Online Restaurant Ordering System API - Technical Documentation

Table of Contents

Architecture Overview

The Online Restaurant Ordering System API follows a layered architecture pattern built with FastAPI, SQLAlchemy, and MySQL.

High-Level Architecture

```
┌─────────────────┐
│   FastAPI App   │ ← HTTP Layer (Routing, Validation)
├─────────────────┤
│    Routers      │ ← Route Handlers & Request Validation
├─────────────────┤
│   Controllers   │ ← Business Logic & Data Processing
├─────────────────┤
│    Services     │ ← Complex Business Operations
├─────────────────┤
│   Models/ORM    │ ← SQLAlchemy Models & Database Layer
├─────────────────┤
│   Database      │ ← MySQL Database
└─────────────────┘
```

Key Architectural Principles

1. **Separation of Concerns**: Each layer has specific responsibilities
2. **Dependency Injection**: Database sessions injected via FastAPI dependencies
3. **Generic Controllers**: Base CRUD controller for code reuse
4. **Service Layer**: Complex business logic abstracted from controllers
5. **Schema Validation**: Pydantic models for request/response validation

Development Environment Setup

Prerequisites

- Python 3.8+
- MySQL 8.0+
- Redis (optional, for caching)
- Git

Installation Steps

1. **Clone the repository:**

```
git clone <repository-url>
cd restaurant-ordering-api
```

2. **Create virtual environment:**

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install dependencies:**

```
pip install -r requirements.txt
```

4. **Set up environment variables:**

```
export DB_HOST=localhost
export DB_NAME=restaurant_order_system
export DB_PORT=3306
export DB_USER=root
export DB_PASSWORD=rootroot
export APP_HOST=localhost
export APP_PORT=8000
```

5. **Create database:**

```
CREATE DATABASE restaurant_order_system;
```

6. **Run the application:**

```
python -m api.main
```

Dependencies

```
# Core Framework
fastapi>=0.104.0
uvicorn[standard]>=0.23.0

# Database
sqlalchemy>=2.0.0
pymysql>=1.1.0
alembic>=1.12.0

# Validation
pydantic>=2.0.0
pydantic[email]

# Utilities
python-multipart
redis>=4.0.0
```

