Technical Documentation

Online Restaurant Ordering System (OROS) API

Group 8
May 6, 2025

Table of Contents

Architecture Overview (with Code Examples)

Customer

The Customer class defines the structure of the customers table in the relational database and establishes a direct mapping between Python objects and database records. This SQLAlchemy model is paired with corresponding Pydantic models (*CustomerBase*, *CustomerCreate*, and *Customer*) to handle data validation and serialization in the API layer.

Class Attributes

Column	Type	Description
_id_	Integer	An integer column that serves as the primary key. It is indexed for efficient querying.
name	String(50)	A String column with a maximum length of 50 characters. This field is required (*nullable=False*)
email	String(500)	A unique and required String field (up to 500 characters) to store customer email addresses. The *unique=True* constraint ensures no two records share the same email.
phone	String(25)	A required String field (up to 25 characters) to store the customer's phone number.
address	String(500)	An optional String field (up to 500 characters) to store the customer's physical address.

```
class Customer(Base):
   __tablename__ = 'customers'

   id = Column(Integer, primary_key=True, index=True)
   name = Column(String(50), nullable=False)
   email = Column(String(500), unique=True, nullable=False)
   phone = Column(String(25), nullable=False)
   address = Column(String(500), nullable=True)
```

Models

- *CustomerBase(BaseModel)* This is the base model from which other customer models inherit. It defines the shared fields used across both creation and retrieval of customer records.

```
○  class CustomerBase(BaseModel):
○     name: str
○     email: EmailStr
○     phone: str
○     address: str | None = None
```

- *CustomerCreate(CustomerBase)* This model handles incoming data for customer creation while maintaining validation through inherited types.

```
class CustomerCreate(CustomerBase):
    pass
```

- *Customer(CustomerBase)* This model is used to structure outgoing data (e.g., in GET responses), typically when returning full customer objects, including the unique id.

```
class Customer(CustomerBase):
    id: int

    class Config:
        orm_mode = True
```

Menu Items

This class represents an item on the restaurant's menu.

Class Attributes

Column	Type	Description
id	Integer	An integer column that serves as the primary key. It is indexed for efficient querying.
dish_name	String(100)	A required string field (up to 100 characters) for the name of the dish
ingredients	String(500)	An optional description of the ingredients.
price	Decimal(5, 2)	A required decimal field (with a default of 0.00) with two decimal places.
calories	Integer	An optional number of calories.
drink_category	String(100)	An optional tag to categorize drinks (e.g., "coffee", "tea").

```
class MenuItem(Base):
    __tablename__ = "menu_items"

    id = Column(Integer, primary_key=True, index=True, autoincrement=True)
    dish_name = Column(String(100), nullable=False)
    ingredients = Column(String(500), nullable=True)
    price = Column(DECIMAL(5, 2), nullable=False, default=0.00)
    calories = Column(Integer, nullable=True)
    drink_category = Column(String(100), nullable=True)  # e.g., 'coffee', 'tea', or None
```

Models

- *MenuItemBase(BaseModel)* The base model that defines the shared fields for menu items across all operations (creation, update, and retrieval). It serves as the foundation for the other Pydantic models.

  ```
  class MenuItemBase(BaseModel):
      dish_name: str
      ingredients: Optional[str] = None
      price: float
      calories: Optional[int] = None
      drink_category: Optional[str] = None
  ```

- *MenuItemCreate(MenuItemBase)* Used for creating new menu items. It inherits all fields from *MenuItemBase*, meaning all required and optional fields are the same. This model is typically used in POST requests where the client provides new item data to be stored in the database.

  ```
  class MenuItemCreate(MenuItemBase):
      pass
  ```

- *MenuItemUpdate(BaseModel)* Used for updating existing menu items. All fields are optional, which allows partial updates—only the fields that need to be modified must be included in the request.

  ```
  class MenuItemUpdate(BaseModel):
      dish_name: Optional[str] = None
      ingredients: Optional[str] = None
      price: Optional[float] = None
      calories: Optional[int] = None
      drink_category: Optional[str] = None
  ```

- *MenuItem(MenuItemBase)* This model is used for returning data to the client (e.g., in GET responses). It extends *MenuItemBase* and includes an additional id field, which uniquely identifies each menu item in the database.

  ```
  class MenuItem(MenuItemBase):
      id: int

      class ConfigDict:
          from_attributes = True
  ```

Order Details

The *OrderDetail* class defines the structure of the order_details table in the relational database and establishes a direct mapping between Python objects and database records. This SQLAlchemy model is paired with corresponding Pydantic models (*OrderDetailBase, OrderDetailCreate*, and *OrderDetail*) to handle data validation and serialization in the API layer.

Class Attributes

Column	Type	Description
id	Integer	An integer column that serves as the primary key. It is

		indexed and auto-incremented.
order_id	Integer (ForeignKey)	A foreign key referencing the *orders* table, linking each detail to a specific order.
amount	Integer	A required field that indicates the quantity or number of items in the order. Indexed for performance.
special_requests	String(1000)	An optional field to store any customer-specific notes or requests (up to 1000 characters).
is_delivery	Boolean	A required boolean field indicating whether the order is for delivery (*True*) or not (*False*).

The class also includes a relationship:

orders A relationship linking the *OrderDetail* to the parent *Order* model.

```
class OrderDetail(Base):
   __tablename__ = "order_details"

   id = Column(Integer, primary_key=True, index=True, autoincrement=True)
   order_id = Column(Integer, ForeignKey("orders.id"))
   amount = Column(Integer, index=True, nullable=False)
   special_requests = Column(String(1000), nullable=True)
   is_delivery = Column(Boolean, nullable=False)

   orders = relationship("Order", back_populates="order_details")
```

Models

- *OrderDetailBase(BaseModel)* This is the base model from which other order detail models inherit. It defines the core fields shared across creation and retrieval of order details.

```
class OrderDetailBase(BaseModel):
   amount: int
   special_requests: str
   is_delivery: bool
```

- *OrderDetailCreate(OrderDetailBase)* This model handles incoming data for creating a new order detail. It inherits the shared fields from *OrderDetailBase* and adds the required *order_id* foreign key reference.

```
class OrderDetailCreate(OrderDetailBase):
   order_id: int
```

- *OrderDetailUpdate(BaseModel)* This model is used for updating existing order details. All fields are optional, enabling partial updates.

```
class OrderDetailUpdate(BaseModel):
   order_id: Optional[int] = None
   amount: Optional[int] = None
   special_requests: Optional[str] = None
```

```
    is_delivery: Optional[bool] = None
```

- *OrderDetail(OrderDetailBase)* This model is used to structure outgoing data (e.g., in GET responses). It includes the unique id and the *order_id* foreign key.

```
class OrderDetail(OrderDetailBase):
    id: int
    order_id: int

    class ConfigDict:
        from_attributes = True
```

Orders

The *Order* class defines the structure of the orders table in the relational database and establishes a direct mapping between Python objects and database records using SQLAlchemy. It is paired with multiple Pydantic models (*OrderBase*, *OrderCreate*, *OrderUpdate*, and *Order*) for API-level data validation and serialization.

Class Attributes

Column	Type	Description
*_id_*	Integer	Primary key for the table. Indexed and auto-incremented.
customer_name	String(100)	Required field to store the customer’s full name.
order_date	DATETIME	The timestamp of when the order was created. Defaults to *datetime.utcnow*.
order_number	String(50)	Optional string for a unique or trackable order identifier.
order_status	String(50)	Indicates the order status (e.g., "Processing", "Completed"). Default is "Processing".
total_price	DECIMAL(10,2)	The total cost of the order (defaults to 0.00).

The class also includes two relationships:

- *order_details* A one-to-many relationship linking this order to multiple *OrderDetail* records. Enables the app to fetch all related items for a single order.
- *payment_information* Links this order to the *PaymentInformation* record that stores transaction-related data.

```
class Order(Base):
   __tablename__ = "orders"

   id = Column(Integer, primary_key=True, index=True, autoincrement=True)
   customer_name = Column(String(100), nullable=False)

   order_date = Column(DATETIME, nullable=False, default=datetime.utcnow)
   order_number = Column(String(50), nullable=True)
   order_status = Column(String(50), nullable=False, default="Processing")
   total_price = Column(DECIMAL(10, 2), nullable=False, default=0.00)


   order_details = relationship("OrderDetail", back_populates="orders")
                payment_information     =     relationship("PaymentInformation",
back_populates="orders")
```

Models

- *OrderBase(BaseModel)* The base Pydantic model shared by all other order-related models. It contains the core fields for order data.

  ```
  class OrderBase(BaseModel):
     customer_name: str
     order_number: Optional[str] = None
     order_status: Optional[str] = "Processing"
     total_price: Optional[float] = 0.00
  ```

- *OrderCreate(OrderBase)* Used for validating incoming data when creating a new order. Inherits from *OrderBase*.

  ```
  class OrderCreate(OrderBase):
     pass
  ```

- *OrderUpdate(BaseModel)* This model supports partial updates to existing orders. All fields are optional to allow PATCH-like operations.

  ```
  class OrderUpdate(BaseModel):
     customer_name: Optional[str] = None
     order_number: Optional[str] = None
     order_status: Optional[str] = None
     total_price: Optional[float] = None
  ```

- *Order(OrderBase)* Used for returning complete order data in GET responses. Includes the order's unique ID, creation date, and associated order details.

  ```
  class Order(OrderBase):
    id: int
    order_date: Optional[datetime] = None
    order_details: list[OrderDetail]

    class ConfigDict:
        from_attributes = True
  ```

Status Models

These are specialized Pydantic models used for updating only the order status. These models make it easier to perform standardized status updates through PATCH endpoints or workflow automation.

```
class OrderStatusReceived(BaseModel):
   order_status: Optional[str] = "Received"


class OrderStatusInProgress(BaseModel):
```

```
   order_status: Optional[str] = "In Progress"


class OrderStatusCompleted(BaseModel):
   order_status: Optional[str] = "Completed"
```

Resource Management

The Resource Management feature is responsible for tracking and organizing the quantities and units of resources used within the system. It serves as a link between raw materials (resources) and their respective quantities in inventory or operational use. This component is essential for recipe preparation, inventory tracking, and production planning.

