Hotel Management System - Developer Documentation

Table of Contents

Architecture Overview

System Architecture

The Hotel Management System follows a microservices architecture pattern with the following components:

```
 ┌──────────────────┐     ┌──────────────────┐
┌──────────────────┐
│   Frontend        │     │   API Gateway     │     │   Load Balancer
│
│   (React)         │◄──►│   (Optional)      │◄──►│
│
└──────────────────┘     └──────────────────┘
└──────────────────┘
         │                         │                         │
         ▼                         ▼                         ▼
┌──────────────────┐     ┌──────────────────┐
┌──────────────────┐
│  Auth Service     │     │  Room Service     │     │ Reservation
│
│  (NestJS)         │     │  (NestJS)         │     │ Service
│
│  Port: 3001       │     │  Port: 3002       │     │ (NestJS)
│
└──────────────────┘     └──────────────────┘     │ Port: 3003
```

```
│
          │                         │
└───────────────────┘
          ▼                         ▼                          │
                                                              ▼
┌───────────────────┐    ┌───────────────────┐
│    MongoDB        │    │    MongoDB        │
┌───────────────────┐
│   (Auth DB)       │    │   (Rooms DB)      │    │    MongoDB
│
└───────────────────┘    └───────────────────┘    │ (Reservations)
│

└───────────────────┘
          │                         │                          │
          ▼                         ▼                          ▼
┌───────────────────┐    ┌───────────────────┐
┌───────────────────┐
│      Redis        │    │      Kafka        │    │    Prometheus
│
│   (Caching)       │    │  (Messaging)      │    │   (Monitoring)
│
└───────────────────┘    └───────────────────┘
└───────────────────┘
```

Technology Stack

Backend

- **Framework**: NestJS (Node.js)
- **Language**: TypeScript
- **API**: GraphQL with Apollo Server
- **Database**: MongoDB with Mongoose ODM
- **Caching**: Redis
- **Message Queue**: Apache Kafka
- **Authentication**: JWT (JSON Web Tokens)
- **Testing**: Jest, Supertest

Frontend

- **Framework**: React 18
- **Language**: JavaScript (JSX)
- **Styling**: Tailwind CSS
- **UI Components**: shadcn/ui
- **State Management**: React Context API
- **GraphQL Client**: Apollo Client
- **Build Tool**: Vite

- **Testing**: React Testing Library

DevOps

- **Containerization**: Docker
- **Orchestration**: Kubernetes
- **CI/CD**: GitHub Actions
- **Monitoring**: Prometheus + Grafana
- **Load Testing**: k6

Design Patterns

Microservices Patterns

- **Database per Service**: Each service has its own database
- **API Gateway**: Single entry point for client requests
- **Service Discovery**: Automatic service registration and discovery
- **Circuit Breaker**: Fault tolerance for service communication
- **Event Sourcing**: Using Kafka for event-driven communication

Backend Patterns

- **Repository Pattern**: Data access abstraction
- **Dependency Injection**: IoC container for loose coupling
- **Decorator Pattern**: NestJS decorators for metadata
- **Strategy Pattern**: Different authentication strategies
- **Observer Pattern**: Event-driven architecture

Development Environment Setup

Prerequisites

```
# Install Node.js (v18+)
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash
-
sudo apt-get install -y nodejs

# Install Docker
curl -fsSL https://get.docker.com -o get-docker.sh
sh get-docker.sh

# Install Docker Compose
sudo curl -L "https://github.com/docker/compose/releases/
download/v2.12.2/docker-compose-$(uname -s)-$(uname -m)" -o /
```

```
usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# Install kubectl
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://
dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/
kubectl
```

Local Development Setup

```
# Clone repository
git clone <repository-url>
cd hotel-management-app

# Install dependencies for all services
npm run install:all

# Start infrastructure services
docker-compose up -d mongodb redis kafka

# Start backend services in development mode
npm run dev:auth &
npm run dev:rooms &
npm run dev:reservations &

# Start frontend
npm run dev:frontend
```

