Restaurant API Technical Documentation

Snack Overflow

Tevin Parathattal, Nicholas Smit, Preston Murray, Jason Springer-Trammell, Walker Thierbach

August 7, 2025

Architecture Overview

The Snack Overflow Restaurant API follows a layered architecture pattern built with FastAPI and SQLAlchemy ORM. The system is designed with clear separation of concerns and follows industry best practices for REST API development.

Core Components

1. Application Layer (FastAPI)

- Framework: FastAPI with Uvicorn ASGI server
- Port: 8000 (configurable)
- Features:
  - Automatic OpenAPI/Swagger documentation
  - Request/Response validation
  - Dependency injection
  - CORS middleware for cross-origin requests

2. Router Layer

- Purpose: Define API endpoints and route HTTP requests
- Structure: Organized by resource type (customers, orders, menu items, etc.)
- Location: /api/routers/
- Key Features:
  - RESTful endpoint definitions
  - HTTP method routing (GET, POST, PUT, DELETE)
  - Path and query parameter handling
  - Response model specification

3. Controller Layer (Business Logic)

- Purpose: Handle business logic and data processing
- Location: /api/controllers/
- Responsibilities:
  - Data validation and transformation
  - Business rule enforcement
  - Database operations coordination
  - Error handling and response formatting

4. Schema Layer (Data Validation)

- Framework: Pydantic models
- Location: /api/schemas/
- Purpose:
  - Request/Response data validation
  - Type safety and serialization
  - API contract definition
- Types:
  - Create schemas for POST requests
  - Update schemas for PUT requests
  - Response schemas for API responses

5. Model Layer (Data Access)

- Framework: SQLAlchemy ORM
- Location: /api/models/
- Features:
  - Database table definitions
  - Relationship mappings
  - Database constraints and indexes
  - Object-relational mapping

6. Database Layer

- Primary: MySQL (Production)
- Development: MySQL (Development & Testing)
- Features:
  - Automatic database creation
  - Migration support through SQLAlchemy
  - Connection pooling
  - Environment-based configuration

Key Design Patterns

1. Repository Pattern

- Controllers act as repositories for data access
- Separation of data access logic from business logic
- Consistent interface for database operations

2. Dependency Injection

- Database sessions injected via FastAPI's dependency system
- Configuration management through dependency injection
- Easy testing and mocking capabilities

3. MVC Architecture

- Models: SQLAlchemy ORM models
- Views: FastAPI routers (API endpoints)
- Controllers: Business logic handlers

4. Domain-Driven Design

- Clear domain boundaries (Customer, Order, Menu, etc.)
- Rich domain models with relationships
- Business logic encapsulated in controllers

Data Flow

1. Request Lifecycle:

Client Request → Router → Controller → Model → Database
↓
Client Response ← Schema ← Controller ← Model ← Database

2. Key Operations:

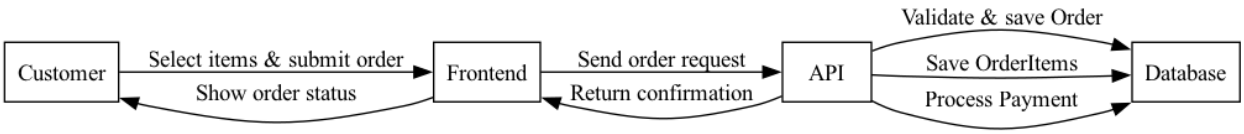
- Validation: Pydantic schemas validate all input/output
- Database: SQLAlchemy manages all database interactions
- Error Handling: Centralized error responses with proper HTTP codes

Testing Architecture

- Test Database: Separate MySQL database for testing
- Environment Switching: Automatic test environment detection
- Isolated Tests: Each test uses fresh database state

Diagrams

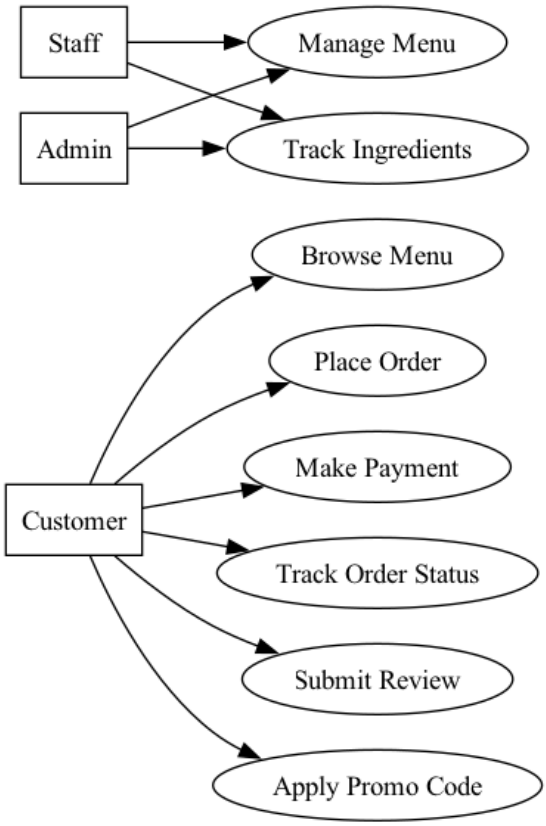
Sequence



This diagram shows each element necessary for our system to work. It also shows and explains the interactions between each element.

A Customer selects their items and submits their order to the Frontend application, which sends the request to our API. The order, items, and payment are saved in our Database, before returning confirmation to the Frontend and showing the status of the order to the Customer.