Class Attributes

Column	Type	Description
*_id_*	Integer	Primary key. Uniquely identifies each resource management record. Indexed.
resource_id	Integer (ForeignKey)	Foreign key linking to the *resources* table.
resource_amount	Integer	The quantity of the resource being tracked.
unit	String(50)	The unit of measurement for the resource (e.g., kg, liters, pieces).

The class also includes a relationship:

- *resources* Establishes a bidirectional relationship with the *Resource* model through *resource_management*, enabling access to resource-specific data.

```
class ResourceManagement(Base):
   __tablename__ = "resource_management"

   id = Column(Integer, primary_key=True, index=True, autoincrement=True)
   resource_id = Column(Integer, ForeignKey("resources.id"))
   resource_amount = Column(Integer)
   unit = Column(String(50), nullable=False)

   resources = relationship("Resource", back_populates="resource_management")
    # resources already has a relationship to recipes; likely can access the
recipes table through that
```

Models

- *ResourceManagementBase* Defines shared fields across create and response models.

```
○  class ResourceManagementBase(BaseModel):
○     resource_amount = int
○     unit = str
```

- *ResourceManagementCreate* Used when creating a new resource management record. Inherits from *ResourceManagementBase*.

```
class ResourceManagementCreate(ResourceManagementBase):
    pass
```

- `ResourceManagementUpdate` Used to update an existing resource management record. All fields are optional to support partial updates.

```
class ResourceManagementUpdate(BaseModel):
    resource_amount: Optional[int] = None
    unit: Optional[str] = None
```

- `ResourceManagement` Used in responses. Includes the unique ID of the record.

```
class ResourceManagement(ResourceManagementBase):
    id: int

    class ConfigDict:
        from_attributes = True
```

Resources

The Resources component of the system manages the raw materials or inventory items used across recipes and production processes. Each resource is uniquely identifiable and tracks both its name and current quantity available. This model serves as a central point for managing stock levels and linking resources to their use in recipes and operational tracking.

Class Attributes

Column	Type	Description
id	Integer	Primary key. Uniquely identifies each resource. Indexed and auto-incremented.
item	String(100)	Name or label of the resource. Must be unique and cannot be null.
amount	Integer	Current quantity of the resource. Cannot be null. Defaults to 0.

The class also includes two relationships:

- *recipes* A one-to-many relationship linking this order to multiple OrderDetail records. Enables the app to fetch all related items for a single order.
- *resource _management* Links each resource to the *ResourceManagement* table for quantity and unit tracking in various contexts.

```
class Resource(Base):
   __tablename__ = "resources"

   id = Column(Integer, primary_key=True, index=True, autoincrement=True)
   item = Column(String(100), unique=True, nullable=False)
   amount = Column(Integer, index=True, nullable=False, server_default='0.0')

   recipes = relationship("Recipe", back_populates="resources")
                resource_management      =     relationship("ResourceManagement",
back_populates="resources")
```

Models

- *ResourceBase* The shared base model for both creation and retrieval.

```
class ResourceBase(BaseModel):
    item: str
    amount: int
```

- *ResourceCreate* Used for creating new resource records. Inherits all fields from *ResourceBase*.

```
class ResourceCreate(ResourceBase):
    pass
```

- *ResourceUpdate* Supports partial updates. All fields are optional.

```
class ResourceUpdate(BaseModel):
    item: Optional[str] = None
    amount: Optional[int] = None
```

- *Resource* Used to return complete resource data in API responses, including the unique *id*.

```
class Resource(ResourceBase):
    id: int

    class ConfigDict:
        from_attributes = True
```

Payment Information

The *PaymentInformation* class defines the structure of the *payment_information* table in the relational database and establishes a direct mapping between Python objects and corresponding database records. This SQLAlchemy model is linked with Pydantic models (*PaymentInformationBase*, *PaymentInformationCreate*, and *PaymentInformation*) to handle data validation and serialization for API interactions.

Class Attributes

Column	**Type**	**Description**
*_id_*	Integer	The primary key of the table. It is auto-incremented and indexed for efficiency.
order_id	Integer (ForeignKey)	A foreign key linking the payment record to a specific order in the *orders* table.
card_information	Integer	Stores the credit/debit card number used for payment.
transaction_status	String(50)	A required field indicating the result of the transaction (e.g., "Success", "Failed").
transaction_type	String(50)	A required field specifying the type of transaction (e.g., "Credit", "Refund").

The class also includes a relationship:

orders Establishes a back-reference to the *Order* class, allowing bi-directional navigation between an order and its payment information.

```
class PaymentInformation(Base):
   __tablename__ = "payment_information"

   id = Column(Integer, primary_key=True, index=True, autoincrement=True)
   order_id = Column(Integer, ForeignKey("orders.id"))
   card_information = Column(Integer)
   transaction_status = Column(String(50), nullable=False)
   transaction_type = Column(String(50), nullable=False)

   orders = relationship("Order", back_populates="payment_information")
```

Models

- *PaymentInformationBase(BaseModel)* Defines the core fields used across both creation and retrieval of payment records. It includes validation rules for required fields and shared structure.

```
class PaymentInformationBase(BaseModel):
   card_information = int
   transaction_status = str
   transaction_type = str
```

- *PaymentInformationCreate(PaymentInformationBase)* Used for creating new payment records. It inherits all required fields from the base model.

```
class PaymentInformationCreate(PaymentInformationBase):
   pass
```

- *PaymentInformationUpdate(BaseModel)* Used for updating existing payment records. All fields are marked optional to allow partial updates.

```
class PaymentInformationUpdate(BaseModel):
   card_information: Optional[int] = None
   transaction_status: Optional[str] = None
   transaction_type: Optional[str] = None
```

- *PaymentInformation(PaymentInformationBase)* Used to represent outgoing data (e.g., in GET responses).

```
class PaymentInformation(PaymentInformationBase):
   id: int


   class ConfigDict:
       from_attributes = True
```

Reviews

The Reviews module handles customer-generated feedback on their experiences. Each review is tied to a customer and includes a text comment and a numerical score, providing qualitative and quantitative insight into user satisfaction.

Class Attributes

Column	Type	Description
*_id_*	Integer	Primary key. Uniquely identifies each review. Indexed for performance.
customer_id	Integer	Foreign key referencing the *customers* table. Indicates the author of the review.
reviews_text	String	The actual written content of the review. Cannot be null.
score	Integer	Numeric score provided by the customer (e.g., 1–5). Cannot be null.

```
class Review(Base):
   __tablename__ = 'reviews'

   id = Column(Integer, primary_key=True, index=True)
   customer_id = Column(Integer, ForeignKey('customers.id'), nullable=False)
   review_text = Column(String, nullable=False)
   score = Column(Integer, nullable=False)
```

Models

- *ReviewBase* Defines the core attributes shared by all review-related operations.

  ```
  class ReviewBase(BaseModel):
     customer_id: int
     review_text: str
     score: int
  ```

- *ReviewCreate* Inherits from *ReviewBase*. Used specifically when creating new review entries.

  ```
  class ReviewCreate(ReviewBase):
     pass
  ```

- *ReviewOut* Extends *ReviewBase* to include the review's unique *id*. Used when returning review data in responses.

  ```
  class ReviewOut(ReviewBase):
     id: int

     class Config:
         orm_mode = True
  ```

Recipes

The Recipes module defines the relationships between drinks and the resources required to make them. Each recipe links a drink to one or more resources (ingredients) and specifies the quantity of each resource needed.

Class Attributes

Column	Type	Description
id	Integer	Primary key. Auto-incremented and uniquely identifies each recipe. Indexed.
drink_id	Integer	Foreign key referencing the *drinks* table. Identifies the drink this recipe is part of.
resource_id	Integer	Foreign key referencing the *resources* table. Represents the ingredient used.
amount	Integer	Required field specifying the quantity of the resource needed. Indexed for performance.

The class also includes two relationships:

- *drinks* Links each recipe to its associated drink
- *resources* Links each recipe to the required resource (ingredient).

```
class Recipe(Base):
   __tablename__ = "recipes"

   id = Column(Integer, primary_key=True, index=True, autoincrement=True)
   drink_id = Column(Integer, ForeignKey("drinks.id"))
   resource_id = Column(Integer, ForeignKey("resources.id"))
   amount = Column(Integer, index=True, nullable=False, server_default='0.0')

   drinks = relationship("Drink", back_populates="recipes")
   resources = relationship("Resource", back_populates="recipes")
```

Models

- *RecipeBase* Defines shared fields used across creation and retrieval operations.

  ```
  ○ class RecipeBase(BaseModel):
  ○    amount: int
  ```

- *RecipeCreate* Used when creating a new recipe. Requires a reference to a drink and a resource, along with the amount needed.

  ```
  ○ class RecipeCreate(RecipeBase):
  ○    drink_id: int
  ○    resource_id: int
  ```

- *RecipeUpdate* Supports partial updates to recipe records. All fields are optional to allow flexibility in PATCH-like operations.

  ```
  ○ class RecipeUpdate(BaseModel):
  ○    drink_id: Optional[int] = None
  ```

```
    resource_id: Optional[int] = None
    amount: Optional[int] = None
```

- *Recipe* Extends *RecipeBase* and includes additional metadata for responses, such as *id*, *drink*, and *resource* relationships.

```
class Recipe(RecipeBase):
    id: int
    drink: Drink = None
    resource: Resource = None

    class ConfigDict:
        from_attributes = True
```