Environment Variables

Create `.env` files for each service:

```
# backend/auth-service/.env
NODE_ENV=development
PORT=3001
MONGODB_URI=mongodb://localhost:27017/hotel-auth
JWT_SECRET=dev-secret-key
JWT_EXPIRES_IN=7d
REDIS_URL=redis://localhost:6379
BCRYPT_ROUNDS=10

# backend/room-service/.env
NODE_ENV=development
PORT=3002
MONGODB_URI=mongodb://localhost:27017/hotel-rooms

# backend/reservation-service/.env
```

```
NODE_ENV=development
PORT=3003
MONGODB_URI=mongodb://localhost:27017/hotel-reservations
ROOM_SERVICE_URL=http://localhost:3002
AUTH_SERVICE_URL=http://localhost:3001
```

Code Structure

Backend Service Structure

```
backend/
├── auth-service/
│   ├── src/
│   │   ├── auth/
│   │   │   ├── auth.controller.ts
│   │   │   ├── auth.service.ts
│   │   │   ├── auth.resolver.ts
│   │   │   ├── auth.module.ts
│   │   │   ├── dto/
│   │   │   ├── entities/
│   │   │   └── guards/
│   │   ├── common/
│   │   │   ├── decorators/
│   │   │   ├── filters/
│   │   │   ├── interceptors/
│   │   │   └── pipes/
│   │   ├── config/
│   │   ├── app.module.ts
│   │   └── main.ts
│   ├── test/
│   ├── Dockerfile
│   └── package.json
├── room-service/
└── reservation-service/
```

Frontend Structure

```
frontend/hotel-frontend-react/
├── src/
│   ├── components/
│   │   ├── ui/           # Reusable UI components
│   │   ├── AuthPage.jsx
│   │   ├── RoomBooking.jsx
│   │   └── AdminDashboard.jsx
│   ├── contexts/
│   │   └── AuthContext.jsx
│   ├── hooks/
│   ├── lib/
│   │   ├── apollo.js      # GraphQL client setup
│   │   └── graphql.js     # GraphQL queries/mutations
│   ├── assets/
│   ├── App.jsx
│   └── main.jsx
├── public/
├── package.json
└── vite.config.js
```

Naming Conventions

Files and Directories

- **PascalCase**: React components (`AuthPage.jsx`)
- **camelCase**: Services, utilities (`authService.ts`)
- **kebab-case**: Directories (`auth-service/`)
- **UPPER_CASE**: Constants (`JWT_SECRET`)

Code

- **PascalCase**: Classes, interfaces, types
- **camelCase**: Variables, functions, methods
- **UPPER_CASE**: Constants, environment variables

API Development

GraphQL Schema Design

Schema-First Approach

```graphql
# schema.graphql
type User {
  id: ID!
  email: String!
  firstName: String!
  lastName: String!
  role: UserRole!
  createdAt: DateTime!
  updatedAt: DateTime!
}

enum UserRole {
  CUSTOMER
  ADMIN
}

input CreateUserInput {
  email: String!
  password: String!
  firstName: String!
  lastName: String!
}

type Mutation {
  createUser(input: CreateUserInput!): User!
}
```

Resolver Implementation

```
// auth.resolver.ts
@Resolver(() => User)
export class AuthResolver {
  constructor(private readonly authService: AuthService) {}

  @Mutation(() => AuthResponse)
  async register(@Args('input') input: RegisterInput):
Promise<AuthResponse> {
    return this.authService.register(input);
  }

  @Query(() => User)
  @UseGuards(JwtAuthGuard)
  async me(@CurrentUser() user: User): Promise<User> {
    return user;
  }
}
```