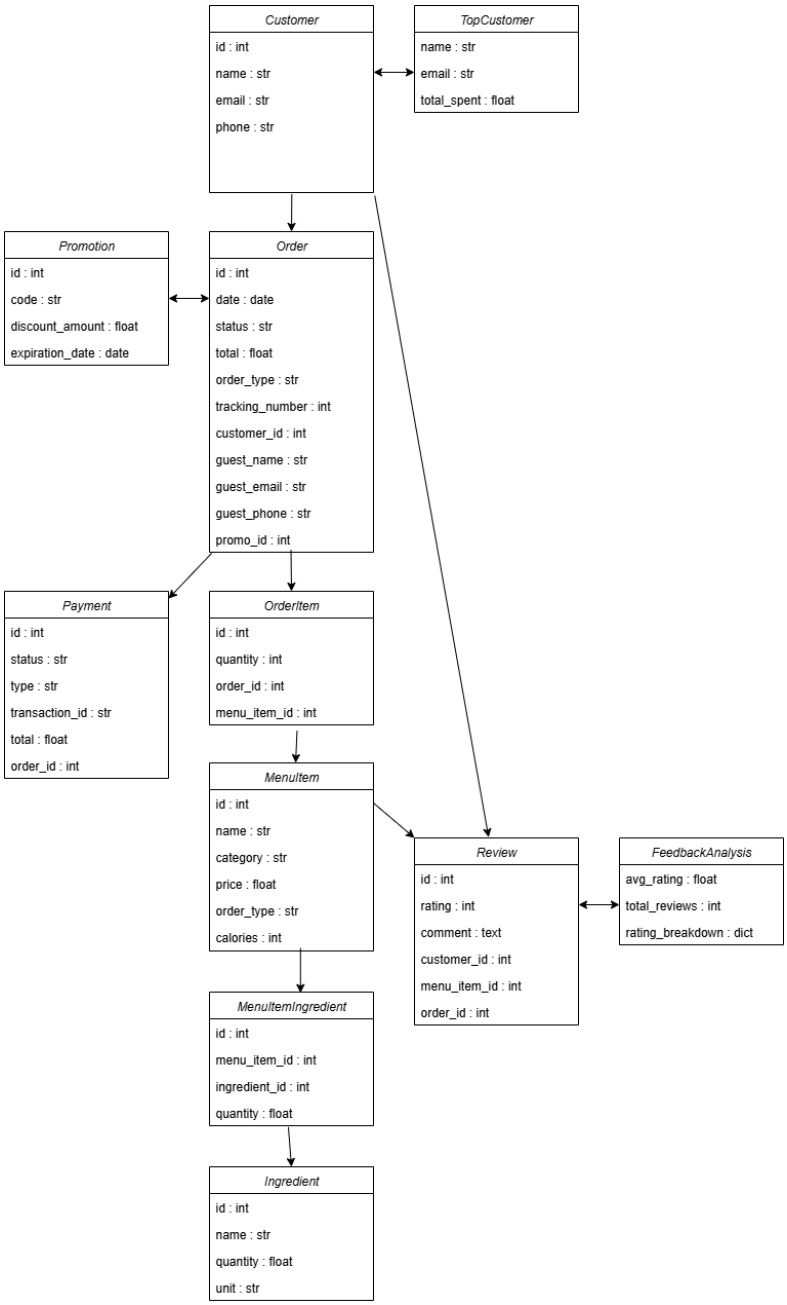
Use Case



This diagram explains the use cases and needs of each party involved in our system.

The Staff and the Admin must both be able to track ingredients and manage the menu in order to keep the system functioning. Customers must be able to browse the menu for their desired order, then be able to place their order. They must also be given the option to apply a promo code, pay for, and track their orders. Finally, reviews can be submitted to alert the staff and admin of any issues / praises they have for the system.

Class



This diagram shows each major class in our system, as well as how they relate to each other. Everything starts with the Customer, who is able to Order and place a Review. Order is linked with Promotion and from there, the Payment class is available. The items ordered are also sent to OrderItem, then to MenuItem to check that said item is available. Then the ingredients are sent through MenuItemIngredient to Ingredient. Reviews that are placed are linked to the FeedbackAnalysis, which handles every review. TopCustomers are also linked to Customer, allowing our restaurant to take pride in our most valuable customers.

Endpoint Documentation

Overview

This is the API documentation for the Snack Overflow restaurant management system. The API is built using FastAPI and provides endpoints for managing customers, menu items, orders, payments, ingredients, promotions, reviews, and analytics.

Base URL: http://localhost:8000
http://localhost:8000/docs to access Swagger UI

Table of Contents

Authentication

Currently, this API does not implement authentication. All endpoints are publicly accessible.

CUSTOMERS

Create Customer

Creates a new customer in the system.

Endpoint: POST /customer

Request Body:

```
{
  "name": "John Doe",
  "email": "john.doe@example.com",
  "phone": "555-123-4567"
}
```

Response: 201 Created

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com",
  "phone": "555-123-4567"
}
```

Get All Customers

Retrieves a list of all customers.

Endpoint: GET /customer

Response: 200 OK

```
[
  {
    "id": 1,
    "name": "John Doe",
    "email": "john.doe@example.com",
    "phone": "555-123-4567"
  },
  {
```

```
    "id": 2,
    "name": "Jane Smith",
    "email": "jane.smith@example.com",
    "phone": "555-987-6543"
  }
]
```


Get Customer by ID

Retrieves a specific customer by their ID.

Endpoint: GET /customer/{customer_id}

Path Parameters:
- customer_id (integer): The unique identifier of the customer

Response: 200 OK


```
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com",
  "phone": "555-123-4567"
}
```


Update Customer

Updates an existing customer's information.

Endpoint: PUT /customer/{customer_id}

Path Parameters:
- customer_id (integer): The unique identifier of the customer

Request Body:


```
{
  "name": "John Updated Doe",
  "email": "john.updated@example.com",
  "phone": "555-111-2222"
}
```


Response: 200 OK


```
{
  "id": 1,
  "name": "John Updated Doe",
```

```
  "email": "john.updated@example.com",
  "phone": "555-111-2222"
}
```


Delete Customer

Deletes a customer from the system.

Endpoint: DELETE /customer/{customer_id}

Path Parameters:
- customer_id (integer): The unique identifier of the customer

Response: 204 No Content

MENU ITEMS

Create Menu Item

Creates a new menu item.

Endpoint: POST /menu_item

Request Body:


```
{
  "name": "Classic Burger",
  "category": "Main Course",
  "price": 12.99,
  "calories": 650,
  "tags": "spicy"
}
```


Response: 201 Created


```
{
  "id": 1,
  "name": "Classic Burger",
  "category": "Main Course",
  "price": 12.99,
  "calories": 650,
  "tags": "spicy"
}
```

Get All Menu Items

Retrieves all menu items.

Endpoint: GET /menu_item

Response: 200 OK


```
[
  {
    "id": 1,
    "name": "Classic Burger",
    "category": "Main Course",
    "price": 12.99,
    "calories": 650,
    "tags": "spicy"
  },
  {
    "id": 2,
    "name": "Caesar Salad",
    "category": "Salad",
    "price": 8.99,
    "calories": 320,
    "tags": "vegetarian"
  }
]
```


Get Menu Items by Tag

Retrieves menu items filtered by a specific tag(vegetarian, spicy, etc.)

Endpoint: GET /menu_item/by_tag

Query Parameters:
- tag (string): The tag to filter by

Example: GET /menu_item/by_tag?tag=spicy

Response: 200 OK


