Team Presentation : Nigar Alkhasova, Nurkyz Bolotbekova, Fevzi Kultar

2025

Software development methodologies

Name of Project:
Food Ordering System

Presented By:
Nigar Alkhasova

Presented To:
Marcin Kacprowicz

General Information

Project Title: Food Ordering System
Authors:
Program / Year: 3rd year
Semester: 5th Semestr
Supervisor: Marcin Kacprowicz AEH
Submission Date:

Project Objective

The goal of this project is to design and implement a Food Ordering System that enables users to browse restaurant menus, add items to their cart, and place online orders efficiently. The system also allows administrators to manage menu items and monitor incoming orders.

The project aims to demonstrate the complete software development process using the Scrum methodology, including requirements analysis, system design with UML diagrams, implementation, and testing. The purpose of the system is to provide a fast, user-friendly, and secure online platform for food ordering that can be accessed from both desktop and mobile devices.

Project Scope

The Food Ordering System consists of several main components, each responsible for a specific function within the system.
The scope includes the following elements:
Web Application (Frontend)
Provides a user-friendly interface for customers to browse restaurant menus, add items to the cart, and place orders.
Developed using modern web technologies such as React.js or HTML/CSS for responsiveness and accessibility across desktop and mobile devices.

Backend Web Service (API)

Implements all business logic, including user authentication, order processing, and menu management.
Developed using Python (Flask or FastAPI) following the MVC architectural pattern and exposing RESTful API endpoints.

Database

Stores system data such as users, menu items, orders, and order statuses.
Implemented using PostgreSQL, ensuring data consistency, security, and reliability.

Admin Panel

A web-based dashboard for administrators to manage menu items, view customer orders, and update order statuses.
Accessible only to authorized admin users.

Deployment & Containerization

The complete system will be deployed using Docker Compose, ensuring modularity, scalability, and ease of maintenance.

Version Control & Collaboration

All source code and documentation will be managed using Git and GitHub to enable team collaboration and version tracking.

System Requirements

1. Functional Requirements

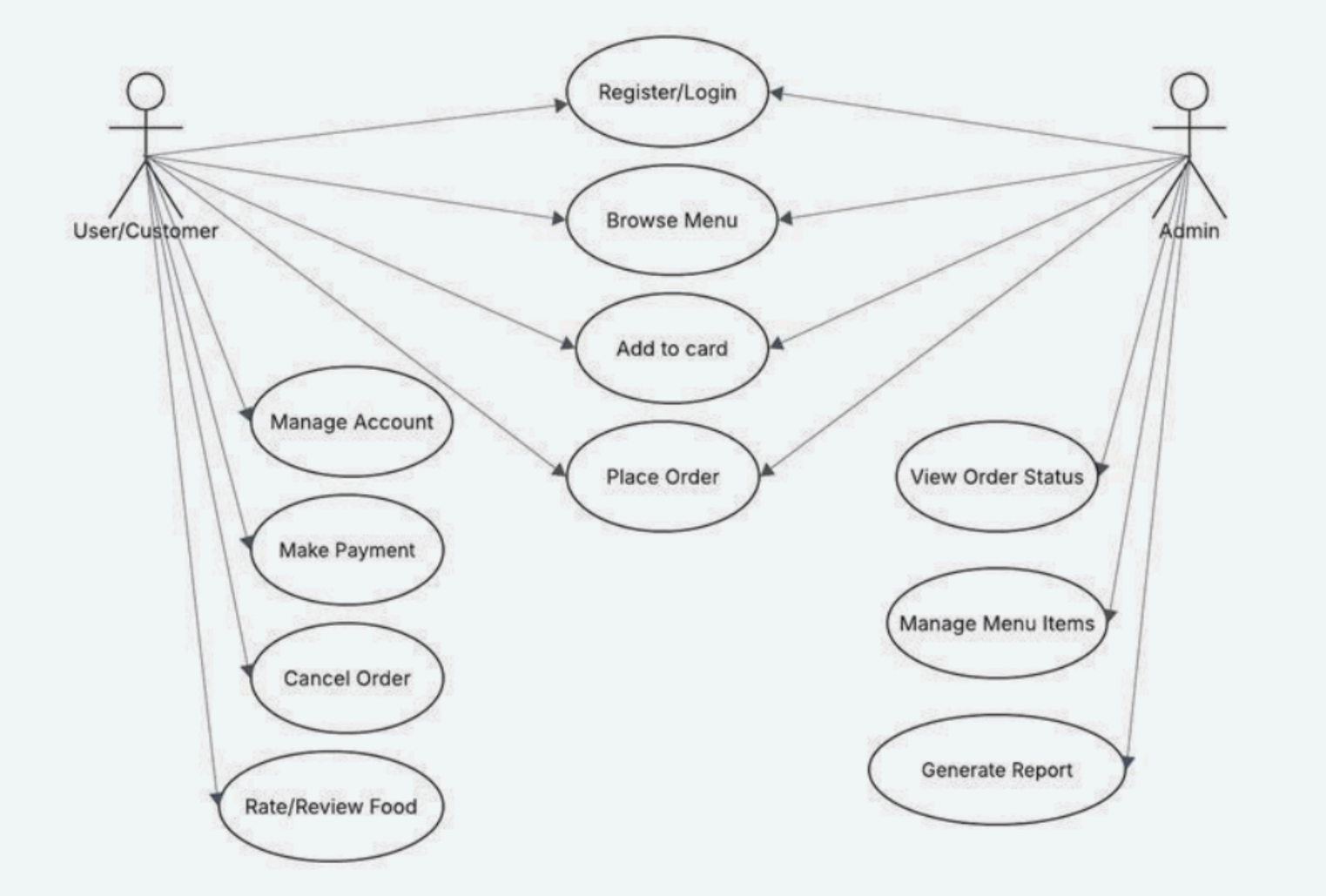
ID	Function Name	Description	Priority
FR1	User Registration & Login	Users can create an account and log in securely using their email and password.	High
FR2	Browse Menu	Users can view available menu items with names, categories, and prices.	High
FR3	Add to Cart & Checkout	Users can add items to their cart and place an order.	High
FR4	View Order Status	Users can check their order status (Pending, Preparing, Delivered).	High
FR5	Admin Menu Management	Admins can add, edit, or delete menu items.	High
FR6	Admin Order Management	Admins can manage incoming orders and update their statuses.	High
FR7	User Profile Management	Users can view or update their account information.	Medium
FR8	Notifications	The system sends notifications or updates when the order status changes.	Medium