Promotion Codes

Jhgvhjmgcvhjmgfcfgj

Class Attributes

Column	Type	Description
id	Integer	Primary key. Auto-incremented and uniquely identifies each promo code.
code	String	A unique string identifier for the promotion (e.g., “SPRINGSALE20”).
expiration_date	DateTime	The date and time after which the promotion code becomes invalid.
is_active	Boolean	Indicates whether the promotion code is currently active. Default is True.

```
class PromotionCode(Base):
    __tablename__ = "promotion_codes"

    id = Column(Integer, primary_key=True, index=True, autoincrement=True)
    code = Column(String(50), unique=True, nullable=False)
    expiration_date = Column(DateTime, nullable=False)
    is_active = Column(Boolean, default=True)  # Optional: allows easy deactivation
```

Models

- *PromotionCodeBase* Defines the common fields shared across multiple operations.

```
class PromotionCodeBase(BaseModel):
    code: str
    expiration_date: datetime
    is_active: Optional[bool] = True
```

- *PromotionCodeCreate* Used for creating new promotion codes. Inherits all fields from *PromotionCodeBase*.

```
class PromotionCodeCreate(PromotionCodeBase):
    pass
```

- *PromotionCodeUpdate* Used for updating existing promotion codes. All fields are optional to allow partial updates.

```
class PromotionCodeUpdate(BaseModel):
    code: Optional[str] = None
    expiration_date: Optional[datetime] = None
    is_active: Optional[bool] = None
```

- *PromotionCode* This model is used to format data sent back to the client (e.g., in GET responses). It includes the unique ID of the promotion code.

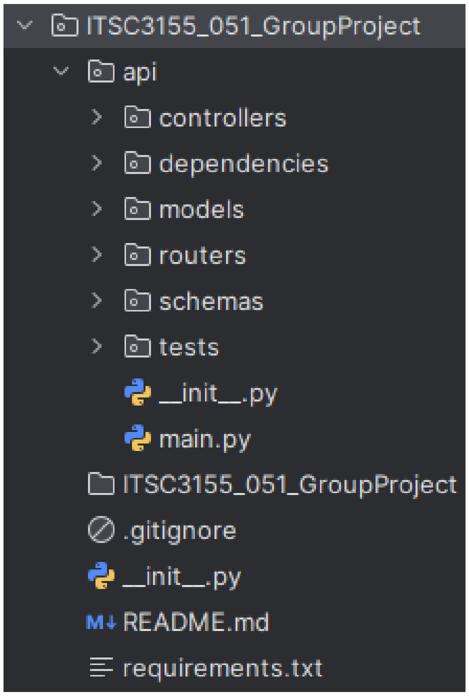
```
class PromotionCode(PromotionCodeBase):
    id: int

    class ConfigDict:
        from_attributes = True
```

Folder Roles

In this OROS API, the routers and controllers folders communicate with each other by separating concerns and allowing clean interaction between the API layer (routers) and the business logic layer (controllers). The schemas and models folders work together to manage data flow between the API and the database. The models folder contains SQLAlchemy models, which define how data is structured and stored in the database through table mappings, data types, and relationships. In contrast, the schemas folder holds Pydantic models used for validating and serializing data sent to and from the API. When a client sends a request, FastAPI uses a schema to validate the input before passing it to the controller, which creates or updates a corresponding SQLAlchemy model and commits it to the database. The result is then converted back into a schema for the API response. This separation of concerns allows for cleaner code organization, better security, and easier maintenance, as it clearly distinguishes between the internal database logic and the external data exchange handled by the API.

Layer	**Folder**	**Responsibility**
API Layer	*routers/*	Defines HTTP endpoints (routes).
Logic	*controllers/*	Handles database logic and business rules.
Models	*models/*	Defines SQLAlchemy DB models.
Schemas	*schemas/*	Defines Pydantic models for validation.

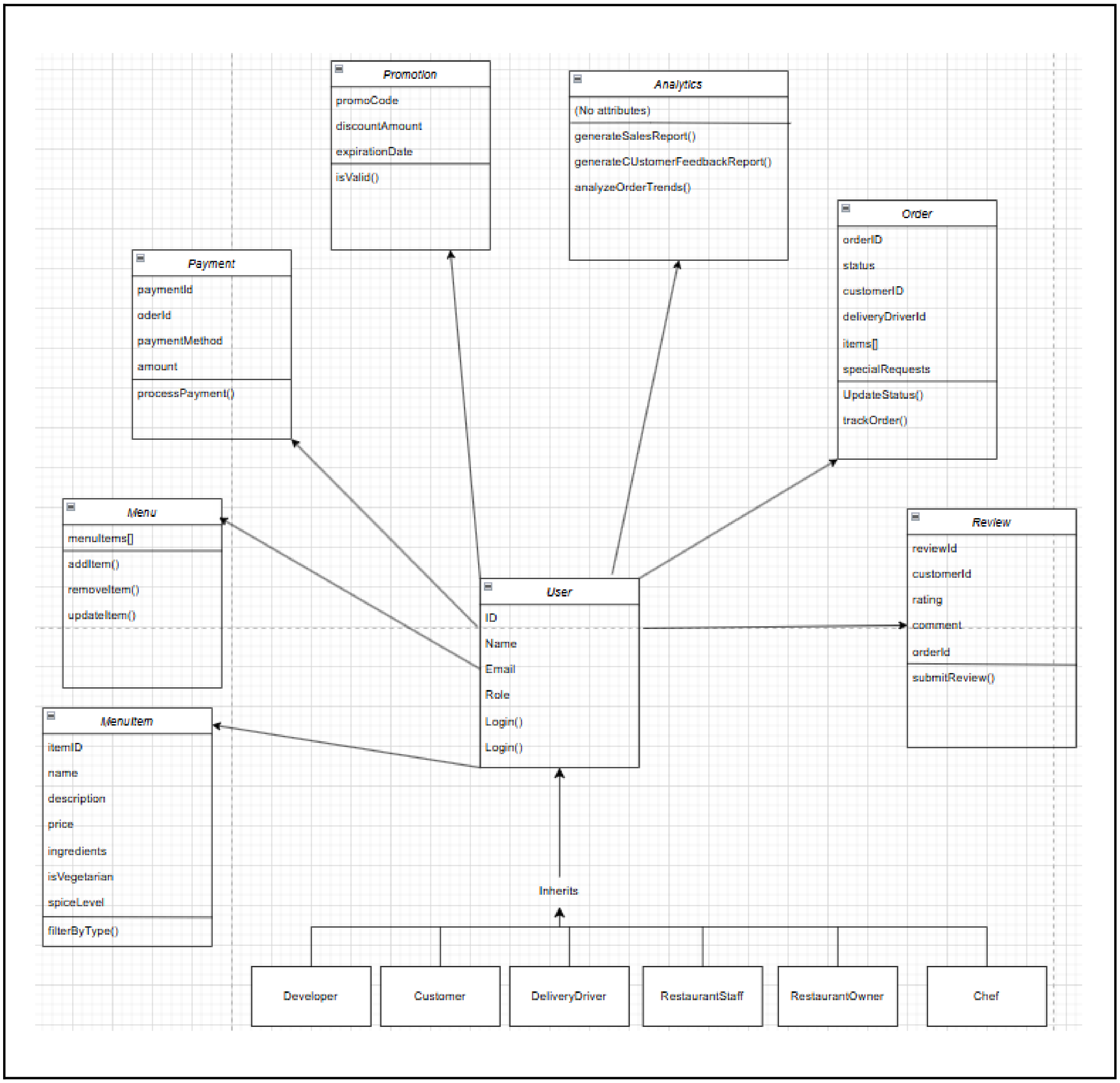


```
ITSC3155_051_GroupProject
  api
    controllers
    dependencies
    models
    routers
    schemas
    tests
    __init__.py
    main.py
  ITSC3155_051_GroupProject
  .gitignore
  __init__.py
  README.md
  requirements.txt
```

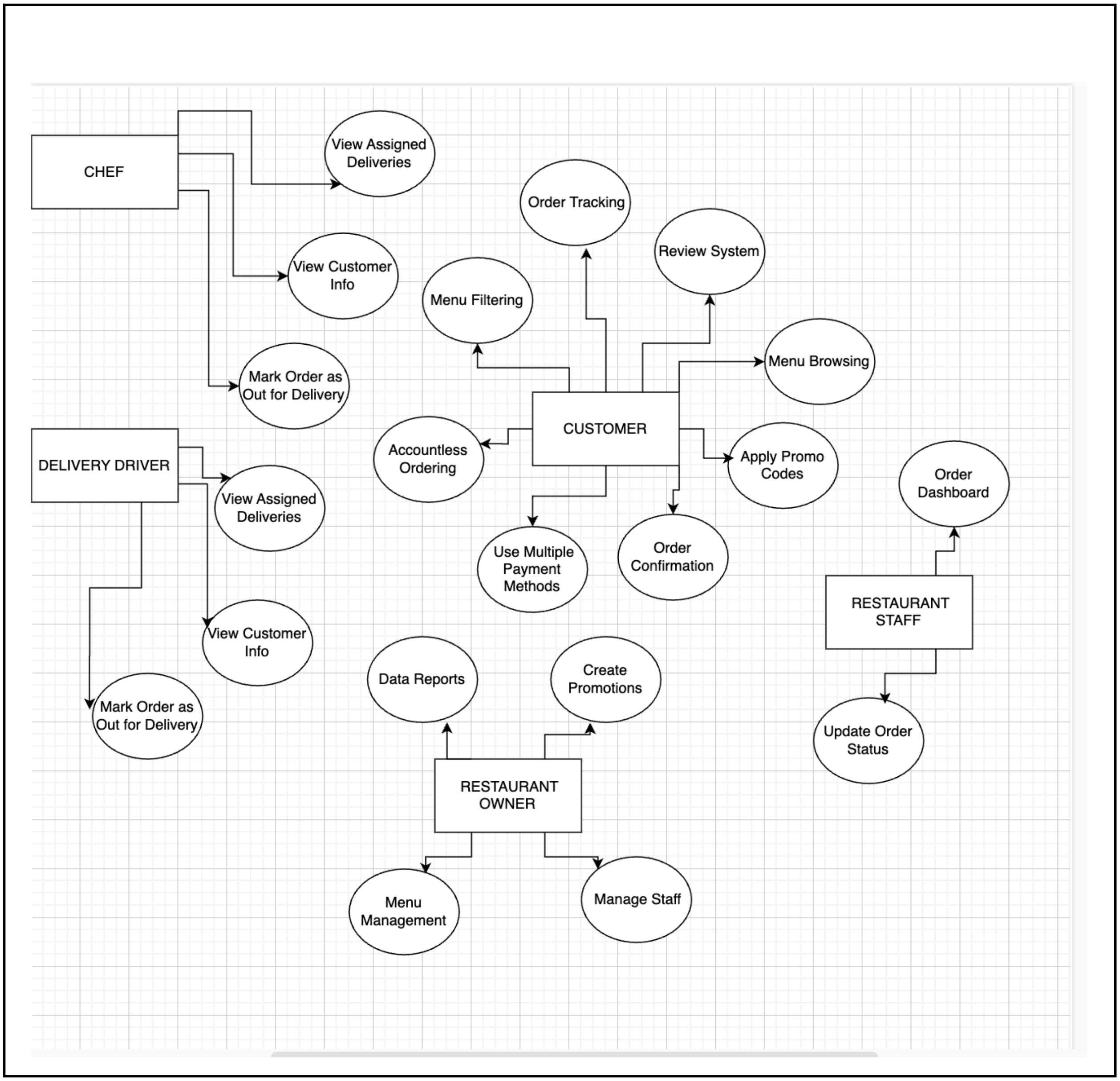
Architecture Overview (Diagrams From Part 2)