```
[
  {
    "id": 1,
    "name": "Classic Burger",
    "category": "Main Course",
    "price": 12.99,
```

```
    "calories": 650,
    "tags": "spicy"
  }
]
```


Search Menu Items

Searches for menu items by name.

Endpoint: GET /menu_item/search

Query Parameters:
- q (string): Search query (minimum 1 character)

Example: GET /menu_item/search?q=burger

Response: 200 OK


```
[
  {
    "id": 1,
    "name": "Classic Burger",
    "category": "Main Course",
    "price": 12.99,
    "calories": 650,
    "tags": "spicy"
  }
]
```


Get Menu Items by Category

Retrieves menu items filtered by category.

Endpoint: GET /menu_item/by_category/{category}

Path Parameters:
- category (string): The category to filter by

Example: GET /menu_item/by_category/Main Course

Response: 200 OK


```
[
  {
    "id": 1,
    "name": "Classic Burger",
```

```
    "category": "Main Course",
    "price": 12.99,
    "calories": 650,
    "tags": "spicy"
  }
]
```


Get Menu Items by Rating

Retrieves menu items filtered by minimum rating.

Endpoint: GET /menu_item/by_rating/{rating}

Path Parameters:
- rating (integer): Minimum rating (1-5)

Example: GET /menu_item/by_rating/4

Response: 200 OK


```
[
  {
    "id": 1,
    "name": "Classic Burger",
    "category": "Main Course",
    "price": 12.99,
    "calories": 650,
    "tags": "spicy"
  }
]
```


Get Menu Item Popularity Insights

Retrieves popularity insights for menu items.

Endpoint: GET /menu_item/popularity_insights

Response: 200 OK


```
[
  {
    "menu_item_id": 1,
    "name": "Caesar Salad",
    "order_count": 134,
    "complaint_count": 3
  },
```

```
  {
    "menu_item_id": 2,
    "name": "BLT Sandwich",
    "order_count": 64,
    "complaint_count": 4
  }
]
```


Get Menu Item by ID

Retrieves a specific menu item by its ID.

Endpoint: GET /menu_item/{menu_item_id}

Path Parameters:
- menu_item_id (integer): The unique identifier of the menu item

Response: 200 OK


```
{
  "id": 1,
  "name": "Classic Burger",
  "category": "Main Course",
  "price": 12.99,
  "calories": 650,
  "tags": "spicy"
}
```


Update Menu Item

Updates an existing menu item.

Endpoint: PUT /menu_item/{menu_item_id}

Path Parameters:
- menu_item_id (integer): The unique identifier of the menu item

Request Body:


```
{
  "id": 1,
  "name": "Deluxe Burger",
  "category": "Main Course",
  "price": 14.99,
  "calories": 750,
```

```
  "tags": "spicy"
}
```


Response: 200 OK


```
{
  "id": 1,
  "name": "Deluxe Burger",
  "category": "Main Course",
  "price": 14.99,
  "calories": 750,
  "tags": "spicy"
}
```


Delete Menu Item

Deletes a menu item from the system.

Endpoint: DELETE /menu_item/{menu_item_id}

Path Parameters:
- menu_item_id (integer): The unique identifier of the menu item

Response: 204 No Content

ORDERS

Create Order

Creates a new order in the system.

Endpoint: POST /order

Request Body:


```
{
  "date": "2024-01-15T12:30:00",
  "status": "pending",
  "order_type": "dine-in",
  "customer_id": 1,
  "guest_name": null,
  "guest_email": null,
  "guest_phone": null,
```

```
 "promotion_id": null,
 "payments": [
  {
   "status": "completed",
   "type": "credit_card",
   "transaction_id": "txn_123456789",
   "total": 25.98
  }
 ],
 "menu_item_ids": [1, 2]
}
```


Response: 201 Created


```
{
 "id": 1,
 "date": "2024-01-15T12:30:00",
 "status": "pending",
 "total": 25.98,
 "order_type": "dine-in",
 "tracking_number": 1001,
 "customer_id": 1,
 "guest_name": null,
 "guest_email": null,
 "guest_phone": null,
 "promotion_id": null,
 "payments": [
  {
   "id": 1,
   "status": "completed",
   "type": "credit_card",
   "transaction_id": "txn_123456789",
   "total": 25.98,
   "order_id": 1
  }
 ]
}
```


Get All Orders

Retrieves all orders in the system.

Endpoint: GET /order

Response: 200 OK

```
[
  {
    "id": 1,
    "date": "2024-01-15T12:30:00",
    "status": "pending",
    "total": 25.98,
    "order_type": "dine-in",
    "tracking_number": 1001,
    "customer_id": 1,
    "guest_name": null,
    "guest_email": null,
    "guest_phone": null,
    "promotion_id": null,
    "payments": []
  }
]
```


Get Order by ID

Retrieves a specific order by its ID.

Endpoint: GET /order/{order_id}

Path Parameters:
- order_id (integer): The unique identifier of the order

Response: 200 OK


```
{
  "id": 1,
  "date": "2024-01-15T12:30:00",
  "status": "pending",
  "total": 25.98,
  "order_type": "dine-in",
  "tracking_number": 1001,
  "customer_id": 1,
  "guest_name": null,
  "guest_email": null,
  "guest_phone": null,
  "promotion_id": null,
  "payments": []
}
```

Get Orders by Customer

Retrieves all orders for a specific customer by id.

Endpoint: GET /order/by_customer/{customer_id}

Path Parameters:
- customer_id (integer): The unique identifier of the customer

Response: 200 OK


```
[
  {
    "id": 1,
    "date": "2024-01-15T12:30:00",
    "status": "pending",
    "total": 25.98,
    "order_type": "dine-in",
    "tracking_number": 1001,
    "customer_id": 1,
    "guest_name": null,
    "guest_email": null,
    "guest_phone": null,
    "promotion_id": null,
    "payments": []
  }
]
```


Get Orders by Date Range

Retrieves orders within a specific date range.

Endpoint: GET /order/by_date/

Query Parameters:
- start_date (string): Start date in YYYY-MM-DD format
- end_date (string): End date in YYYY-MM-DD format

Example: GET /order/by_date/?start_date=2024-01-01&end_date=2024-01-31

Response: 200 OK