2. Non-functional Requirements

ID	Function Name	Description	Priority
NFR1	Performance / Response Time	API requests shall respond within 2 seconds under normal load (95th percentile)	High
NFR2	Security & Privacy	Passwords must be hashed (bcrypt/argon2). All sensitive data transmitted over TLS. Admin endpoints require role-based authorization.	High
NFR3	Availability / Recovery	System availability shall be ≥ 99%; maximum acceptable recovery time after failure ≤ 8 hours.	High
NFR4	Data Consistency	Order creation (Order + OrderItems) must be atomic (use DB transactions) to avoid partial or inconsistent orders.	High
NFR5	Usability / Mobile-friendly	UI must be responsive and allow key user flows (browse, add to cart, checkout) on common mobile screen sizes.	Medium

UML Diagrams

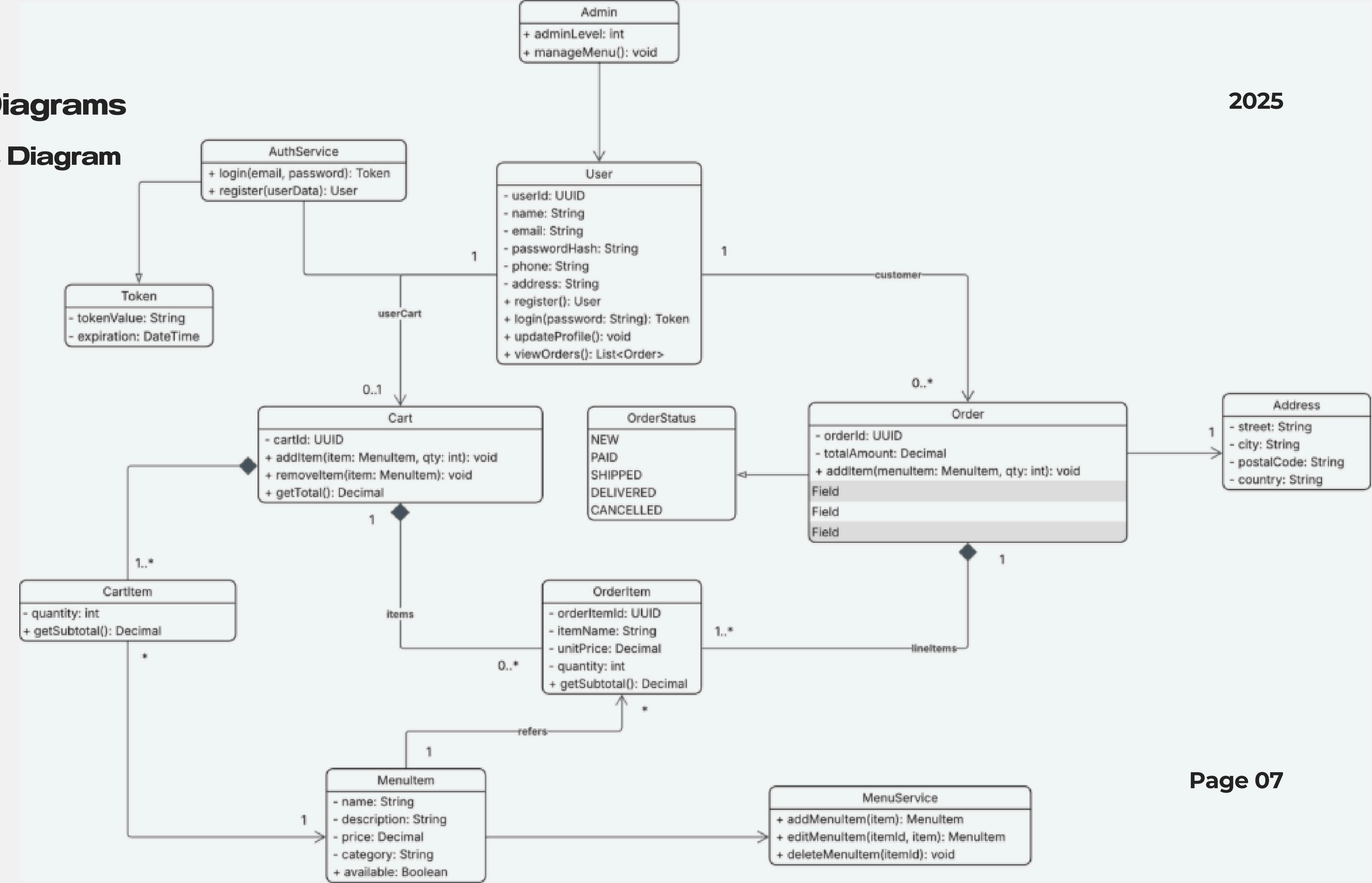
1. Use Case Diagram



UML Diagrams

2025

2. Class Diagram



Methodology Implementation

1. Implementation of the Selected Methodology – Scrum

Our team selected Scrum as the primary software development methodology due to its iterative nature, adaptability, and strong focus on communication and incremental delivery. Scrum is well-suited for modern web systems such as the Food Ordering System, where requirements may evolve, new functionalities may emerge, and components need to be developed in small, manageable increments.

Scrum allowed the team to:

- break down the system into smaller functional pieces,
- prioritize tasks based on importance,
- deliver working features at the end of each sprint,
- continuously evaluate progress and make adjustments when needed.

To support the Scrum workflow:

- GitHub was used for version control, pull requests, and tracking code contributions.
- Trello / Notion was used for backlog management, sprint boards, and tracking task progress.
- Microsoft Teams & WhatsApp were used for coordination and check-ins.