Some of the information and code snippets detailed above differ slightly from the diagrams we created during Part 2 of the assignment. These diagrams (Class Diagram, Use Case Diagram, and Component Diagram) can be seen below.

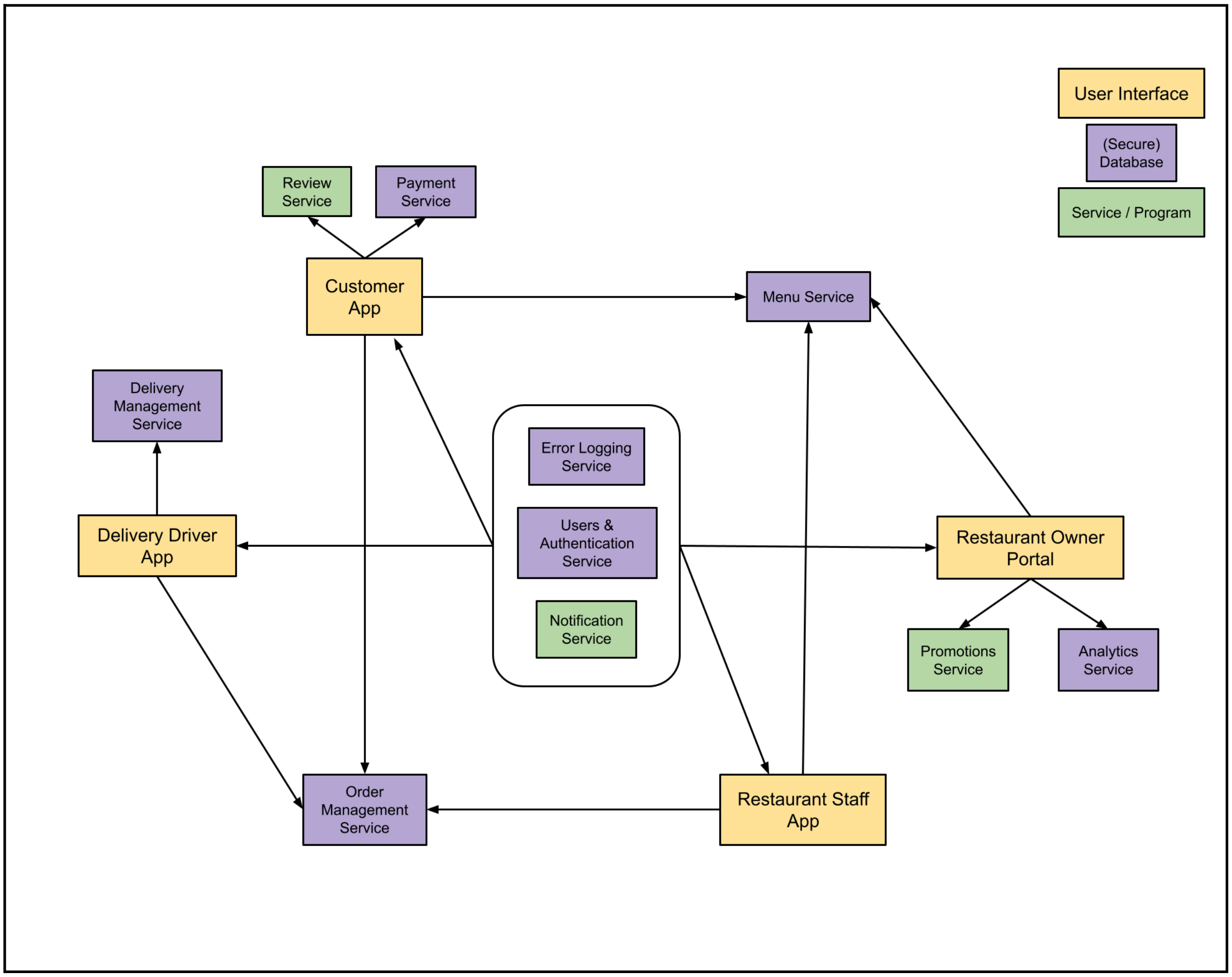
Class diagram



Use Case diagram



Component diagram



Endpoint Documentation (with Code Examples)

api/routers/index.py

While *index.py* itself does not define individual endpoints, it plays a critical role in routing and modular architecture, making sure that all API modules (like *orders*, *order_details*, and *order_mgmt*) are registered with the main FastAPI app.

Included Routers

Router Name	Description	Example Endpoint
orders.router	Handles endpoints related to order creation and listing.	● *POST /orders/* ● *GET /orders/*
order_details.router	Manages detailed items in each order.	● *GET /orderdetails/{id}*
order_mgmt.router	May include endpoints for order status, history, or updates.	● *PUT /orders/{id}/status*

```
def load_routes(app):
   app.include_router(orders.router)
   app.include_router(order_details.router)
   app.include_router(order_mgmt.router)
```

order_details.py

The following endpoints manage order detail records. Each route communicates with the controller layer to perform operations such as creating, reading, updating, and deleting order detail entries in the database. The code snippets pictured below are from two different folders, with the first code snippet from each section belonging to the file api/controllers/order_details.py and the second belonging to the file api/routers/order_details.py.

- **Create Order Detail**

Endpoint	*POST /orderdetails/*
Function	*create(db: Session, request)*
Description	Creates a new order detail entry in the database.
Successful Request Status Code	201 Created
Unsuccessful Request Status Code	400 Bad Request (If a database constraint fails or input is invalid.)

```
def create(db: Session, request):
   new_item = model.OrderDetail(
       order_id=request.order_id,
       amount=request.amount,
       special_requests=request.special_requests,
       is_delivery=request.is_delivery
   )

   try:
       db.add(new_item)
       db.commit()
       db.refresh(new_item)
   except SQLAlchemyError as e:
       error = str(e.__dict__['orig'])
                   raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)

   return new_item
```

```
@router.post("/", response_model=schema.OrderDetail)
def create(request: schema.OrderDetailCreate, db: Session = Depends(get_db)):
    return controller.create(db=db, request=request)
```

- **Get All Order Details**

Endpoint	*GET /orderdetails/*
Function	*read_all(db: Session)*
Description	Retrieves all order detail records in the system.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If a query error occurs.)

```
def read_all(db: Session):
    try:
        result = db.query(model.OrderDetail).all()
    except SQLAlchemyError as e:
        error = str(e.__dict__['orig'])
                    raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
    return result
```

```
@router.get("/", response_model=list[schema.OrderDetail])
def read_all(db: Session = Depends(get_db)):
    return controller.read_all(db)
```

- **Read one Order Detail by ID**

Endpoint	*GET /orderdetails/{id}*
Function	*read_one(db: Session, item_id)*
Description	Retrieves a specific order detail by its ID.
Path Parameters	*id* (integer): The unique ID of the order detail to retrieve.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If a query error occurs.) 404 Not Found (If no record is found for the given ID.)

```
def read_one(db: Session, item_id):
    try:
        item = db.query(model.OrderDetail).filter(model.OrderDetail.id ==
item_id).first()
        if not item:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Id not found!")
    except SQLAlchemyError as e:
        error = str(e.__dict__['orig'])
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
    return item
```

```
@router.get("/{item_id}", response_model=schema.OrderDetail)
def read_one(item_id: int, db: Session = Depends(get_db)):
    return controller.read_one(db, item_id=item_id)
```

- **Update Order Detail**

Endpoint	*PUT /orderdetails/{id}*
Function	*update(db: Session, item_id, request)*
Description	Updates a specific order detail. Supports partial updates by excluding unset fields.
Path Parameters	*id* (integer): The ID of the order detail to update.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If the ID does not exist.) 404 Not Found (If validation fails or a DB error occurs.)

```
def update(db: Session, item_id, request):
    try:
        item = db.query(model.OrderDetail).filter(model.OrderDetail.id ==
item_id)
        if not item.first():
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Id not found!")
        update_data = request.dict(exclude_unset=True)
        item.update(update_data, synchronize_session=False)
        db.commit()
    except SQLAlchemyError as e:
        error = str(e.__dict__['orig'])
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
    return item.first()
```

```
@router.put("/{item_id}", response_model=schema.OrderDetail)
def update(item_id: int, request: schema.OrderDetailUpdate, db: Session =
Depends(get_db)):
    return controller.update(db=db, request=request, item_id=item_id)
```