```
[
  {
    "id": 1,
    "date": "2024-01-15T12:30:00",
```

```
    "status": "pending",
    "total": 25.98,
    "order_type": "dine-in",
    "tracking_number": 1001,
    "customer_id": 1,
    "guest_name": null,
    "guest_email": null,
    "guest_phone": null,
    "promotion_id": null,
    "payments": []
  }
]
```


Track Order by Tracking Number

Retrieves an order by its tracking number.

Endpoint: GET /order/track/{tracking_number}

Path Parameters:
- tracking_number (integer): The tracking number of the order

Response: 200 OK


```
{
  "id": 1,
  "date": "2024-01-15T12:30:00",
  "status": "pending",
  "total": 25.98,
  "order_type": "dine-in",
  "tracking_number": 1001,
  "customer_id": 1,
  "guest_name": null,
  "guest_email": null,
  "guest_phone": null,
  "promotion_id": null,
  "payments": []
}
```


Update Order

Updates an existing order.

Endpoint: PUT /order/{order_id}

Path Parameters:
- order_id (integer): The unique identifier of the order

Request Body:

```
{
  "status": "completed",
  "order_type": "takeout"
}
```


Response: 200 OK

```
{
  "id": 1,
  "date": "2024-01-15T12:30:00",
  "status": "completed",
  "total": 25.98,
  "order_type": "takeout",
  "tracking_number": 1001,
  "customer_id": 1,
  "guest_name": null,
  "guest_email": null,
  "guest_phone": null,
  "promotion_id": null,
  "payments": []
}
```


Delete Order

Deletes an order from the system.

Endpoint: DELETE /order/{order_id}

Path Parameters:
- order_id (integer): The unique identifier of the order

Response: 204 No Content

ORDER ITEMS

Create Order Item

Creates a new order item.

Endpoint: POST /order_item

Request Body:

```
{
  "order_id": 1,
  "menu_item_id": 1,
  "quantity": 2,
}
```


Response: 201 Created

```
{
  "id": 1,
  "order_id": 1,
  "menu_item_id": 1,
  "quantity": 2,
}
```


Get All Order Items

Retrieves all order items.

Endpoint: GET /order_item

Response: 200 OK

```
[
  {
    "id": 1,
    "order_id": 1,
    "menu_item_id": 1,
    "quantity": 2,
  }
]
```


Get Order Item by ID

Retrieves a specific order item by its ID.

Endpoint: GET /order_item/{order_item_id}

Path Parameters:
- order_item_id (integer): The unique identifier of the order item

Response: 200 OK

```
{
```

```
 "id": 1,
 "order_id": 1,
 "menu_item_id": 1,
 "quantity": 2,
}
```


Update Order Item

Updates an existing order item.

Endpoint: PUT /order_item/{order_item_id}

Path Parameters:
- order_item_id (integer): The unique identifier of the order item

Request Body:


```
{
 "quantity": 3,
 "": "Extra cheese, no onions"
}
```


Response: 200 OK


```
{
 "id": 1,
 "order_id": 1,
 "menu_item_id": 1,
 "quantity": 3,
}
```


Delete Order Item

Deletes an order item from the system.

Endpoint: DELETE /order_item/{order_item_id}

Path Parameters:
- order_item_id (integer): The unique identifier of the order item

Response: 204 No Content

PAYMENTS

Create Payment

Creates a new payment record.

Endpoint: POST /payment

Request Body:

```
{
  "status": "completed",
  "type": "credit_card",
  "transaction_id": "txn_987654321",
  "total": 25.98,
  "order_id": 1
}
```

Response: 201 Created

```
{
  "id": 1,
  "status": "completed",
  "type": "credit_card",
  "transaction_id": "txn_987654321",
  "total": 25.98,
  "order_id": 1
}
```

Get All Payments

Retrieves all payment records.

Endpoint: GET /payment

Response: 200 OK

```
[
  {
    "id": 1,
    "status": "completed",
    "type": "credit_card",
    "transaction_id": "txn_987654321",
    "total": 25.98,
    "order_id": 1
  }
]
```

Get Payment by ID

Retrieves a specific payment by its ID.

Endpoint: GET /payment/{payment_id}

Path Parameters:
- payment_id (integer): The unique identifier of the payment. It is returned in the response when creating an order in the payments section.

Response: 200 OK

```
{
  "id": 1,
  "status": "completed",
  "type": "credit_card",
  "transaction_id": "txn_987654321",
  "total": 25.98,
  "order_id": 1
}
```


Update Payment

Updates an existing payment record.

Endpoint: PUT /payment/{payment_id}

Path Parameters:
- payment_id (integer): The unique identifier of the payment. It is returned in the response when creating an order in the payments section.

Request Body:

```
{
  "status": "refunded",
  "type": "credit_card"
}
```


Response: 200 OK

```
{
  "id": 1,
  "status": "refunded",
  "type": "credit_card",
  "transaction_id": "txn_987654321",
  "total": 25.98,
```

```
  "order_id": 1
}
```


Delete Payment

Deletes a payment record from the system.

Endpoint: DELETE /payment/{payment_id}

Path Parameters:
- payment_id (integer): The unique identifier of the payment. It is returned in the response when creating an order in the payments section.

Response: 204 No Content

Get Revenue Report

Retrieves revenue data within a date range.

Endpoint: GET /payment/revenue/

Query Parameters:
- start_date (string): Start date in YYYY-MM-DD format
- end_date (string): End date in YYYY-MM-DD format

Example: GET /payment/revenue/?start_date=2024-01-01&end_date=2024-01-31

Response: 200 OK


```
{
  "total_revenue": 1250.75,
  "total_orders": 45,
  "total_customers": 32
}
```


INGREDIENTS

Create Ingredient

Creates a new ingredient in the inventory.

Endpoint: POST /ingredient

Request Body:

```
{
  "name": "Ground Beef",
  "quantity": 100,
  "unit": "lbs",
  "cost_per_unit": 6.99
}
```

Response: 201 Created

```
{
  "id": 1,
  "name": "Ground Beef",
  "quantity": 100,
  "unit": "lbs",
  "cost_per_unit": 6.99
}
```

Get All Ingredients

Retrieves all ingredients in the inventory.

Endpoint: GET /ingredient

Response: 200 OK