Sprint Planning

At the beginning of each sprint, the team reviewed the Product Backlog, which included:

- functional requirements (FR1–FR8),
- system features,
- UML diagrams,
- API endpoints,
- testing tasks.

During Sprint Planning:

- high-priority tasks were selected for the Sprint Backlog,
- the team estimated workload using story points,
- responsibilities were distributed based on team members' strengths (backend, frontend, database, documentation).

Scrum planning helped create a realistic and structured development timeline.

Sprint Execution

2025

The development phase was carried out across three sprints, each with clearly defined goals.

Sprint 1 — Requirements & System Design

Duration: 2 weeks
Main Deliverables:

- Requirements specification (FR & NFR)
- System scope documentation
- UML Use Case Diagram
- UML Class Diagram
- Initial project structure (Flask setup, GitHub repository)
-

Activities Completed:

- Analyzed functional needs of the Food Ordering System
- Created diagrams representing system behavior and data model
- Set up Flask project, virtual environment, and base folders
- Prepared initial database structure (User, MenuItem, Order)

Outcome:
A solid technical foundation and complete system design ready for implementation.

Sprint 2 — Core Functionalities Development

Duration: 2 weeks
Main Deliverables:

- User Authentication (JWT)
- Menu Browsing Endpoint
- Cart Management Endpoints
- Initial UI for login and menu display

Activities Completed:

- Implemented registration and login logic
- Added JWT-based authentication middleware
- Developed menu listing API
- Added routes for adding/removing items from the cart
- Connected controllers with database models
- Conducted integration tests on authentication and cart flow

Outcome:
The system became functional for basic user flows such as login, viewing menu, and interacting with the cart.