- **Delete Order Detail**

Endpoint	*DELETE /orderdetails/{id}*
Function	*delete(db: Session, item_id)*
Description	Deletes a specific order detail by its ID.
Path Parameters	*`id`* (integer): The ID of the order detail to delete.
Successful Request Status Code	204 No Content (No body is returned if deletion is successful)
Unsuccessful Request Status Code	400 Bad Request (If the specified ID does not exist.) 404 Not Found (If deletion fails due to a DB error.)

```
def delete(db: Session, item_id):
    try:
        item = db.query(model.OrderDetail).filter(model.OrderDetail.id ==
item_id)
        if not item.first():
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Id not found!")
        item.delete(synchronize_session=False)
        db.commit()
    except SQLAlchemyError as e:
        error = str(e.__dict__['orig'])
```

```
                  raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

```
@router.delete("/{item_id}")
def delete(item_id: int, db: Session = Depends(get_db)):
    return controller.delete(db=db, item_id=item_id)
```

orders.py

The following endpoints manage order detail records. Each route communicates with the controller layer to perform operations such as creating, reading, updating, and deleting order detail entries in the database. The code snippets pictured below are from two different folders, with the first code snippet from each section belonging to the file api/controllers/orders.py and the second belonging to the file api/routers/orders.py.

- **Create Order**

Endpoint	*POST /orders/*
Function	*create(db: Session, request)*
Description	Creates a new order entry in the database.
Successful Request Status Code	201 Created
Unsuccessful Request Status Code	400 Bad Request (If a database constraint fails or input is invalid.)

```
def create(db: Session, request):
    new_item = model.Order(
        customer_name=request.customer_name,
        order_number=request.order_number,
        order_status=request.order_status,
        total_price=request.total_price
    )

    try:
        db.add(new_item)
        db.commit()
        db.refresh(new_item)
    except SQLAlchemyError as e:
        error = str(e.__dict__['orig'])
                  raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)

```

```
    return new_item
```

```
@router.post("/", response_model=schema.Order)
def create(request: schema.OrderCreate, db: Session = Depends(get_db)):
    return controller.create(db=db, request=request)
```

- **Get All Orders**

Endpoint	*GET /orders/*
Function	*read_all(db: Session)*
Description	Retrieves all order records in the system.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If a query error occurs.)

```
def read_all(db: Session):
    try:
        result = db.query(model.Order).all()
    except SQLAlchemyError as e:
        error = str(e.__dict__['orig'])
                   raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
    return result
```

```
@router.get("/", response_model=list[schema.Order])
def read_all(db: Session = Depends(get_db)):
    return controller.read_all(db)
```

- **Read one Order by ID**

Endpoint	*GET /orders/{id}*
Function	*read_one(db: Session, item_id)*
Description	Retrieves a specific order by its ID.
Path Parameters	*id* (integer): The unique ID of the order to retrieve.
Successful Request Status Code	200 OK
Unsuccessful	400 Bad Request (If a query error occurs.)

Request Status Code	404 Not Found (If no record is found for the given ID.)

```
def read_one(db: Session, item_id):
   try:
       item = db.query(model.Order).filter(model.Order.id == item_id).first()
       if not item:
                    raise  HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Id not found!")
   except SQLAlchemyError as e:
       error = str(e.__dict__['orig'])
                  raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
   return item
```

```
@router.get("/{item_id}", response_model=schema.Order)
def read_one(item_id: int, db: Session = Depends(get_db)):
   return controller.read_one(db, item_id=item_id)
```

- **Update Order**

Endpoint	*PUT /order/{id}*
Function	*update(db: Session, item_id, request)*
Description	Updates a specific order. Supports partial updates by excluding unset fields.
Path Parameters	*id* (integer): The ID of the order to update.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If the ID does not exist.) 404 Not Found (If validation fails or a DB error occurs.)

```
def update(db: Session, item_id, request):
   try:
       item = db.query(model.Order).filter(model.Order.id == item_id)
       if not item.first():
                    raise  HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Id not found!")
       update_data = request.dict(exclude_unset=True)
       item.update(update_data, synchronize_session=False)
       db.commit()
   except SQLAlchemyError as e:
       error = str(e.__dict__['orig'])
```

```
                   raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
   return item.first()
```

```
@router.put("/{item_id}", response_model=schema.Order)
def   update(item_id:   int,   request:   schema.OrderUpdate,   db:   Session   =
Depends(get_db)):
   return controller.update(db=db, request=request, item_id=item_id)
```

- **Delete Order**

Endpoint	*DELETE /orders/{id}*
Function	*delete(db: Session, item_id)*
Description	Deletes a specific order by its ID.
Path Parameters	*id* (integer): The ID of the order to delete.
Successful Request Status Code	204 No Content (No body is returned if deletion is successful)
Unsuccessful Request Status Code	400 Bad Request (If the specified ID does not exist.) 404 Not Found (If deletion fails due to a DB error.)

```
def delete(db: Session, item_id):
   try:
       item = db.query(model.Order).filter(model.Order.id == item_id)
       if not item.first():
                    raise  HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Id not found!")
       item.delete(synchronize_session=False)
       db.commit()
   except SQLAlchemyError as e:
       error = str(e.__dict__['orig'])
                   raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
   return Response(status_code=status.HTTP_204_NO_CONTENT)
```

```
@router.delete("/{item_id}")
def delete(item_id: int, db: Session = Depends(get_db)):
   return controller.delete(db=db, item_id=item_id)
```

- **Get All Orders by Order Date**

Endpoint	*GET /orders/read_all_by_date*
Function	*read_all_by_date(db: Session)*

Description	Retrieves all order records in the system, sorted by order date.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If a query error occurs.)

```
def read_all_by_date(db: Session):
   try:
       result = db.query(model.Order).order_by(model.Order.order_date).all()
   except SQLAlchemyError as e:
       error = str(e.__dict__['orig'])
                   raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
   return result
```

```
@router.get("/read_all_by_date", response_model=list[schema.Order])
def read_all_by_date(db: Session = Depends(get_db)):
   return controller.read_all_by_date(db)
```

- **Get All Orders by Order Status**

Endpoint	*GET /orders/real_all_by_status*
Function	*read_all_by_status(db: Session)*
Description	Retrieves all order records in the system, sorted by order status.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If a query error occurs.)

```
def read_all_by_status(db: Session):
   try:
                                                                 result         =
db.query(model.Order).order_by(desc(model.Order.order_status)).all()
   except SQLAlchemyError as e:
       error = str(e.__dict__['orig'])
                   raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
   return result
```

```
@router.get("/read_all_by_status", response_model=list[schema.Order])
def read_all_by_status(db: Session = Depends(get_db)):
   return controller.read_all_by_status(db)
```

- **Get All Orders by Order Number**

Endpoint	*GET /orders/read_all_by_number*
Function	*read_all_by_number(db: Session)*
Description	Retrieves all order records in the system, sorted by order number.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request (If a query error occurs.)

```
def read_all_by_number(db: Session):
   try:
       result = db.query(model.Order).order_by(model.Order.order_number).all()
   except SQLAlchemyError as e:
       error = str(e.__dict__['orig'])
                   raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=error)
   return result

@router.get("/read_all_by_number", response_model=list[schema.Order])
def read_all_by_number(db: Session = Depends(get_db)):
   return controller.read_all_by_number(db)
```

order_mgmt.py

The *order_mgmt.py* file provides a set of endpoints under the */orderstatus* prefix to manage and update the status of existing orders in the database. These routes are grouped under the Order Status Mgmt tag and support modifying an order's *order_status* field to predefined stages: "Received", "In Progress", and "Completed".

- **Update Order Status to Received**

Endpoint	*PUT /order_status/_received/{item_id}*
Function	*update_received*
Description	Updates the *order_status* of a specified order to "Received".
Path Parameters	*item_id* (integer): The unique ID of the order to update.
Successful	200 OK

Request Status Code	
Unsuccessful Request Status Code	400 Bad Request 404 Not Found (if the order doesn't exist or the update fails.)