```
[
  {
    "name": "Lettuce",
    "quantity": 100,
    "unit": "grams",
    "id": 1
  },
  {
    "name": "Tomato",
    "quantity": 80,
    "unit": "grams",
    "id": 2
  },
]
```

Get Ingredient by ID

Retrieves a specific ingredient by its ID.

Endpoint: GET /ingredient/{ingredient_id}

Path Parameters:
- ingredient_id (integer): The unique identifier of the ingredient

Response: 200 OK

```
{
  "name": "Lettuce",
  "quantity": 100,
  "unit": "grams",
  "id": 1
}
```


Update Ingredient

Updates an existing ingredient.

Endpoint: PUT /ingredient/{ingredient_id}

Path Parameters:
- ingredient_id (integer): The unique identifier of the ingredient

Request Body:

```
{
  "quantity": 75
}
```


Response: 200 OK

```
{
  "id": 1,
  "name": "Ground Beef",
  "quantity": 75,
  "unit": "lbs"
}
```


Delete Ingredient

Deletes an ingredient from the inventory.

Endpoint: DELETE /ingredient/{ingredient_id}

Path Parameters:
- ingredient_id (integer): The unique identifier of the ingredient

Response: 204 No Content

MENU ITEM INGREDIENTS

Create Menu Item Ingredient

Associates an ingredient with a menu item and specifies the quantity needed.

Endpoint: POST /menu_item_ingredient

Request Body:

```
{
  "menu_item_id": 1,
  "ingredient_id": 1,
  "quantity": 0.25
}
```


Response: 201 Created

```
{
  "id": 1,
  "menu_item_id": 1,
  "ingredient_id": 1,
  "quantity": 0.25
}
```


Error Response (Insufficient Stock): 422 Unprocessable Entity

```
{
  "detail": "Not enough stock for ingredient Ground Beef. Available: 10, required: 20"
}
```


Get All Menu Item Ingredients

Retrieves all menu item ingredient associations.

Endpoint: GET /menu_item_ingredient

Response: 200 OK

```
[
  {
    "id": 1,
    "menu_item_id": 1,
```

```
    "ingredient_id": 1,
    "quantity": 0.25
  }
]
```


Get Menu Item Ingredient by ID

Retrieves a specific menu item ingredient association by its ID.

Endpoint: GET /menu_item_ingredient/{mii_id}

Path Parameters:
- mii_id (integer): The unique identifier of the menu item ingredient

Response: 200 OK


```
{
  "id": 1,
  "menu_item_id": 1,
  "ingredient_id": 1,
  "quantity": 0.25
}
```


Update Menu Item Ingredient

Updates an existing menu item ingredient association.

Endpoint: PUT /menu_item_ingredient/{mii_id}

Path Parameters:
- mii_id (integer): The unique identifier of the menu item ingredient

Request Body:


```
{
  "quantity": 0.5
}
```


Response: 200 OK


```
{
  "id": 1,
  "menu_item_id": 1,
  "ingredient_id": 1,
  "quantity": 0.5
}
```

Delete Menu Item Ingredient

Removes an ingredient association from a menu item.

Endpoint: DELETE /menu_item_ingredient/{mii_id}

Path Parameters:
- mii_id (integer): The unique identifier of the menu item ingredient

Response: 204 No Content

PROMOTIONS

Create Promotion

Creates a new promotional offer.

Endpoint: POST /promotion

Request Body:

```
{
  "code": "Summer Special",
  "discount_amount": 25,
  "expiration_date": "2025-08-07T02:57:18.446Z"
}
```


Response: 201 Created

```
{
  "code": "Summer Special",
  "discount_amount": 25,
  "expiration_date": "2025-08-07T02:57:18.459Z",
  "id": 1
}
```


Get All Promotions

Retrieves all promotional offers.

Endpoint: GET /promotion

Response: 200 OK

```
[
  {
```

```
    "code": "WELCOME25",
    "discount_amount": 25,
    "expiration_date": "2025-12-31T00:00:00",
    "id": 1
  }
]
```


Get Promotion by ID

Retrieves a specific promotion by its ID.

Endpoint: GET /promotion/{promotion_id}

Path Parameters:
- promotion_id (integer): The unique identifier of the promotion

Response: 200 OK


```
{
  "code": "WELCOME25",
  "discount_amount": 25,
  "expiration_date": "2025-12-31T00:00:00",
  "id": 1
}
```


Update Promotion

Updates an existing promotion.

Endpoint: PUT /promotion/{promotion_id}

Path Parameters:
- promotion_id (integer): The unique identifier of the promotion

Request Body:


```
{
  "discount_amount": 30
}
```


Response: 200 OK


```
{
  "id": 1,
  "name": "Summer Special",
  "discount_amount": 30,
```

"expiration_date": "2025-08-07T02:57:18.459Z"
}

Delete Promotion

Deletes a promotion from the system.

Endpoint: DELETE /promotion/{promotion_id}

Path Parameters:
- promotion_id (integer): The unique identifier of the promotion

Response: 204 No Content

REVIEWS

Create Review

Creates a new customer review for a menu item.

Endpoint: POST /review

Request Body:
{
"customer_id": 1,
"menu_item_id": 1,
"rating": 5,
"comment": "Absolutely delicious! Best burger I've ever had.",
"date": "2024-01-15T14:30:00"
}

Response: 201 Created
{
"id": 1,
"customer_id": 1,
"menu_item_id": 1,
"rating": 5,
"comment": "Absolutely delicious! Best burger I've ever had.",
"date": "2024-01-15T14:30:00"
}

Get All Reviews

Retrieves all customer reviews.

Endpoint: GET /review

Response: 200 OK

```
[
  {
    "id": 1,
    "customer_id": 1,
    "menu_item_id": 1,
    "rating": 5,
    "comment": "Absolutely delicious! Best burger I've ever had.",
    "date": "2024-01-15T14:30:00"
  }
]
```


Get Review by ID

Retrieves a specific review by its ID.

Endpoint: GET /review/{review_id}

Path Parameters:
- review_id (integer): The unique identifier of the review

Response: 200 OK

```
{
  "id": 1,
  "customer_id": 1,
  "menu_item_id": 1,
  "rating": 5,
  "comment": "Absolutely delicious! Best burger I've ever had.",
  "date": "2024-01-15T14:30:00"
}
```


Get Reviews by Menu Item

Retrieves all reviews for a specific menu item.

Endpoint: GET /review/menu/{menu_item_id}

Path Parameters:

- menu_item_id (integer): The unique identifier of the menu item

Response: 200 OK