Service Layer Pattern

```
// auth.service.ts
@Injectable()
export class AuthService {
  constructor(
    @InjectModel(User.name) private userModel: Model<User>,
    private jwtService: JwtService,
  ) {}

  async register(input: RegisterInput): Promise<AuthResponse> {
    // Validate input
    await this.validateRegistrationInput(input);
    // Hash password
    const hashedPassword = await bcrypt.hash(input.password,
10);

    // Create user
    const user = await this.userModel.create({
      ...input,
      password: hashedPassword,
    });

    // Generate token
    const token = this.generateToken(user);

    return { user, token };
  }

  private async validateRegistrationInput(input:
RegisterInput): Promise<void> {
    const existingUser = await this.userModel.findOne({ email:
input.email });
    if (existingUser) {
      throw new ConflictException('User already exists');
    }
  }

  private generateToken(user: User): string {
    const payload = { sub: user._id, email: user.email, role:
user.role };
    return this.jwtService.sign(payload);
  }
}
```

Error Handling

```
// common/filters/graphql-exception.filter.ts
@Catch()
export class GraphQLExceptionFilter implements ExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    if (exception instanceof HttpException) {
      return new GraphQLError(exception.message, {
        extensions: {
          code: exception.getStatus(),
          timestamp: new Date().toISOString(),
        },
      });
    }

    return new GraphQLError('Internal server error', {
      extensions: {
        code: 'INTERNAL_ERROR',
        timestamp: new Date().toISOString(),
      },
    });
  }
}
```

Validation

```
// dto/register.input.ts
@InputType()
export class RegisterInput {
  @Field()
  @IsEmail()
  email: string;

  @Field()
  @MinLength(8)
  @Matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/, {
    message: 'Password must contain uppercase, lowercase, and
number',
  })
  password: string;

  @Field()
  @IsNotEmpty()
  @MaxLength(50)
  firstName: string;

  @Field()
  @IsNotEmpty()
  @MaxLength(50)
  lastName: string;
}
```

Database Schema

MongoDB Collections

Users Collection (auth-service)

```
{
  _id: ObjectId,
  email: String, // unique index
  password: String, // bcrypt hashed
  firstName: String,
  lastName: String,
  role: String, // enum: 'customer', 'admin'
  phoneNumber: String,
  address: String,
  isActive: Boolean,
  lastLogin: Date,
  createdAt: Date,
  updatedAt: Date
}
```

Rooms Collection (room-service)

```
{
  _id: ObjectId,
  roomNumber: String, // unique index
  roomType: String, // enum: 'standard', 'deluxe', 'suite'
  price: Number,
  availability: Boolean, // index
  description: String,
  amenities: [String],
  maxGuests: Number,
  images: [String], // URLs to room images
  createdAt: Date,
  updatedAt: Date
}
```

Reservations Collection (reservation-service)

```
{
  _id: ObjectId,
  userId: String, // index
  roomId: String, // index
  checkInDate: Date, // compound index with checkOutDate
  checkOutDate: Date,
  numberOfGuests: Number,
  totalPrice: Number,
  status: String, // enum: 'pending', 'confirmed', 'cancelled',
'completed'
  paymentId: String,
  specialRequests: String,
  createdAt: Date,
  updatedAt: Date
}
```

Database Indexes

```
// Users collection
db.users.createIndex({ email: 1 }, { unique: true })
db.users.createIndex({ role: 1 })
db.users.createIndex({ isActive: 1 })

// Rooms collection
db.rooms.createIndex({ roomNumber: 1 }, { unique: true })
db.rooms.createIndex({ availability: 1 })
db.rooms.createIndex({ roomType: 1 })
db.rooms.createIndex({ price: 1 })

// Reservations collection
db.reservations.createIndex({ userId: 1 })
db.reservations.createIndex({ roomId: 1 })
db.reservations.createIndex({ checkInDate: 1, checkOutDate: 1 })
db.reservations.createIndex({ status: 1 })
db.reservations.createIndex({ createdAt: -1 })
```