Database Schema

Entity Relationship Diagram

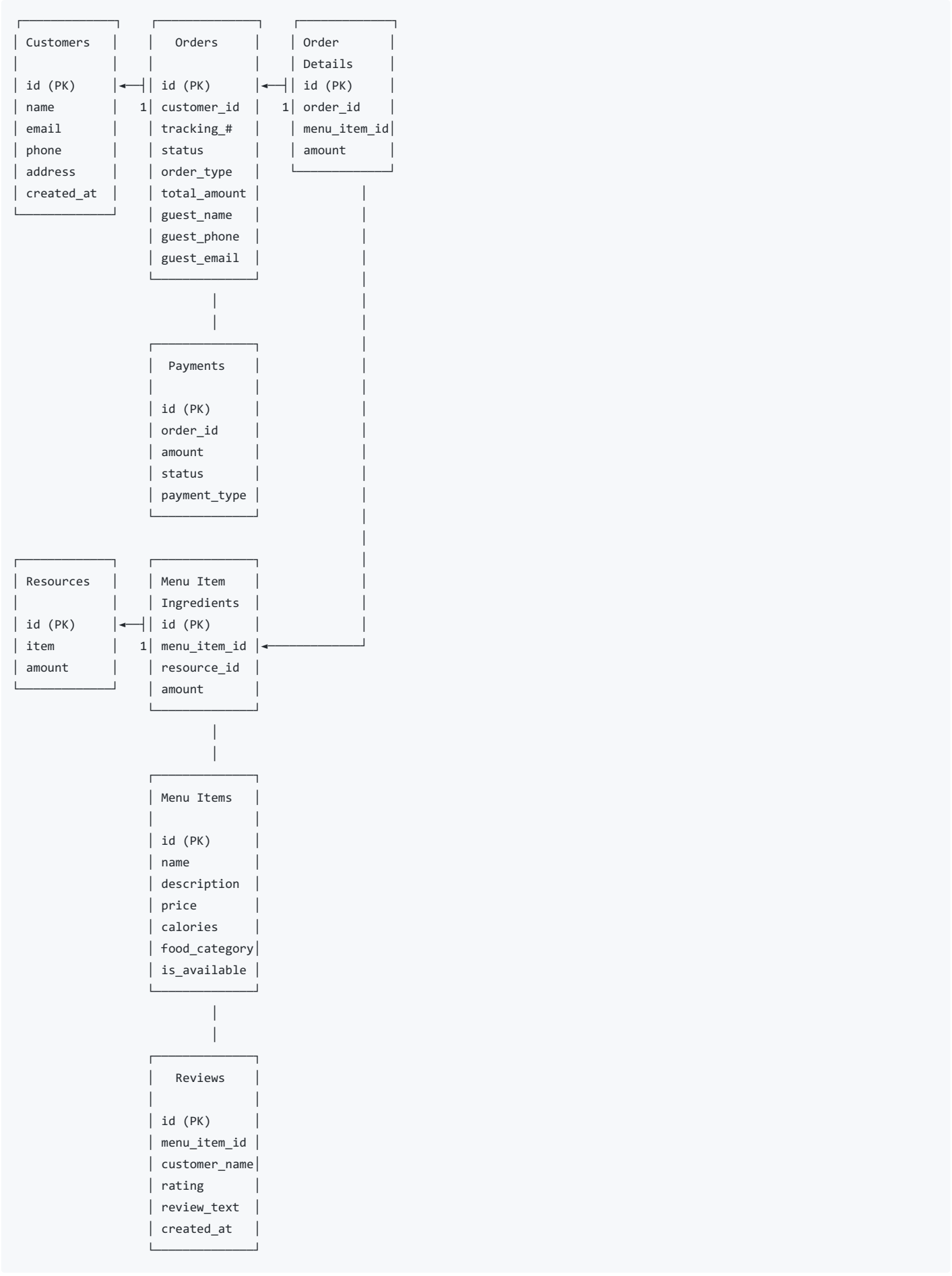

Table Definitions

```
 -- Customers table
CREATE TABLE customers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    customer_name VARCHAR(100),
    customer_email VARCHAR(100),
    customer_phone VARCHAR(100),
    customer_address VARCHAR(100),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Menu Items table
CREATE TABLE menu_items (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) UNIQUE NOT NULL,
    description VARCHAR(255),
    price DECIMAL(10,2) NOT NULL,
    calories INT NOT NULL,
    food_category ENUM('vegetarian','vegan','gluten_free','regular','keto','low_carb') DEFAULT 'regular',
    is_available BOOLEAN DEFAULT TRUE
);

-- Orders table
CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    tracking_number VARCHAR(20) UNIQUE,
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    description VARCHAR(300),
    status ENUM('pending','confirmed','in_progress','awaiting_pickup','out_for_delivery','cancelled','completed') DEFAULT 'pending',
    order_type ENUM('dine_in','takeout','delivery') DEFAULT 'dine_in',
    subtotal DECIMAL(10,2),
    tax_amount DECIMAL(10,2),
    discount_amount DECIMAL(10,2) DEFAULT 0,
    total_amount DECIMAL(10,2),
    promotion_code VARCHAR(50),
    guest_name VARCHAR(100),
    guest_phone VARCHAR(20),
    guest_email VARCHAR(100),
    estimated_completion DATETIME,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

API Endpoints

Authentication & Authorization

Currently, no authentication is required. Future versions will implement JWT-based authentication.

Endpoint Categories

Category	Prefix	Description
Customer Actions	`/customer_actions`	Public-facing customer operations
Staff Actions	`/staff_actions`	Staff management operations
Admin Actions	`/administrator_actions`	Administrative operations
Resources	Various	CRUD operations for all entities

Customer-Facing Endpoints

Menu Operations

```
GET /customer_actions/menu/search
Parameters:
  - search_term: Optional[str] - Search in name/description
  - category: Optional[FoodCategory] - Filter by food category
  - max_price: Optional[float] - Maximum price filter
  - sort_by: str - Sort criteria (name, price_asc, price_desc)
```

Order Operations

```
POST /customer_actions/orders/guest
Body:
{
  "guest_info": {
    "guest_name": "string",
    "guest_phone": "string",
    "guest_email": "string",
    "order_type": "takeout"
  },
  "order_items": [
    {"menu_item_id": 1, "quantity": 2}
  ]
}

GET /customer_actions/orders/track/{tracking_number}
Response:
{
  "id": 123,
  "tracking_number": "ORD4F2A8B1C",
  "status": "in_progress",
  "order_date": "2024-08-07T14:30:00Z",
  "total_amount": 28.47
}
```

Staff Management Endpoints

Inventory Management

```
GET /staff_actions/inventory/low-stock?threshold=10
POST /staff_actions/inventory/check-availability
Body: [{"menu_item_id": 1, "quantity": 2}]
```

Analytics

```
GET /staff_actions/analytics/menu-performance
GET /staff_actions/analytics/review-insights
GET /staff_actions/revenue/daily?target_date=2024-08-07
```

CRUD Endpoints

Menu Items