```
[
  {
    "id": 1,
    "customer_id": 1,
    "menu_item_id": 1,
    "rating": 5,
    "comment": "Absolutely delicious! Best burger I've ever had.",
    "date": "2024-01-15T14:30:00"
  },
  {
    "id": 2,
    "customer_id": 2,
    "menu_item_id": 1,
    "rating": 4,
    "comment": "Great taste, but a bit pricey.",
    "date": "2024-01-16T12:00:00"
  }
]
```


Update Review

Updates an existing review.

Endpoint: PUT /review/{review_id}

Path Parameters:
- review_id (integer): The unique identifier of the review

Request Body:

```
{
  "rating": 4,
  "comment": "Good burger, but could use more seasoning."
}
```


Response: 200 OK

```
{
  "id": 1,
  "customer_id": 1,
  "menu_item_id": 1,
  "rating": 4,
  "comment": "Good burger, but could use more seasoning.",
```

```
  "date": "2024-01-15T14:30:00"
}
```


Get Feedback Analysis

Retrieves comprehensive feedback analysis across all reviews.

Endpoint: GET /review/analysis

Response: 200 OK


```
{
  "average_rating": 3.83,
  "total_reviews": 6,
  "ratings_breakdown": {
    "2": 1,
    "3": 1,
    "4": 2,
    "5": 2
  }
}
```


ANALYTICS

Get Top Customers

Retrieves the top 3 customers by total spending.

Endpoint: GET /analytics/top-customers

Response: 200 OK


```
[
  {
    "name": "John Doe",
    "email": "john.doe@example.com",
    "total_spent": 450.75
  },
  {
    "name": "Jane Smith",
    "email": "jane.smith@example.com",
    "total_spent": 320.50
  },
  {
```

```
    "name": "Bob Johnson",
    "email": "bob.johnson@example.com",
    "total_spent": 275.25
  }
]
```


ERROR RESPONSES

The API uses standard HTTP status codes to indicate the success or failure of requests.

Common Error Codes

400 Bad Request


```
{
  "detail": "Invalid request data"
}
```


404 Not Found


```
{
  "detail": "Resource not found"
}
```


422 Unprocessable Entity


```
{
  "detail": [
    {
      "loc": ["body", "email"],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```


500 Internal Server Error


```
{
  "detail": "Internal server error"
}
```

DATA TYPES AND ENUMS

Order Types
- dine-in: Customer dining in the restaurant
- takeout: Customer picking up order
- delivery: Order delivered to customer

Payment Types
- cash
- credit_card
- debit_card
- mobile_payment

Payment Status
- pending
- completed
- failed
- refunded

Order Status
- pending
- confirmed
- preparing
- ready
- completed

- cancelled

NOTES

1. All datetime fields should be provided in format: YYYY-MM-DD
2. All monetary values are in USD and should be provided as floating-point numbers
3. Database is MySQL-based with SQLite support and automatic table creation on startup
4. The system includes automatic seeding of initial data if the database is empty and permissions are granted in config.py.

Testing

PyTest is required for the provided testing portion of this API.

Testing Setup

The conftest.py file is executed before each test can be run. It first wipes the database to ensure no overlapping information is created, then populates the database with all necessary information:

- customer
- ingredient
- menu_item
- menu_item_ingredient
- order
- order_item
- payment
- promotion
- review

This information is used for each test and ensures the database is properly populated and accessible.

test_customer.py

test_read_customer

- Receives response through "/customer/{test_data['customer_id']}"
- Checks that the response status is 200
- Checks that the returned customer data matches the information in conftest.py

test_delete_customer

- Receives response through "/customer/{test_data['customer_id']}"
- Checks that the response status is 200 or 204
- Checks that the customer was deleted by expecting 404 or 422 when attempting to retrieve it again

test_analytics.py

test_top_customers_analytics

- Receives response through "/analytics/top-customers"
- Checks that the response status is 200
- Checks that the response is a list with at most 3 items
- Checks that each item contains "name", "email", and "total_spent"
- Checks that "total_spent" is a float

test_ingredient.py

test_delete_ingredient

- Receives response through "/ingredient/{test_data['ingredient_id']}"
- Checks the response status code is valid (200 or 204)
- Checks that the ingredient was successfully deleted by expecting a 404 or 422 when trying to retrieve it again

test_track_ingredient_inventory

- Receives response through "/ingredient/"
- Checks that the response is successful (status code 200)
- Checks that the returned list of ingredients is valid and non-empty
- Finds the ingredient by ID within the list
- Checks that the ingredient exists in the list
- Checks that the quantity field is present, is numeric (int or float), and is non-negative

test_ingredient_stock_validation

- Receives response through "/ingredient/{test_data['ingredient_id']}"
- Retrieves the available quantity of the ingredient
- Sends a POST request to "/menu_item_ingredient/" with an excessive quantity
- Checks that the response is a failure (status code 400 or 422) for excessive quantity
- Sends a POST request to "/menu_item_ingredient/" with a valid quantity
- Checks that the request succeeds (status code 200 or 201) when the quantity is acceptable

test_menu_item.py

test_delete_menu_item_ingredient

- Receives response through "/menu_item_ingredient/{test_data['mii_id']}"
- Checks the response status code is valid (200 or 204)
- Checks the item was properly deleted by expecting 404 or 422 when attempting to retrieve it again

test_read_menu_item_by_category

- Receives response through "/menu_item/by_category/Appetizer" and "/menu_item/by_category/Dessert"
- Checks the response status is 200
- Checks the "Appetizer" category includes "Caesar Salad"
- Checks the "Dessert" category returns an empty list

test_read_menu_item_by_rating

- Receives response through "/menu_item/by_rating/4"
- Checks the response status is 200

test_search_menu_items

- Receives response through "/menu_item/search?q=salad" and "/menu_item/search?q=notarealitem"
- Checks the response status is 200
- Checks that items returned for "salad" contain "salad" in the name
- Checks that a search for a nonexistent item returns an empty list

test_delete_menu_item

- Receives response through "/menu_item/{test_data['menu_item_id']}"
- Checks the response status code is valid (200 or 204)
- Checks the menu item was deleted by expecting 404 or 422 on subsequent get request

test_popularity_insights

- Receives response through "/menu_item/popularity_insights"
- Checks the response status is 200
- Checks that the response is a list
- Checks each item (if any exist) includes "menu_item_id", "name", "order_count", and "complaint_count" fields