Sprint Execution

Sprint 3 — Order Management & Testing

Duration: 2 weeks
Main Deliverables:

- Order placement endpoint
- Admin features (add/delete menu items, update order status)
- Full testing suite (unit, integration, functional)
- Final documentation and project polishing

Activities Completed:

- Developed API to submit orders and track statuses
- Implemented admin routes for menu and order management
- Conducted UI testing and debugging
- Performed system and regression testing
- Finalized methodology and technical documentation

Outcome:
A fully working system that supports complete user and admin workflows.

Daily Scrum / Check-ins

Although physical daily meetings were not possible, the team followed the spirit of Daily Scrum by:

- sharing progress updates in a WhatsApp group,
- reporting blockers (e.g., Flask configuration, JWT expiration),
- assigning and reassigning tasks as needed,
- checking Trello status updates.

Short asynchronous check-ins ensured continuous progress and coordination.

Sprint Review

At the end of each sprint, the team demonstrated:

- **Sprint 1:** UML diagrams and initial project structure
- **Sprint 2:** Working authentication and menu browsing
- **Sprint 3:** Order placement, admin panel operations, and test results
-

Feedback from the supervisor helped improve:

- clarity of API documentation,
- database relationships,
- error handling mechanisms.

Sprint Retrospective

After each sprint, the team reflected on:

- what worked well,
- what challenges occurred,
- what improvements were needed for the next sprint.
-

Improvements included:

- better task breakdown,
- more frequent communication,
- adding more detailed API logs,
- earlier testing of database queries.
-

This continuous improvement cycle strengthened team collaboration and workflow efficiency.

Testing Methods and Approaches

To verify stability, correctness, and compliance with requirements, multiple testing types were applied:

Unit Testing
- *Tested individual controllers and service functions*
- *Example: registration, JWT creation, add/remove cart item*

Integration Testing
- *Verified communication between components*
- *Login flow → Token generation → Authorized requests*
- *Cart → Order conversion*

Functional Testing
- *Validated FR1–FR8 (register, login, menu, checkout, order status)*

UI/UX Testing
- *Checked responsiveness, navigation flow, and visual consistency*
- *Ensured buttons, forms, and error messages worked correctly*

System Testing
- *Full end-to-end flow from account creation → order placement → admin update*

Regression Testing
- *Re-tested authentication and order components after bug fixes*
- *Ensured new changes didn't break existing functionality*

Technical Documentation of the Implemented Code

Overview

The backend was developed using **Python (Flask)** following the **MVC architecture** pattern.
The Flask application exposes a REST API that allows clients to interact with models through controllers.
Each API endpoint corresponds to a specific operation in the food ordering workflow.

Project Structure

```
/project
 ├── app.py
 ├── controllers/
 ├── models/
 ├── services/
 ├── templates/
 ├── static/
 ├── tests/
 ├── config/
 └── requirements.txt
```

The structure ensures:

- separation of concerns,
- scalability,
- clean responsibility distribution.

MVC Architecture in the Project

Component	Description	Example in Code
Model	Represents database schema & logic	User, Order, MenuItem
View	HTML templates or JSON responses	/templates/*.html + controller returns
Controller	Handles requests, interacts with models, prepares responses	user_controller.py

This pattern ensures clean organization and easier debugging/maintenance.