```
@router.put("_received/{item_id}", response_model=schema.Order)
def   update_received(item_id:   int,   request:   schema.OrderStatusReceived,   db:
Session = Depends(get_db)):
   return controller.update(db=db, request=request, item_id=item_id)
```

- **Update Order Status to In Progress**

Endpoint	*PUT /order_status/_in_progress/{item_id}*
Function	*update_in_progress*
Description	Updates the *order_status* of a specified order to "In Progress".
Path Parameters	*item_id* (integer): The unique ID of the order to update.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request 404 Not Found

```
@router.put("_in_progress/{item_id}", response_model=schema.Order)
def update_in_progress(item_id: int, request: schema.OrderStatusInProgress, db:
Session = Depends(get_db)):
   return controller.update(db=db, request=request, item_id=item_id)
```

- **Update Order Status to Completed**

Endpoint	*PUT /order_status/_completed/{item_id}*
Function	*update_completed*
Description	Updates the *order_status* of a specific order to "Completed".
Path Parameters	*item_id* (integer): The unique ID of the order to update.
Successful Request Status Code	200 OK
Unsuccessful Request Status Code	400 Bad Request 404 Not Found

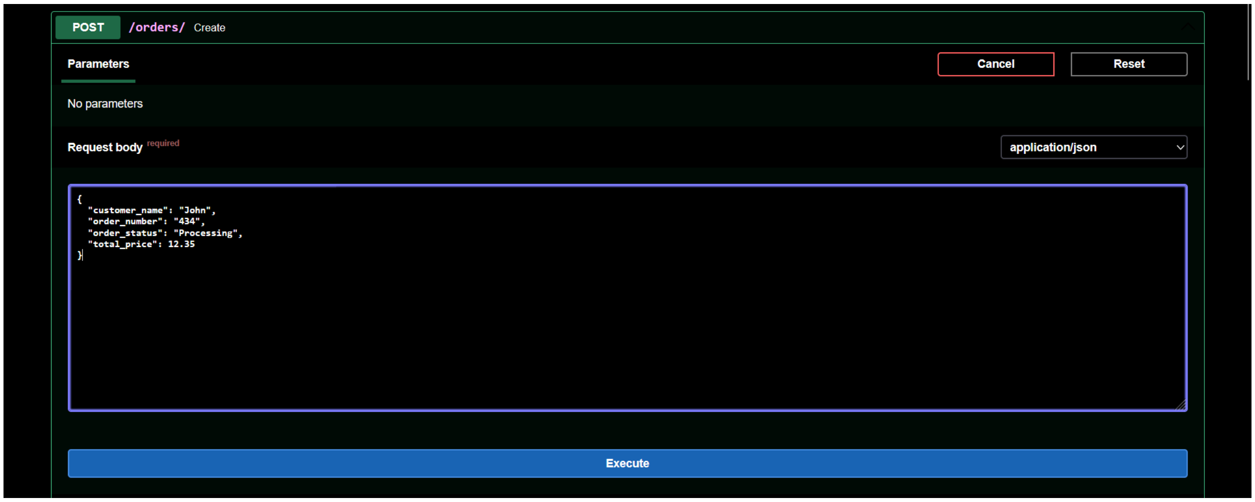
```
@router.put("_completed/{item_id}", response_model=schema.Order)
```

```
def  update_completed(item_id:  int,  request:  schema.OrderStatusCompleted,  db:
Session = Depends(get_db)):
   return controller.update(db=db, request=request, item_id=item_id)
```

Endpoint (Sample Requests)

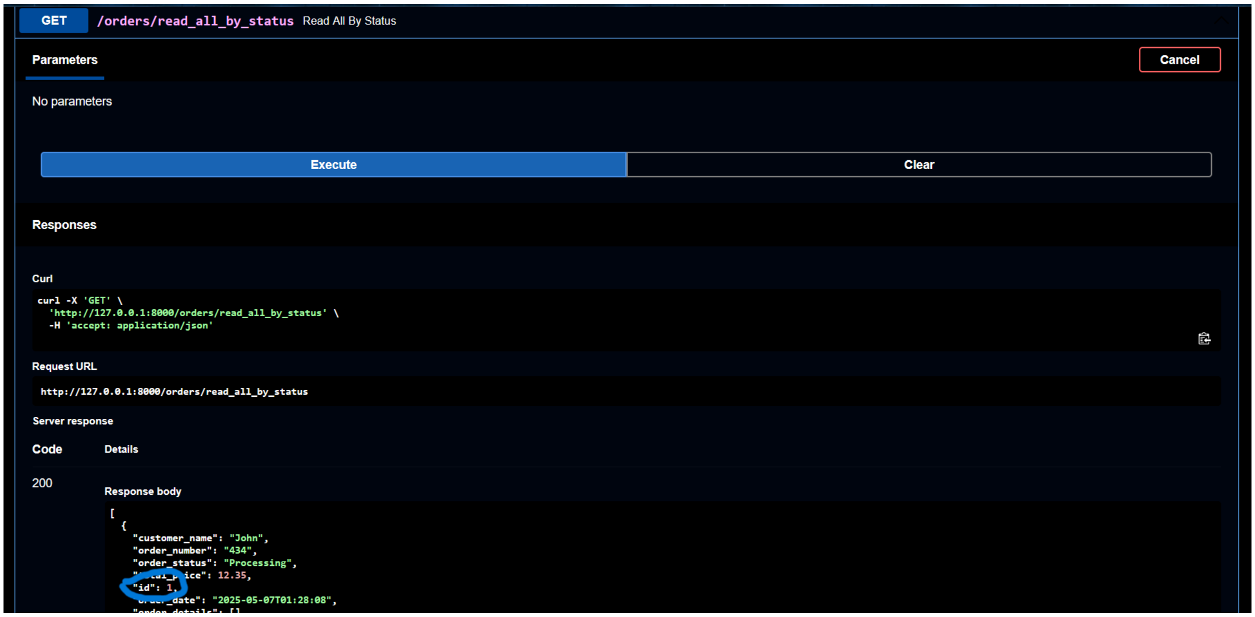
Create and Update an Order

Under the “*POST /orders/ Create*” endpoint, replace “*string*” in “*customer_name*” and “*order_number*” with the related information. Replace 0 with the price for the order:



```
{
  "customer_name": "John",
  "order_number": "434",
  "order_status": "Processing",
  "total_price": 12.35
}
```

Now, click Execute, and the order should be added to the database. You can check that the order posted by using any of the GET buttons:



Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/orders/read_all_by_status' \
  -H 'accept: application/json'
```

Request URL

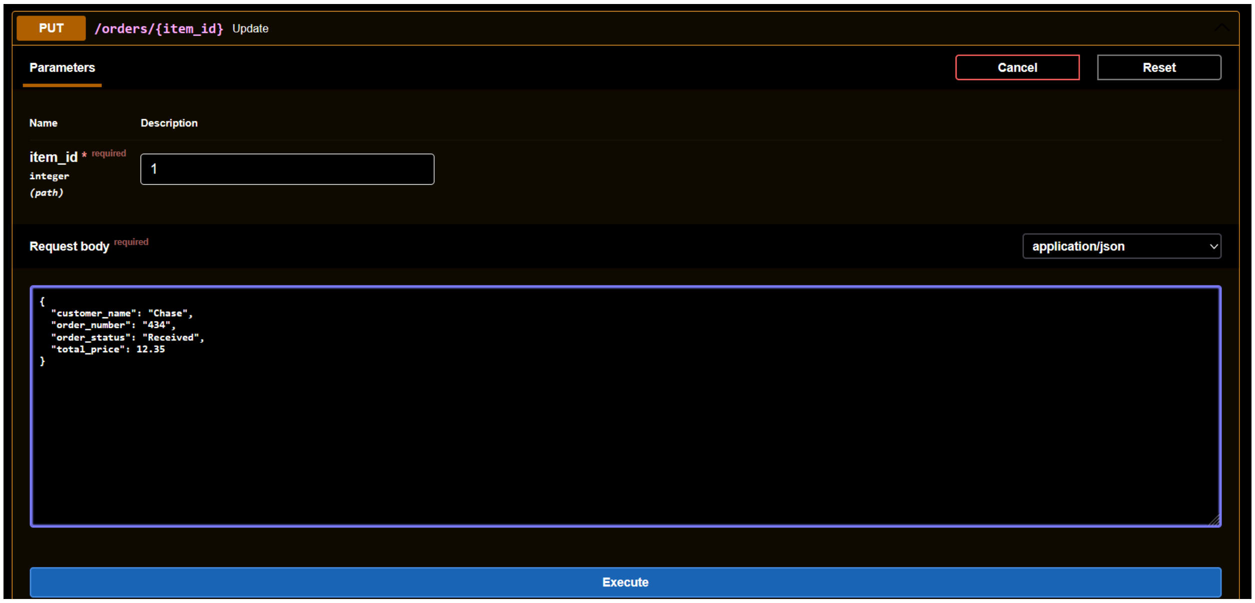
```
http://127.0.0.1:8000/orders/read_all_by_status
```

Server response

Code	Details
200	Response body

```
[
  {
    "customer_name": "John",
    "order_number": "434",
    "order_status": "Processing",
    "total_price": 12.35,
    "id": 1,
    "order_date": "2025-05-07T01:28:08",
    "order_details": []
```

To update an order using the “*PUT /orders/ Update*” endpoint, use the order’s “id” number [circled in the above image] and fill in the updated information. Then, click Execute:

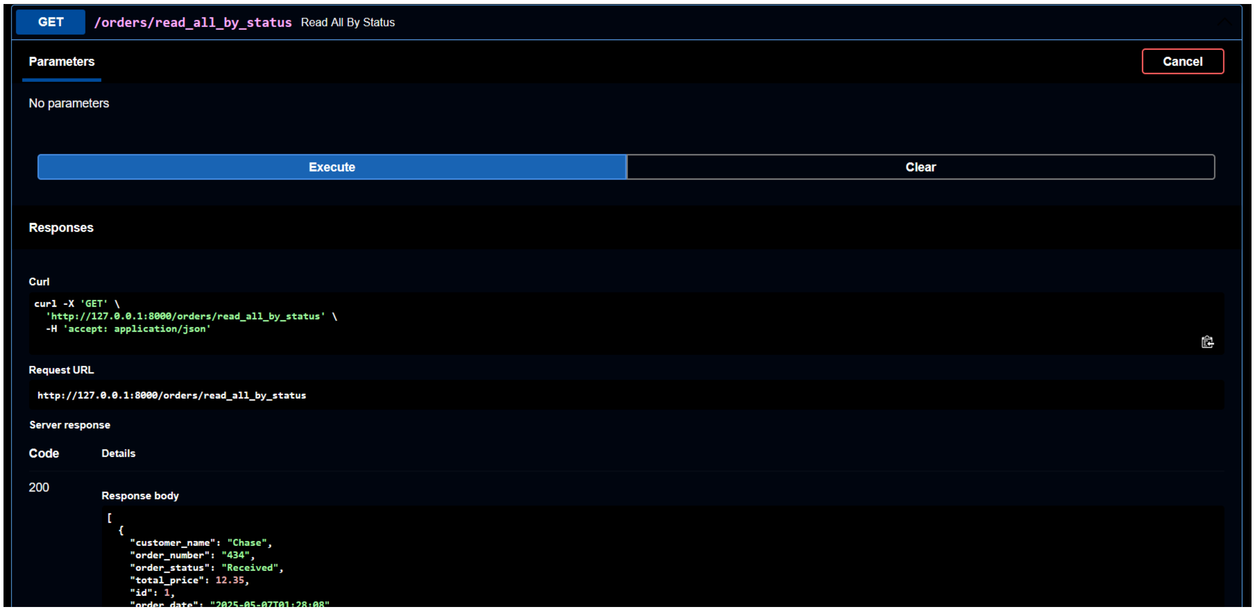


item_id * required
integer
(path)
1

application/json

```
{
  "customer_name": "Chase",
  "order_number": "434",
  "order_status": "Received",
  "total_price": 12.35
}
```