```
GET /menu_items/                    # List all items
POST /menu_items/                   # Create new item
GET /menu_items/{id}                # Get specific item
PUT /menu_items/{id}                # Update item
DELETE /menu_items/{id}             # Delete item
GET /menu_items/search              # Advanced search
PATCH /menu_items/{id}/availability # Toggle availability
```

Code Structure

```
api/
├── __init__.py
├── main.py                      # FastAPI application entry point
├── controllers/                 # Business logic controllers
│   ├── __init__.py
│   ├── base_controller.py       # Generic CRUD controller
│   ├── customers.py
│   ├── menu_items.py
│   ├── orders.py
│   └── ...
├── dependencies/                # FastAPI dependencies
│   ├── __init__.py
│   ├── config.py               # Configuration management
│   └── database.py             # Database connection
├── models/                      # SQLAlchemy models
│   ├── __init__.py
│   ├── customers.py
│   ├── menu_items.py
│   ├── orders.py
│   └── model_loader.py         # Database table creation
├── routers/                     # FastAPI route handlers
│   ├── __init__.py
│   ├── index.py                # Route registration
│   ├── customers.py
│   ├── menu_items.py
│   ├── customer_actions.py
│   └── staff_actions.py
├── schemas/                     # Pydantic validation models
│   ├── __init__.py
│   ├── customers.py
│   ├── menu_items.py
│   ├── orders.py
│   └── common.py
├── services/                    # Complex business logic
│   ├── __init__.py
│   ├── order_services.py
│   ├── menu_services.py
│   ├── inventory_services.py
│   └── analytics_services.py
└── utils/                       # Utility functions
    ├── __init__.py
    └── caching.py
```

Key Components

Base Controller Pattern

```
class BaseCRUDController(Generic[ModelType, CreateSchemaType, UpdateSchemaType]):
    def __init__(self, model: Type[ModelType]):
        self.model = model

    @handle_db_errors
    def create(self, db: Session, request: CreateSchemaType) -> ModelType:
        """Create a new item."""
        if hasattr(request, 'model_dump'):
            obj_data = request.model_dump()
        elif hasattr(request, 'dict'):
            obj_data = request.dict()

        new_item = self.model(**obj_data)
        db.add(new_item)
        db.commit()
        db.refresh(new_item)
        return new_item

    @handle_db_errors
    def read_all(self, db: Session) -> List[ModelType]:
        """Retrieve all items."""
        return db.query(self.model).all()
```

Error Handling Decorator

```
def handle_db_errors(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except SQLAlchemyError as e:
            error_msg = str(e.orig) if hasattr(e, 'orig') and e.orig else str(e)

            if "Duplicate entry" in error_msg:
                raise HTTPException(
                    status_code=status.HTTP_409_CONFLICT,
                    detail="Resource already exists"
                )
            elif "foreign key constraint" in error_msg.lower():
                raise HTTPException(
                    status_code=status.HTTP_400_BAD_REQUEST,
                    detail="Cannot complete operation due to related data"
                )
            else:
                raise HTTPException(
                    status_code=status.HTTP_400_BAD_REQUEST,
                    detail=f"Database error: {error_msg}"
                )
        except Exception as e:
            raise HTTPException(
                status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                detail=f"An unexpected error occurred: {str(e)}"
            )
    return wrapper
```

Database Configuration

```
class conf:
    db_host = os.getenv("DB_HOST", "localhost")
    db_name = os.getenv("DB_NAME", "restaurant_order_system")
    db_port = int(os.getenv("DB_PORT", "3306"))
    db_user = os.getenv("DB_USER", "root")
    db_password = os.getenv("DB_PASSWORD", "rootroot")

SQLALCHEMY_DATABASE_URL = f"mysql+pymysql://{conf.db_user}:{quote_plus(conf.db_password)}@{conf.db_host}:{conf.db_port}/{conf.db_nam