Testing Guidelines

Unit Testing

```
 // auth.service.spec.ts
describe('AuthService', () => {
  let service: AuthService;
  let userModel: Model<User>;

  beforeEach(async () => {
    const module: TestingModule = await
Test.createTestingModule({
      providers: [
        AuthService,
        {
          provide: getModelToken(User.name),
          useValue: mockUserModel,
        },
        {
          provide: JwtService,
          useValue: mockJwtService,
        },
      ],
    }).compile();

    service = module.get<AuthService>(AuthService);
    userModel =
module.get<Model<User>>(getModelToken(User.name));
  });

  describe('register', () => {
    it('should create a new user successfully', async () => {
      // Arrange
      const input = {
        email: 'test@example.com',
        password: 'Password123',
        firstName: 'John',
        lastName: 'Doe',
      };

      mockUserModel.findOne.mockResolvedValue(null);
      mockUserModel.create.mockResolvedValue(mockUser);

      // Act
      const result = await service.register(input);

      // Assert
      expect(result).toHaveProperty('user');
      expect(result).toHaveProperty('token');
      expect(mockUserModel.create).toHaveBeenCalledWith(
        expect.objectContaining({
          email: input.email,
          firstName: input.firstName,
          lastName: input.lastName,
        }),
      );
    });
  });
});
```

Integration Testing

```
// auth.e2e-spec.ts
describe('AuthController (e2e)', () => {
  let app: INestApplication;

  beforeEach(async () => {
    const moduleFixture: TestingModule = await
Test.createTestingModule({
      imports: [AppModule],
    }).compile();

    app = moduleFixture.createNestApplication();
    await app.init();
  });

  it('/graphql (POST) register mutation', () => {
    return request(app.getHttpServer())
      .post('/graphql')
      .send({
        query: `
          mutation Register($input: RegisterInput!) {
```

```
          register(input: $input) {
            token
            user {
              email
              firstName
              lastName
            }
          }
        }
      `,
      variables: {
        input: {
          email: 'test@example.com',
          password: 'Password123',
          firstName: 'John',
          lastName: 'Doe',
        },
      },
    })
    .expect(200)
    .expect((res) => {
      expect(res.body.data.register).toBeDefined();
      expect(res.body.data.register.token).toBeDefined();
    });
  });
});
```

Frontend Testing

```
// AuthPage.test.jsx
import { render, screen, fireEvent, waitFor } from '@testing-
library/react';
import { MockedProvider } from '@apollo/client/testing';
import AuthPage from './AuthPage';
import { LOGIN_MUTATION } from '../lib/graphql';

const mocks = [
  {
    request: {
      query: LOGIN_MUTATION,
      variables: {
        input: {
          email: 'test@example.com',
          password: 'password123',
        },
      },
    },
    result: {
      data: {
        login: {
          token: 'mock-token',
          user: {
            id: '1',
            email: 'test@example.com',
            firstName: 'John',
            lastName: 'Doe',
            role: 'customer',
          },
        },
      },
    },
  },
];

test('should login user successfully', async () => {
  render(
    <MockedProvider mocks={mocks} addTypename={false}>
      <AuthPage />
    </MockedProvider>
  );

  fireEvent.change(screen.getByLabelText(/email/i), {
    target: { value: 'test@example.com' },
  });
  fireEvent.change(screen.getByLabelText(/password/i), {
    target: { value: 'password123' },
  });
  fireEvent.click(screen.getByRole('button', { name: /sign in/
i }));

  await waitFor(() => {
    expect(mockLogin).toHaveBeenCalledWith({
      token: 'mock-token',
      user: expect.objectContaining({
        email: 'test@example.com',
      }),
    });
  });
});
```

Test Coverage

Maintain minimum test coverage: - **Unit Tests**: 80% code coverage - **Integration Tests**: All API endpoints - **E2E Tests**: Critical user workflows

```
 # Run tests with coverage
npm run test:cov

# Run specific test suite
```

```
npm run test auth.service.spec.ts

# Run e2e tests
npm run test:e2e
```

Contributing Guidelines

Git Workflow

Branch Naming Convention

```
feature/HMS-123-add-user-authentication
bugfix/HMS-456-fix-reservation-validation
hotfix/HMS-789-security-patch
release/v1.2.0
```

Commit Message Format

```
type(scope): description

[optional body]

[optional footer]
```

Examples:

```
feat(auth): add JWT token refresh functionality
fix(reservations): resolve date validation issue
docs(api): update GraphQL schema documentation
test(rooms): add unit tests for room service
```