Design Pattern - Singleton

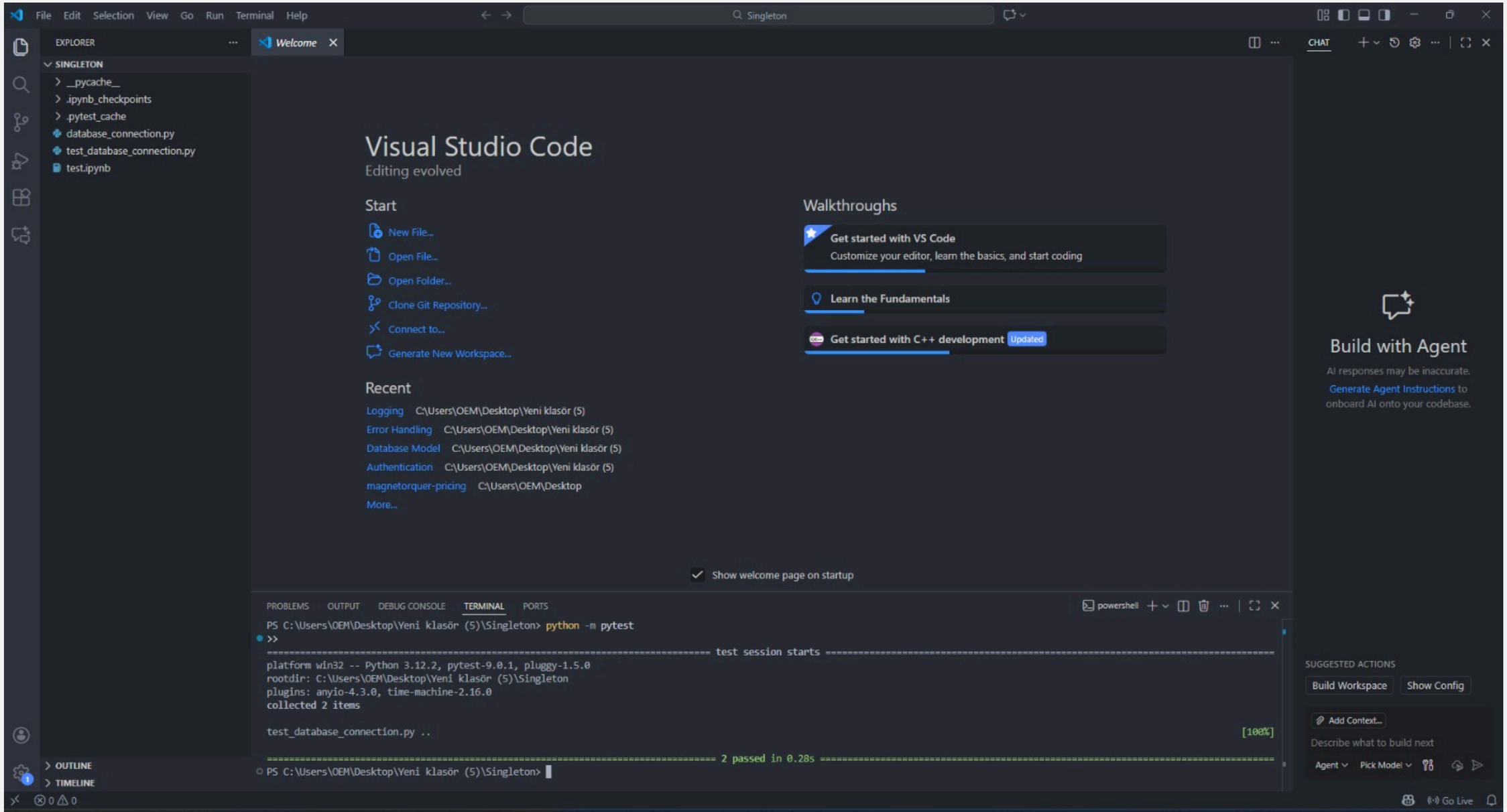
To guarantee that a single database connection is used throughout the application, the **Singleton** design pattern was implemented in db_connection.py.

Example:

```python
class DatabaseConnection:
    __instance = None

    @staticmethod
    def get_instance():
        if DatabaseConnection.__instance is None:
            DatabaseConnection()
        return DatabaseConnection.__instance

    def __init__(self):
        if DatabaseConnection.__instance is not None:
            raise Exception("This class is a singleton!")
        else:
            self.connection = self.create_connection()
            DatabaseConnection.__instance = self

    def create_connection(self):
        from sqlalchemy import create_engine
        return create_engine('sqlite:///food_ordering.db')
```

Design Pattern - Singleton - Output

2025



Design Pattern - Singleton

Explanation: The Singleton design pattern was applied in db_connection.py to ensure that only a single, centralized database connection instance exists throughout the entire application lifecycle. In many software systems, especially those interacting with relational or NoSQL databases, creating multiple parallel connections can lead to performance degradation, duplicated resource usage, and potential data inconsistencies. The Singleton pattern directly addresses this issue by restricting object creation and guaranteeing that all modules share the same connection instance.

By using this pattern, the application avoids unnecessary overhead from repeatedly opening new database connections. Instead, the first time the connection is requested, an instance is created and then reused every time afterward. This approach improves efficiency, reduces memory consumption, and ensures stable, predictable communication with the database. Additionally, having a single shared connection helps prevent concurrency-related problems, such as conflicting transactions or race conditions when multiple modules attempt to access the database simultaneously. The pattern also centralizes connection management, making debugging, configuration, and future maintenance significantly easier. Overall, the Singleton implementation contributes to a cleaner architecture and more reliable behavior in the data access layer.

REST API Endpoints

The backend follows **RESTful principles**, mapping each functionality to an appropriate HTTP method.

