);

This ensures messages survive broker restarts.”

### 2. How do you implement the retry mechanism?

**Answer**: “We use a dead letter queue with TTL:

await this.channel.assertQueue(this.retryQueue, {   
 durable: true,  
 arguments: {  
 'x-dead-letter-exchange': '',  
 'x-dead-letter-routing-key': this.mainQueue,  
 'x-message-ttl': 60000 // 1 minute  
 }  
});

Failed messages are moved to retry queue and return to main queue after TTL.”

### 3. How do you handle different notification types?

**Answer**: “We use a switch statement with type-specific handlers:

switch (notification.type) {  
 case 'EMAIL':  
 await this.sendEmail(notification);  
 break;  
 case 'SMS':  
 await this.sendSMS(notification);  
 break;  
 case 'IN\_APP':  
 await this.sendInAppNotification(notification);  
 break;  
}

Each type has its own implementation method.”

### 4. How do you ensure type safety?

**Answer**: “We use TypeScript interfaces and type definitions:

interface Notification {  
 id: string;  
 type: NotificationType;  
 status: NotificationStatus;  
 // ...  
}

This ensures compile-time type checking and better IDE support.”

### 5. How do you handle service initialization?

**Answer**: “We use an initialization pattern:

async initialize(): Promise<void> {  
 if (this.isInitialized) return;  
 // Initialize connections and channels  
 this.isInitialized = true;  
}

This ensures services are properly initialized before use.”

## Technical Concepts

### 1. Message Queue Patterns

* Producer-Consumer pattern
* Dead Letter Queue pattern
* Retry pattern with exponential backoff

### 2. Error Handling Strategies

* Graceful degradation
* Fallback mechanisms
* Comprehensive logging

### 3. State Management

* In-memory storage using Map
* Status tracking
* Atomic updates

### 4. Connection Management

* Connection pooling
* Channel management
* Proper cleanup

### 5. Performance Considerations

1. **Queue Optimization**
   * Message persistence
   * Batch processing
   * Connection pooling
2. **Memory Management**
   * Proper cleanup of resources
   * Connection and channel management
   * Map-based storage
3. **Error Recovery**
   * Automatic retries
   * Fallback mechanisms
   * Status tracking

### 6. Security Considerations

1. **Environment Variables**
   * Sensitive data management
   * Configuration security
   * API key protection
2. **Input Validation**
   * Type checking
   * Data validation
   * Error handling
3. **API Security**
   * CORS configuration
   * Rate limiting
   * Authentication (can be added)

## Best Practices Implemented

1. **Code Organization**
   * Clear separation of concerns
   * Modular architecture
   * Single responsibility principle
2. **Type Safety**
   * TypeScript interfaces
   * Strong typing
   * Compile-time checks
3. **Error Handling**
   * Try-catch blocks
   * Error logging
   * User-friendly messages
4. **Resource Management**
   * Connection handling
   * Graceful shutdown
   * Memory management

This structure follows clean architecture principles and makes the code: - Easy to maintain - Easy to test - Easy to extend - Easy to understand - Easy to debug