Pull Request Process

1. **Create Feature Branch**: `bash git checkout -b feature/HMS-123-add-feature`

2. **Make Changes and Commit**: `bash git add . git commit -m "feat(scope): add new feature"`

3. **Push and Create PR**: `bash git push origin feature/HMS-123-add-feature`

4. **PR Requirements**:

5. [ ] All tests pass
6. [ ] Code coverage maintained
7. [ ] Documentation updated
8. [ ] Reviewed by at least 2 developers
9. [ ] No merge conflicts

Code Review Checklist

General

- [ ] Code follows project conventions
- [ ] No hardcoded values or secrets
- [ ] Error handling implemented
- [ ] Logging added where appropriate
- [ ] Performance considerations addressed

Backend

- [ ] Input validation implemented
- [ ] Database queries optimized
- [ ] Authentication/authorization checked
- [ ] API documentation updated
- [ ] Tests cover new functionality

Frontend

- [ ] Components are reusable
- [ ] Accessibility standards met
- [ ] Responsive design implemented
- [ ] Error states handled
- [ ] Loading states implemented

Development Standards

Code Formatting

```bash
# Install Prettier and ESLint
npm install --save-dev prettier eslint

# Format code
npm run format
```

```
# Lint code
npm run lint
```

TypeScript Guidelines

```
// Use explicit types
interface UserCreateInput {
  email: string;
  password: string;
  firstName: string;
  lastName: string;
}

// Use enums for constants
enum UserRole {
  CUSTOMER = 'customer',
  ADMIN = 'admin',
}

// Use generics for reusable code
class Repository<T> {
  async findById(id: string): Promise<T | null> {
    // implementation
  }
}
```

Deployment Procedures

Development Deployment

```
# Build all services
npm run build:all

# Run tests
npm run test:all

# Start with Docker Compose
docker-compose up -d
```

Staging Deployment

```
# Build and tag images
docker build -t hotel-auth:staging ./backend/auth-service
docker build -t hotel-rooms:staging ./backend/room-service
docker build -t hotel-reservations:staging ./backend/
```

```
reservation-service
docker build -t hotel-frontend:staging ./frontend/hotel-
frontend-react

# Deploy to staging environment
kubectl apply -f k8s/staging/
```

Production Deployment

```
# Tag for production
docker tag hotel-auth:staging hotel-auth:v1.0.0
docker tag hotel-rooms:staging hotel-rooms:v1.0.0
docker tag hotel-reservations:staging hotel-reservations:v1.0.0
docker tag hotel-frontend:staging hotel-frontend:v1.0.0

# Push to registry
docker push hotel-auth:v1.0.0
docker push hotel-rooms:v1.0.0
docker push hotel-reservations:v1.0.0
docker push hotel-frontend:v1.0.0

# Deploy to production
kubectl apply -f k8s/production/
```

Rollback Procedures

```
# Rollback to previous version
kubectl rollout undo deployment/auth-service
kubectl rollout undo deployment/room-service
kubectl rollout undo deployment/reservation-service
kubectl rollout undo deployment/frontend

# Check rollout status
kubectl rollout status deployment/auth-service
```

Monitoring and Alerting

Health Checks

```
// health.controller.ts
@Controller('health')
export class HealthController {
  @Get()
  check(): { status: string; timestamp: string } {
    return {
      status: 'ok',
      timestamp: new Date().toISOString(),
    };
  }

  @Get('ready')
  ready(): { status: string } {
    // Check database connectivity
    // Check external service dependencies
    return { status: 'ready' };
  }
}
```

Metrics Collection

```
// metrics.service.ts
@Injectable()
export class MetricsService {
  private readonly httpRequestsTotal = new Counter({
    name: 'http_requests_total',
    help: 'Total number of HTTP requests',
    labelNames: ['method', 'route', 'status'],
  });

  incrementHttpRequests(method: string, route: string, status:
number): void {
    this.httpRequestsTotal.inc({ method, route, status:
status.toString() });
  }
}
```

This developer documentation is maintained by the development team and updated with each release.