Functionality	Endpoint	Method	Description
Register User	/register	POST	Create a new user account
Login	/login	POST	Authenticate user and issue JWT token
View Menu	/menu	GET	Retrieve all available menu items
Add to Cart	/cart/add	POST	Add a selected item to the user's cart
Place Order	/order	POST	Submit a new order
View Orders	/orders/<user_id>	GET	Retrieve all orders for a given user
Add Menu Item (Admin)	/admin/add_item	POST	Add new menu item
Delete Menu Item	/admin/delete_item/<id>	DELETE	Remove menu item
Update Order Status	/admin/update_order/<id>	PUT	Update order status

Design Pattern - Singleton

The backend of the system follows **RESTful architectural principles**, where each operation within the application is represented by a meaningful and well-structured HTTP request. REST (Representational State Transfer) promotes clear separation of responsibilities between the client and server, making the system scalable, maintainable, and easy to integrate with external services. Each endpoint is designed to correspond to a specific resource or action, and the HTTP method indicates the nature of the operation—such as retrieving data, creating new records, updating existing information, or deleting items.

To ensure clarity and consistency, each functionality in the food ordering platform is mapped to an appropriate HTTP method:

- **POST** is used for actions that create new data, such as user registration, login to obtain a JWT token, adding items to the cart, placing orders, or adding new menu items by administrators.
- **GET** retrieves data without modifying it, such as fetching the full menu or listing all orders associated with a user.
- **PUT** updates existing records, such as changing an order's status.
- **DELETE** removes resources from the system, such as deleting menu items.

By adhering to REST conventions, the application ensures predictable behavior across endpoints, improves interoperability, and supports a clean, standardized communication flow between the client interface and backend server. This makes the system easier to test, maintain, and extend in future iterations.

Design Pattern - Singleton

The backend of the system follows **RESTful architectural principles**, where each operation within the application is represented by a meaningful and well-structured HTTP request. REST (Representational State Transfer) promotes clear separation of responsibilities between the client and server, making the system scalable, maintainable, and easy to integrate with external services. Each endpoint is designed to correspond to a specific resource or action, and the HTTP method indicates the nature of the operation—such as retrieving data, creating new records, updating existing information, or deleting items.

To ensure clarity and consistency, each functionality in the food ordering platform is mapped to an appropriate HTTP method:

- **POST** is used for actions that create new data, such as user registration, login to obtain a JWT token, adding items to the cart, placing orders, or adding new menu items by administrators.
- **GET** retrieves data without modifying it, such as fetching the full menu or listing all orders associated with a user.
- **PUT** updates existing records, such as changing an order's status.
- **DELETE** removes resources from the system, such as deleting menu items.

By adhering to REST conventions, the application ensures predictable behavior across endpoints, improves interoperability, and supports a clean, standardized communication flow between the client interface and backend server. This makes the system easier to test, maintain, and extend in future iterations.