test_order.py

test_read_order_date_range

- Receives response through "/order?start_date=2025-01-01&end_date=2025-12-31"
- Checks that the response status is 200

test_read_order_by_tracking

- Receives response through "/order/track/{test_data['tracking_number']}"
- Checks that the response status is 200
- Checks that the tracking_number field is an integer

test_delete_order_item

- Receives response through "/order_item/{test_data['order_item_id']}"
- Checks the response status is 200 or 204
- Checks the item was deleted by expecting 404 or 422 on retrieval

test_delete_order

- Receives response through "/order/{test_data['order_id']}"
- Checks the response status is 200 or 204
- Checks the order was deleted by expecting 404 or 422 on retrieval

test_place_order

- Sends request to "/order"
- Creates a new order with valid details
- Checks that the response status is 200

test_track_order_status

- Receives order through "/order/{test_data['order_id']}"
- Checks that status exists and is one of the expected values
- Receives response through "/order/track/{test_data['tracking_number']}"
- Checks that tracking information matches the original order status

test_guest_checkout

- Sends request to "/order" without customer_id
- Includes guest details like name, email, and phone

- Checks that the response is 200
- Checks guest name is correct and customer_id is None

test_guest_checkout_with_promotion

- Sends request to "/order" with guest details and a promotion ID
- Checks that the response is 200
- Checks guest name and correct promotion ID

test_registered_customer_checkout_with_promotion

- Sends request to "/order" with a registered customer and promotion ID
- Checks that the response is 200
- Checks that both customer_id and promotion_id match the test data

test_order_with_invalid_promotion

- Sends request to "/order" with a non-existent promotion_id
- Checks that the response status is 400
- Checks that the error message includes "Invalid or expired promotion"

test_order_with_multiple_payments

- Sends request to "/order" with two payment entries
- Checks that the response is 200
- Checks that both payments are present and types are "credit_card" and "cash"

test_payment.py

test_delete_payment

- Receives response through "/payment/{test_data['payment_id']}"
- Checks the response status is 200 or 204
  Checks that the payment was deleted by expecting 404 or 422 on retrieval

test_make_payment

- Sends request to "/payment/" with a new payment payload
  Includes details like order_id, status, type, total, and transaction_id
- Checks that the response status is 200 or 201

test_update_payment

- Sends request to "/payment/{test_data['payment_id']}"
  Updates the payment status to "refunded"
- Checks that the response status is 200

test_promotion.py

test_delete_promotion

- Receives response through "/promotion/{test_data['promotion_id']}"
- Checks that the response status is 200 or 204
  Checks that the promotion was deleted by expecting 404 or 422 on retrieval

test_create_promotion

- Sends request to "/promotion/" with a new promotion payload
- Includes a unique code, discount_amount, and expiration_date
- Checks that the response status is 200 or 201
- Checks that the returned data matches the payload
- Verifies the presence of an "id" field in the response

test_revenue.py

test_get_revenue

- Receives response through "/payment/revenue/?start_date=2025-01-01&end_date=2025-12-31"
  Checks that the response status is 200
  Checks that the response includes "total_revenue", "total_orders", and "total_customers" fields
- Checks that "total_customers" is a number within the expected range (0 to 9999)

test_review.py

test_update_review

- Receives response through "/review/{test_data['review_id']}" (before update)
- Sends update request to "/review/{test_data['review_id']}" with a new rating and comment
  Checks that the response status is 200
- Checks that the comment has been changed from the previous value

test_read_review_by_menu_item

- Receives response through "/review/menu/{test_data['menu_item_id']}"
- Checks that the response status is 200

test_delete_review

- Receives response through "/review/{test_data['review_id']}"
- Checks the response status is 200 or 204
- Checks that the review was deleted by expecting 404 or 422 on retrieval

test_post_review

- Sends request to "/review/" with a new review payload
- Checks that the response status is 200 or 201
  Receives response through "/review/menu/{test_data['menu_item_id']}"
- Checks that the new review appears in the list of reviews for the menu item

test_feedback_analysis

- Receives response through "/review/analysis"
- Checks that the response status is 200
- Checks that the response includes "average_rating", "total_reviews", and "ratings_breakdown"
- Checks that "ratings_breakdown" is a dictionary

Development Environment

Prerequisites:

- Python 3.8 or higher
- Git
- Code editor
- Terminal/Command Line access

Required Dependencies:

```
fastapi          # Web framework for building APIs

uvicorn          # ASGI server for running FastAPI

sqlalchemy       # ORM for database operations

pymysql          # MySQL database driver

pytest           # Testing framework

pytest-mock      # Mocking for tests

httpx            # HTTP client for testing

cryptography     # For security features
```

Project Structure

```
├── api/
│   ├── __init__.py
│   ├── main.py               # Application entry point
│   ├── seed.py               # Database seeding
│   ├── controllers/          # Business logic
│   │   ├── customer.py
│   │   ├── order.py
│   │   ├── payment.py
│   │   └── ...
│   ├── dependencies/         # Configuration and database
│   │   ├── config.py
│   │   └── database.py
│   ├── models/               # SQLAlchemy models
│   │   ├── customer.py
│   │   ├── order.py
│   │   ├── payment.py
│   │   └── ...
│   ├── routers/              # API endpoints
│   │   ├── customer.py
│   │   ├── order.py
│   │   ├── payment.py
│   │   └── ...
│   ├── schemas/              # Pydantic schemas
│   │   ├── customer.py
│   │   ├── order.py
│   │   ├── payment.py
│   │   └── ...
│   └── tests/                # Test files
├── requirements.txt
└── README.md
```

Step 1: Download the API

First, download the restaurant API to your computer:

```
git clone https://github.com/B1naryB0t/SWE-Final-Project-Snack-Overflow.git
cd SWE-Final-Project-Snack-Overflow
```

Don't have Git? You can download the files directly from GitHub by clicking the green "Code" button and selecting "Download ZIP".

Step 2: Set Up Your Environment

Create a safe, isolated environment for the API:

On Windows:

```
python -m venv .venv
.venv\Scripts\activate
```

On Mac/Linux:

```
python -m venv .venv
source .venv/bin/activate
```

Step 3: Install Required Components

Install all the necessary components:

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

Step 4: Choose Your Database

The API works with two database options:

Option A: SQLite (Recommended for Beginners)
This is an option! SQLite is perfect for testing and small restaurants. No additional setup required.