Updated order viewed in the database:



```
curl -X 'GET' \
  'http://127.0.0.1:8000/orders/read_all_by_status' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/orders/read_all_by_status
```

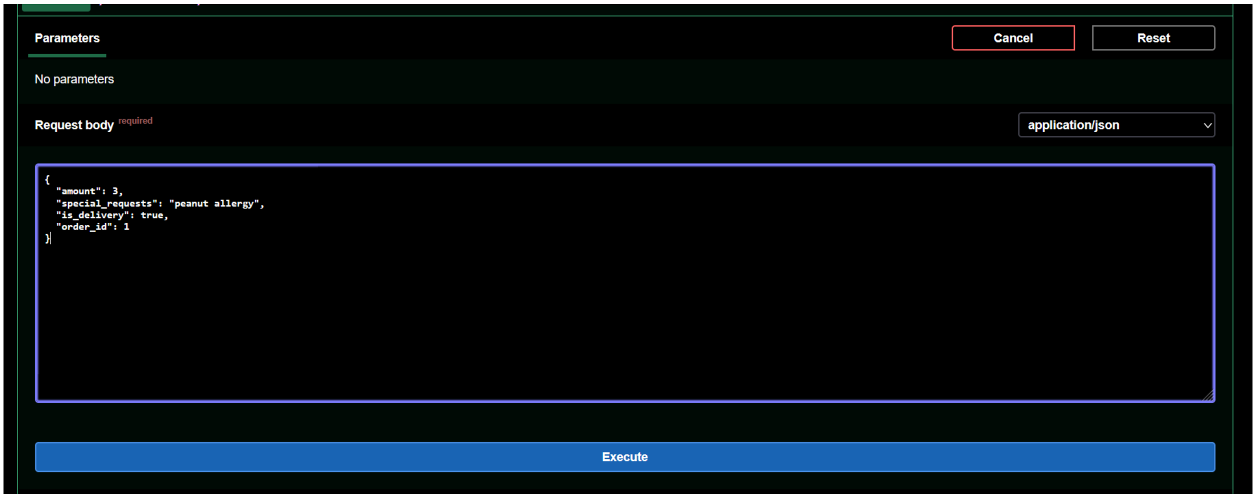
Server response

200

```
[
  {
    "customer_name": "Chase",
    "order_number": "434",
    "order_status": "Received",
    "total_price": 12.35,
    "id": 1,
    "order_date": "2025-05-07T01:28:08"
```

Add Details to an Order (Order Details)

After an order has been created with the Order endpoint, you can add more information using Order Details. Under the “*POST /orders/ Create*” endpoint, fill out the required information and replace the number next to “*order_id*” with the “*id*” number from the order these details will be attached to:



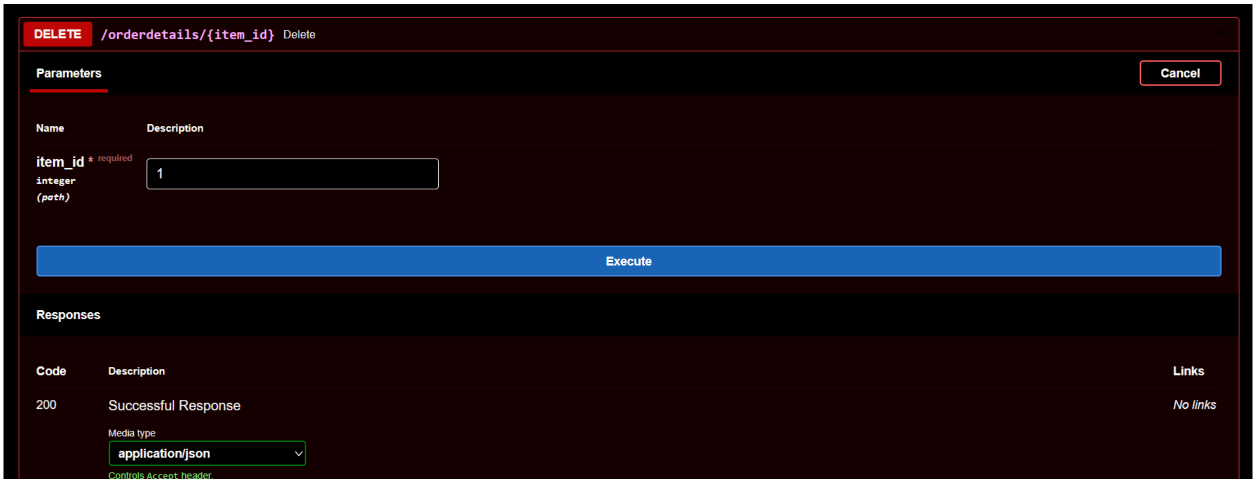
No parameters

Request body required

```
{
  "amount": 3,
  "special_requests": "peanut allergy",
  "is_delivery": true,
  "order_id": 1
}
```

Delete an Order

When deleting an order, first delete the details using the “*DELETE /orderdetails/{item_id} Delete”* endpoint using the id number for the order. Then, do the same thing with the “*DELETE /orders/{item_id} Delete*” endpoint.

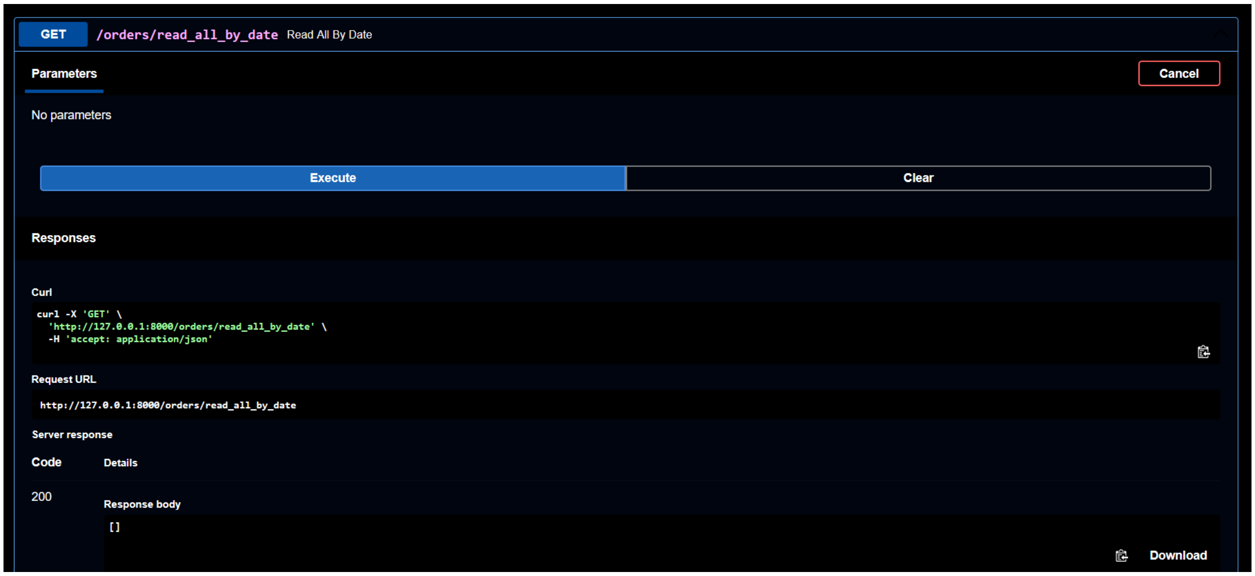


/orderdetails/{item_id} Delete

Name	Description
item_id * required integer *(path)*	1

Code	Description	Links
200	Successful Response Media type application/json Controls Accept header.	*No links*

View endpoint showing that the order was removed (thus, an empty database):



No parameters

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/orders/read_all_by_date' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/orders/read_all_by_date
```

Server response

Code	Details
200	Response body `[]`

Development Environment

Required Tools and Technologies

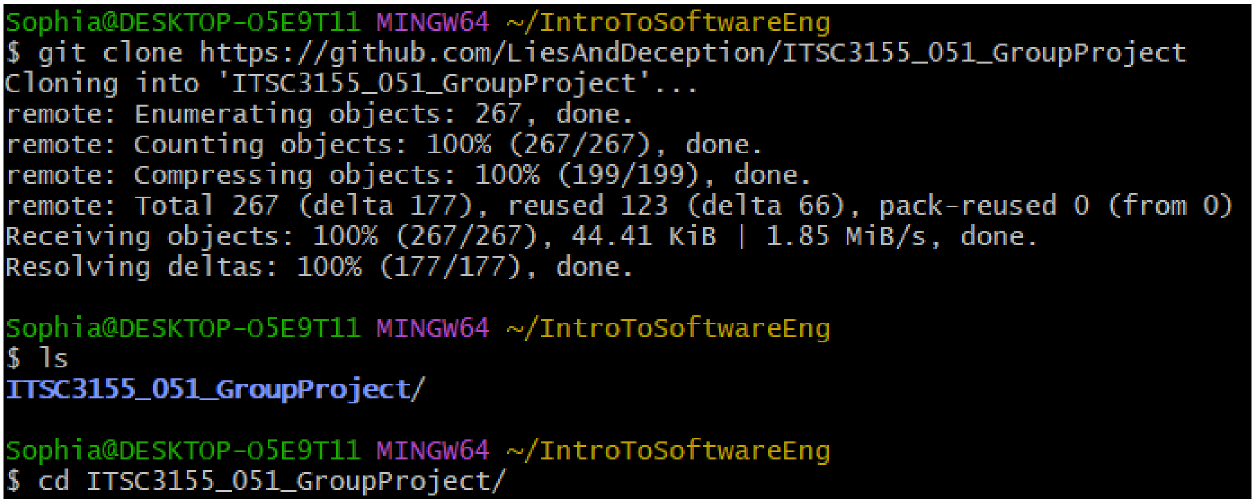
To begin working with the project, ensure that you have the following tools installed:

- **Python 3.10+**
- **MySQL** (ensure you have your password and access credentials ready)
- **FastAPI** (for building APIs)
- **PyCharm** (recommended IDE, but any terminal or integrated development environment will work)
- **GitHub Integration** in your IDE (to pull/push code from the repository)
- **Git** (for version control and cloning the repository)

Setup Instructions (Also referenced in the User Manual)

1. **Clone the Github repository**

   Start by cloning the project repository to your local machine.