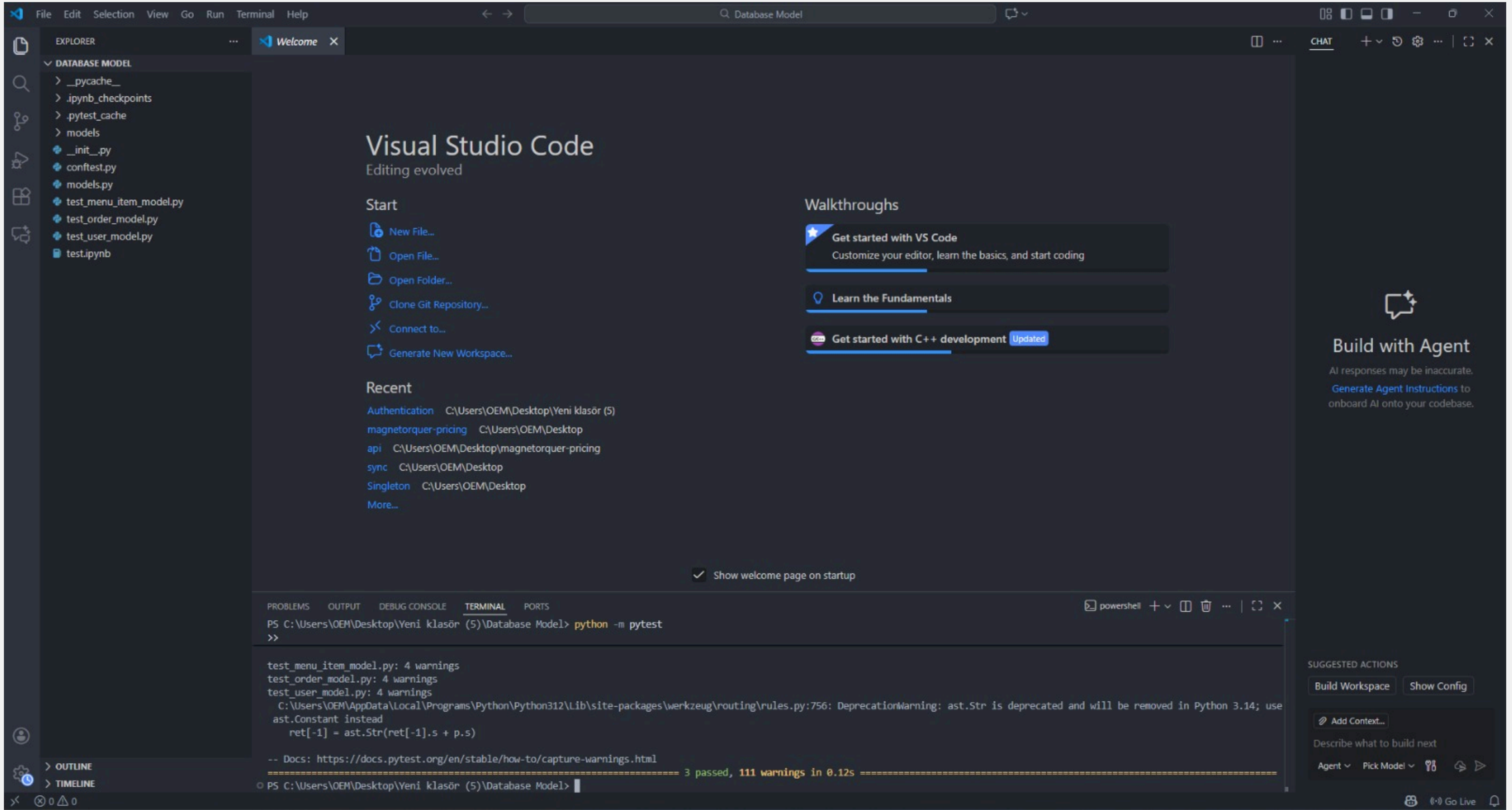
Database Model

The database contains four main tables: **User, MenuItem, Order, and OrderItem.** These models are implemented using SQLAlchemy.
Examples:

```python
import pytest
from flask import Flask
from models import db, User, MenuItem, Order

@pytest.fixture
def test_app():
    app = Flask(__name__)
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///:memory:"
    app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

    db.init_app(app)

    with app.app_context():
        db.create_all()
        yield app, db
```

```python
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()


class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True)
    password = db.Column(db.String(255))


class MenuItem(db.Model):
    __tablename__ = "menu_items"
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    price = db.Column(db.Float)


class Order(db.Model):
    __tablename__ = "orders"
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey("users.id"))
    status = db.Column(db.String(50))
```

Database Model - Output



Database Model

The database architecture of the application is designed to reflect the core components of a typical food ordering system. It is composed of four primary tables—**User, MenuItem, Order,** and **OrderItem**—each responsible for storing specific types of information essential to system operations. These data models are implemented using **SQLAlchemy**, a Python-based ORM (Object-Relational Mapping) library that allows the application to interact with the database through Python objects rather than raw SQL queries. This improves maintainability, readability, and reduces the likelihood of query-related errors.

- The **User** table stores authentication and profile information for each customer, including login credentials and personal details.
- The **MenuItem** table represents the available items on the restaurant's menu, including item names, prices, descriptions, and availability status.
- The **Order** table records each order placed by users, along with timestamps, total price, and current order status (e.g., pending, preparing, completed).
- The **OrderItem** table acts as a relational link between orders and menu items, capturing which items were included in each order and in what quantity.

Together, these tables create a structured relational model that ensures data integrity, supports efficient queries, and provides a reliable foundation for key operations such as user authentication, browsing menu items, placing orders, and order management. Using SQLAlchemy also allows for easier modifications and scalability, making the database layer flexible for future enhancements.

Authentication (JWT Security)

Secure user authentication is implemented using **JSON Web Tokens (JWT).**
Example:

```python
import jwt
import datetime

def create_jwt_token(user_id):
    payload = {
        'user_id': user_id,
        'exp': datetime.datetime.utcnow() + datetime.timedelta(hours=2)
    }
    return jwt.encode(payload, 'SECRET_KEY', algorithm='HS256')
```