Option B: MySQL (For Larger Operations)
If you need a more robust database, edit the file api/dependencies/config.py:

Change this line:
DATABASE_URL = "mysql+pymysql://username:password@localhost/restaurant_db"
To this (replace with your MySQL details):
DATABASE_URL = "sqlite:///app.db"

The database is automatically created when you first run the application. Tables are created using SQLAlchemy models.

Step 5: Start Your Restaurant API

Launch the API server:

uvicorn api.main:app --reload

Success! You should see a message like:
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)

Step 6: Explore Your API

Open your web browser and visit:

• API Documentation: http://127.0.0.1:8000/docs

Running Tests

This project has tests built in using pytest. Run pytest in the terminal to ensure your code is in a good shape.

Troubleshooting

1. Import Errors: Make sure you're running commands from the project root directory
2. Database Errors: Delete existing .db files and restart the application
3. Port Already in Use: Change the port in config.py or kill the process using port 8000
4. Virtual Environment: Always activate your virtual environment before running commands

Complete Set-Up From Scratch

```
git clone <repository-url>
cd SWE-Final-Project-Snack-Overflow
python -m venv venv
source .venv/bin/activate  # On macOS/Linux
pip install -r requirements.txt
uvicorn api.main:app —reload
```

Code Examples

Database Design: Logic For Easy Switch Between SQLite and MySQL

database.py

```
from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker, declarative_base

from .config import DATABASE_URL

# Automatic MySQL Database Provisioning
if DATABASE_URL.startswith("mysql"):
    import re

    match = re.match(r"mysql\+pymysql://([^:]+):([^@]+)@([^:/]+):?(\d+)?/([^?]+)",
DATABASE_URL)
    if match:
        user, password, host, port, db_name = match.groups()
        port = port or "3306"
        root_url = f"mysql+pymysql://{user}:{password}@{host}:{port}/"
        engine_tmp = create_engine(root_url)
        with engine_tmp.connect() as conn:
            conn.execute(text(f"CREATE DATABASE IF NOT EXISTS {db_name}"))
        engine_tmp.dispose()

# Database-Specific Engine Configuration
if DATABASE_URL.startswith("sqlite"):
    engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
else:
    engine = create_engine(DATABASE_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Configuration Management

config.py

```
import os

class Config:
   app_host = "localhost"
   app_port = 8000

APP_ENV = os.getenv("APP_ENV", "dev")

if APP_ENV == "test":
   DATABASE_URL = "mysql+pymysql://username:password@localhost/test_db"
else:
   DATABASE_URL = "mysql+pymysql://username:password@localhost/app_db"
```

The system implements automatic database detection and configuration through environment-based URL parsing. When a MySQL connection string is detected, the code uses regex to extract connection parameters, establishes a temporary connection to the MySQL server, and automatically creates the target database if it doesn't exist before disposing the temporary connection. For SQLite databases, the engine is configured with check_same_thread=False to ensure compatibility with FastAPI's asynchronous operations, while other database types use standard SQLAlchemy engine creation. This design enables seamless switching between SQLite for development/testing and MySQL for production environments without requiring any changes to application models or business logic - simply by modifying the DATABASE_URL configuration variable.

Order Creation - Complex Business Logic

```
def create(db: Session, order: OrderCreate):
    # 1. Validate customer/guest info
    if order.customer_id:
        customer = db.query(Customer).get(order.customer_id)
        if not customer:
            raise HTTPException(status_code=400, detail="Customer not found")
    else:
        if not (order.guest_name and order.guest_email):
            raise HTTPException(status_code=400, detail="Guest info required")

    # 2. Check inventory - calculate required ingredients
    menu_items = db.query(MenuItem).filter(MenuItem.id.in_(order.menu_item_ids)).all()
    required_ingredients = {}
    for item in menu_items:
        for mii in item.menu_item_ingredients:
            ingr = mii.ingredient
            required_ingredients[ingr.id] = required_ingredients.get(ingr.id, 0) +
mii.quantity

    # 3. Validate and update stock
    for ingr_id, required_qty in required_ingredients.items():
        ingr = db.query(Ingredient).get(ingr_id)
        if ingr.quantity < required_qty:
            raise HTTPException(status_code=400, detail=f"Not enough '{ingr.name}' in
stock")
        ingr.quantity -= required_qty

    # 4. Calculate total with promotions
    total_price = sum(item.price for item in menu_items)
    if order.promotion_id:
        promo = db.query(Promotion).filter(
            Promotion.id == order.promotion_id,
            Promotion.expiration_date >= datetime.utcnow()
        ).first()
        if promo:
            total_price = max(total_price - promo.discount_amount, 0)

    # 5. Create order with tracking number
    db_order = Order(
        date=order.date, status=order.status, total=total_price,
        tracking_number=random.randint(100000, 999999),
        customer_id=order.customer_id, guest_name=order.guest_name
```

```
    )
    db.add(db_order)
    db.commit()

    # 6. Create linked payments and order items
    for payment_data in order.payments:
        db_payment = Payment(
            status=payment_data.status, type=payment_data.type,
            total=payment_data.total, order_id=db_order.id
        )
        db.add(db_payment)

    for item in menu_items:
        db.add(OrderItem(order_id=db_order.id, menu_item_id=item.id, quantity=1))

    db.commit()
    return db_order
```

What it does: Validates customers/guests, checks ingredient inventory, deducts stock, applies promotions, creates order with tracking number, and links payments/items. Demonstrates real restaurant business logic with inventory management and complex validation.

Execution of Program and Database Seeding

```python
import uvicorn
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from .dependencies.config import Config
from .dependencies.database import Base, engine
from .models import model_loader
from .routers import index as indexRoute
from .seed import seed_if_needed

# 1. Load all SQLAlchemy models for table creation
model_loader.index()

# 2. Create all database tables based on models
Base.metadata.create_all(bind=engine)

# 3. Populate database with initial data if empty
seed_if_needed()

# 4. Initialize FastAPI app with CORS
app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# 5. Register all API routes
indexRoute.load_routes(app)

if __name__ == "__main__":
    uvicorn.run(app, host=Config.app_host, port=Config.app_port)
```

What the seeding does:

Model Loading - Imports all SQLAlchemy models so they're registered
Table Creation - Automatically creates database tables if they don't exist
Smart Seeding - seed_if_needed() checks if database is empty and populates with sample customers, menu items, ingredients, and test data
One-Time Setup - Only seeds on first run, subsequent starts skip seeding
Result: Fresh deployments get a working database with sample data immediately - no manual setup required.