```
Sophia@DESKTOP-05E9T11 MINGW64 ~/IntroToSoftwareEng
$ git clone https://github.com/LiesAndDeception/ITSC3155_051_GroupProject
Cloning into 'ITSC3155_051_GroupProject'...
remote: Enumerating objects: 267, done.
remote: Counting objects: 100% (267/267), done.
remote: Compressing objects: 100% (199/199), done.
remote: Total 267 (delta 177), reused 123 (delta 66), pack-reused 0 (from 0)
Receiving objects: 100% (267/267), 44.41 KiB | 1.85 MiB/s, done.
Resolving deltas: 100% (177/177), done.

Sophia@DESKTOP-05E9T11 MINGW64 ~/IntroToSoftwareEng
$ ls
ITSC3155_051_GroupProject/

Sophia@DESKTOP-05E9T11 MINGW64 ~/IntroToSoftwareEng
$ cd ITSC3155_051_GroupProject/
```

   You can access the GitHub repository here: [LiesAndDeception/ITSC3155_051_GroupProject: Group Project Repository for ITSC3155-051](https://github.com/LiesAndDeception/ITSC3155_051_GroupProject)

2. **Install Python Dependencies**

   Install the required Python packages using pip:

```
pip install fastapi
pip install "uvicorn[standard]"
pip install sqlalchemy
pip install pymysql
pip install pytest
pip install pytest-mock
pip install httpx
pip install cryptography
```

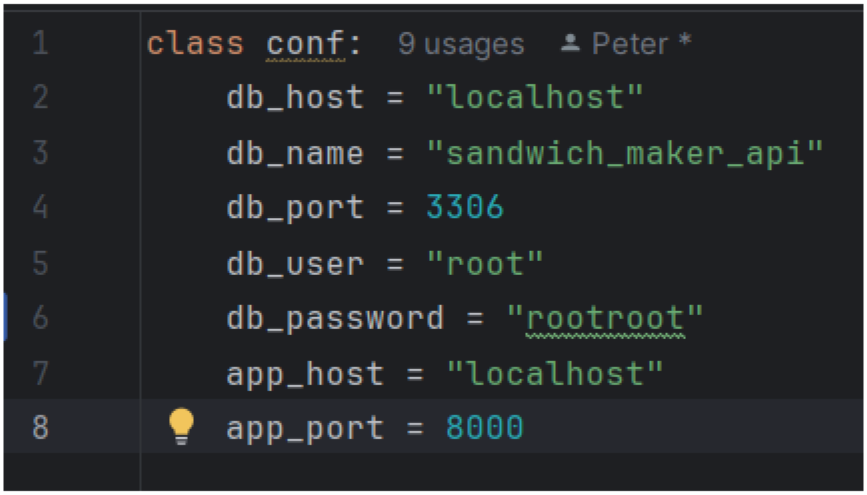
You may also choose to store and install dependencies via a requirements.txt file for ease of use:

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

3. **Configure MySQL**

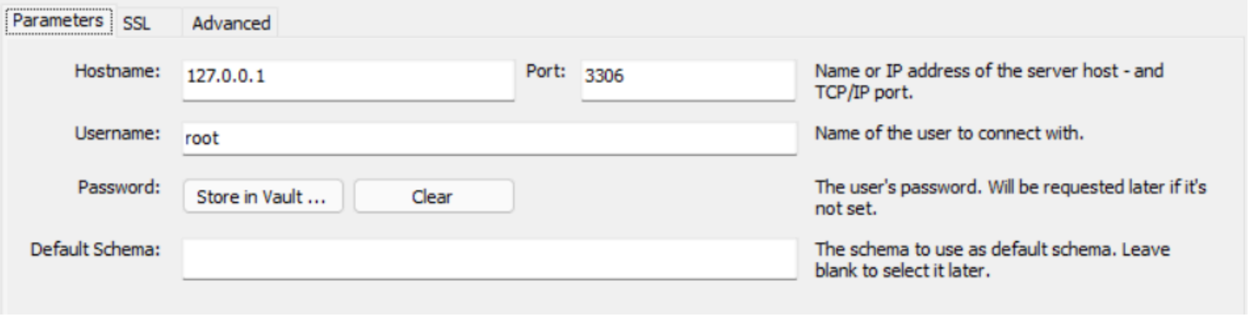
   In api\dependencies\config.py, replace “db_name” with a chosen name for the api (i.e. “restaurant_system_api”), and replace “db_password” with the password for your MySQL Workbench local instance connection.

   api\dependencies\config.py with a placeholder password and “sandwich_maker_api” as a database name:



```
class conf:
    db_host = "localhost"
    db_name = "sandwich_maker_api"
    db_port = 3306
    db_user = "root"
    db_password = "rootroot"
    app_host = "localhost"
    app_port = 8000
```

   Other “db_xxx” variable values should reflect the MySQL local connection information, such as the instance below.



Hostname:	127.0.0.1 Port: 3306	Name or IP address of the server host - and TCP/IP port.
Username:	root	Name of the user to connect with.
Password:		The user's password. Will be requested later if it's not set.
Default Schema:		The schema to use as default schema. Leave blank to select it later.

4. **Run the Development Server**

   Use Uvicorn to start the FastAPI server with hot reloading enabled:

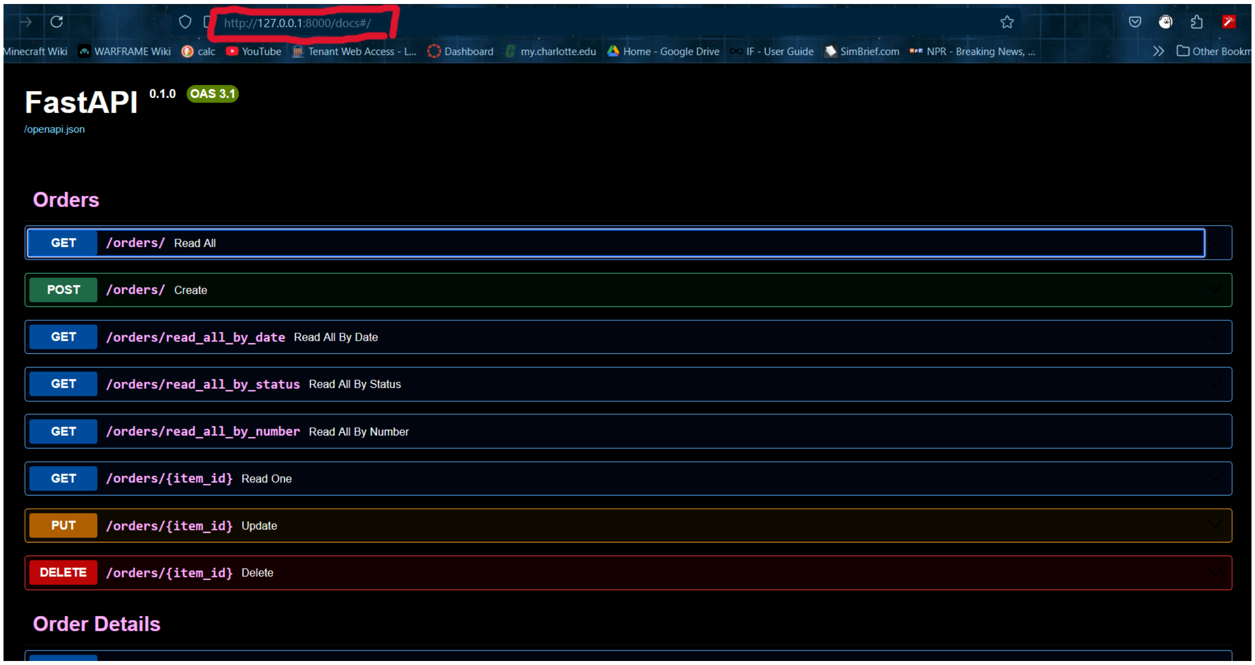
```
uvicorn api.main:app --reload
```

5. **Access the API Documentation**

Once the server is running, navigate to the following URL in your browser to interact with the API via the built-in documentation interface:

http://127.0.0.1:8000/docs

This interface allows you to test API endpoints and view schema definitions directly from the browser.



FastAPI 0.1.0 OAS 3.1

/openapi.json

Orders

GET /orders/ Read All

POST /orders/ Create

GET /orders/read_all_by_date Read All By Date

GET /orders/read_all_by_status Read All By Status

GET /orders/read_all_by_number Read All By Number

GET /orders/{item_id} Read One

PUT /orders/{item_id} Update

DELETE /orders/{item_id} Delete

Order Details