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

Service Layer

Order Service Example

```
class OrderService:
    @staticmethod
    def create_guest_order(db: Session, guest_info: Dict, order_items: List[Dict]):
        """Create order for guest customer"""

        # Calculate totals
        subtotal = 0
        order_details_data = []

        for item in order_items:
            menu_item = db.query(MenuItem).filter(MenuItem.id == item['menu_item_id']).first()
            if not menu_item:
                raise HTTPException(status_code=404, detail=f"Menu item {item['menu_item_id']} not found")

            if not menu_item.is_available:
                raise HTTPException(status_code=400, detail=f"{menu_item.name} is currently unavailable")

            item_total = float(menu_item.price) * item['quantity']
            subtotal += item_total

            order_details_data.append({
                'menu_item_id': item['menu_item_id'],
                'amount': item['quantity']
            })

        # Apply tax and discounts
        tax_rate = 0.08
        tax_amount = subtotal * tax_rate

        # Apply promotion if provided
        discount_amount = 0
        if guest_info.get('promotion_code'):
            promotion = db.query(Promotion).filter(
                Promotion.code == guest_info['promotion_code']
            ).first()
            if promotion:
                discount_amount = subtotal * (promotion.discount_percent / 100)

        total_amount = subtotal + tax_amount - discount_amount

        # Create order
        new_order = Order(
            guest_name=guest_info['guest_name'],
            guest_phone=guest_info['guest_phone'],
            guest_email=guest_info.get('guest_email'),
            description=guest_info.get('description'),
            order_type=guest_info.get('order_type', OrderType.DINE_IN),
```

```
            subtotal=subtotal,
            tax_amount=tax_amount,
            discount_amount=discount_amount,
            total_amount=total_amount,
            promotion_code=guest_info.get('promotion_code')
        )

        db.add(new_order)
        db.commit()
        db.refresh(new_order)

        # Create order details
        for detail_data in order_details_data:
            order_detail = OrderDetail(
                order_id=new_order.id,
                **detail_data
            )
            db.add(order_detail)

        db.commit()
        return new_order
```

Caching Strategy

Redis-based Caching with Fallback

```
class CacheManager:
    def __init__(self, redis_url: str = "redis://localhost:6379"):
        try:
            self.redis_client = redis.from_url(redis_url)
            self.redis_client.ping()
        except (ConnectionError, TimeoutError):
            self.redis_client = None
            self._memory_cache: Dict[str, Dict] = {}

    @cached(ttl=300, key_prefix="menu_search")
    def search_menu_items(self, db: Session, **kwargs):
        # Implementation cached for 5 minutes
        pass
```

Cache Invalidation

```
def _invalidate_menu_cache(self):
    """Invalidate menu-related cache entries"""
    cache.clear_pattern("menu_search:*")
    cache.clear_pattern("menu_item:*")
```

Error Handling

HTTP Status Codes

Code	Usage	Description
200	Success	Request processed successfully
201	Created	Resource created successfully
204	No Content	Successful deletion
400	Bad Request	Invalid input data or business rule violation

Code	Usage	Description
404	Not Found	Resource not found
409	Conflict	Resource already exists or constraint violation
422	Unprocessable Entity	Invalid data format
500	Server Error	Unexpected server error

Error Response Format

```
{
  "detail": "Specific error message describing the issue"
}
```

Custom Exception Handling

```
@handle_db_errors
def create(self, db: Session, request: CreateSchemaType) -> ModelType:
    existing_item = db.query(self.model).filter(
        self.model.name == request.name
    ).first()
    if existing_item:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail=f"Resource with name '{request.name}' already exists"
        )
```

Testing

Unit Testing Structure

```
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from api.main import app
from api.dependencies.database import get_db

# Test database setup
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

def override_get_db():
    try:
        db = TestingSessionLocal()
        yield db
    finally:
        db.close()

app.dependency_overrides[get_db] = override_get_db
client = TestClient(app)

def test_create_menu_item():
    response = client.post(
        "/menu_items/",
        json={
            "name": "Test Burger",
            "description": "A test burger",
            "price": 10.99,
            "calories": 500,
            "food_category": "regular"
        }
    )
    assert response.status_code == 201
    assert response.json()["name"] == "Test Burger"
```

Integration Testing

```
def test_guest_order_workflow():
    # Create menu items
    menu_response = client.post("/menu_items/", json=menu_item_data)
    menu_item_id = menu_response.json()["id"]

    # Place guest order
    order_response = client.post(
        "/customer_actions/orders/guest",
        json={
            "guest_info": guest_data,
            "order_items": [{"menu_item_id": menu_item_id, "quantity": 2}]
        }
    )
    assert order_response.status_code == 201

    # Track order
    tracking_number = order_response.json()["tracking_number"]
    track_response = client.get(f"/customer_actions/orders/track/{tracking_number}")
    assert track_response.status_code == 200
```

Deployment

Production Environment Variables

```
# Database Configuration
DB_HOST=production-mysql-host
DB_NAME=restaurant_order_system
DB_USER=api_user
DB_PASSWORD=secure_password
DB_PORT=3306

# Application Configuration
APP_HOST=0.0.0.0
APP_PORT=8000

# Optional: Redis Configuration
REDIS_URL=redis://redis-host:6379
```

Docker Configuration

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```