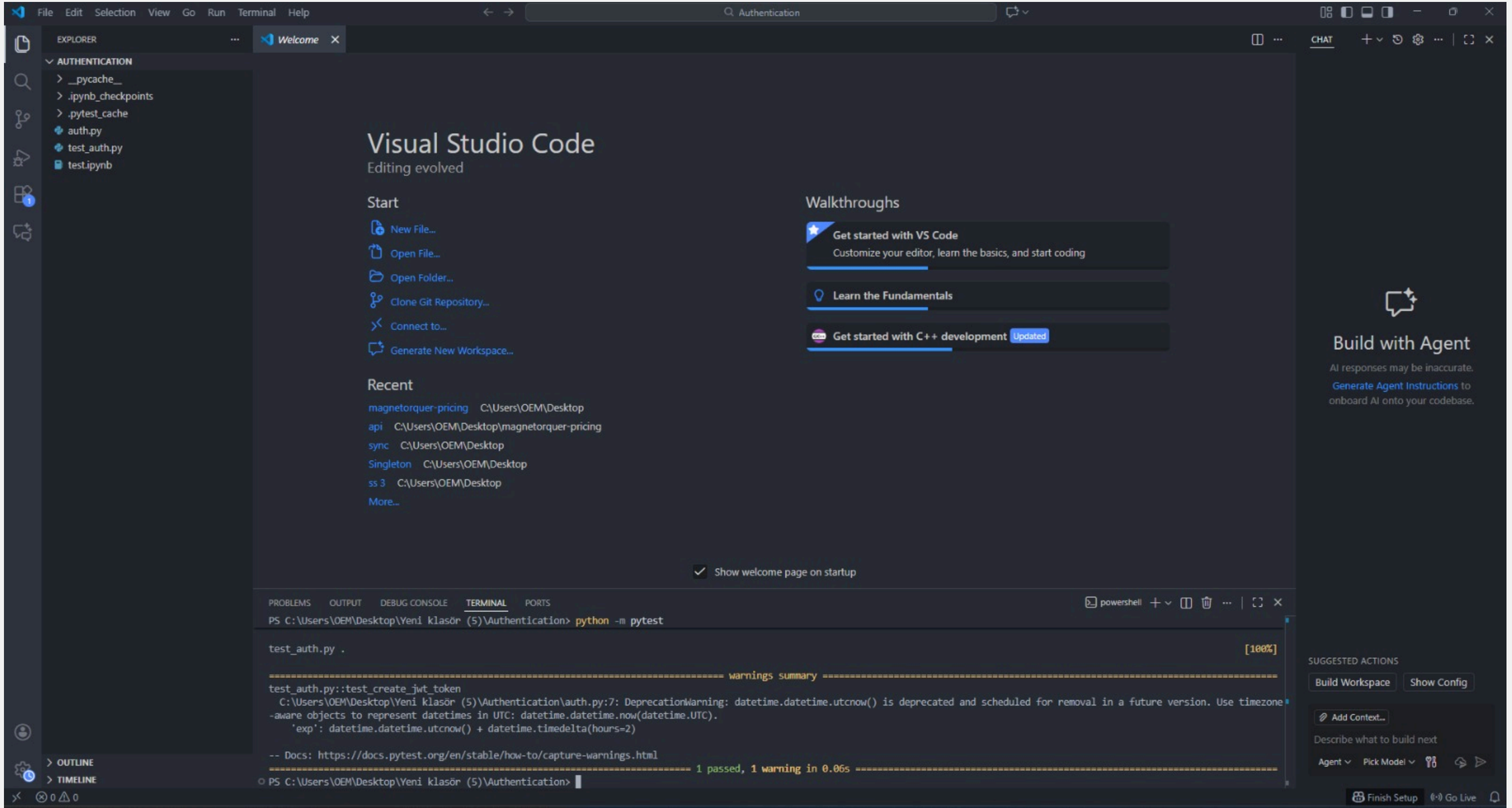
Secure user authentication in the system is implemented using JSON Web Tokens (JWT), a modern, stateless authentication mechanism widely used in distributed applications. JWT enables the backend to verify user identity without storing session data on the server, thereby improving scalability and reducing overhead.

During the login process, the server generates a signed JWT containing essential user information, such as the user ID and role. This token is returned to the client and must be included in the header of every subsequent API request. Because the token is cryptographically signed, the server can validate its authenticity and ensure that it has not been tampered with.
JWT provides several key security advantages:

- **Statelessness:** No server-side session storage is required, which simplifies architecture and improves performance.
- **Integrity:** Tokens are signed using a secret key, ensuring that only the server can issue valid tokens.
- **Role-based access control:** Admin-only functionality (such as adding menu items or updating order status) can be protected by verifying user roles embedded within the token.
- **Expiration handling:** Tokens include automatic expiration times to reduce the risk of unauthorized long-term access.

Overall, JWT ensures that each API request is securely authenticated, prevents unauthorized actions, and maintains a lightweight and efficient security model suitable for modern REST-based systems.

Authentication (JWT Security) - Output



Error Handling

Global and endpoint-specific exception handling ensures stable application behavior and consistent error responses.
Example:

```python
import pytest
from flask import Flask
from error_handler import register_error_handlers

@pytest.fixture
def app():
    app = Flask(__name__)
    register_error_handlers(app)
    return app
```

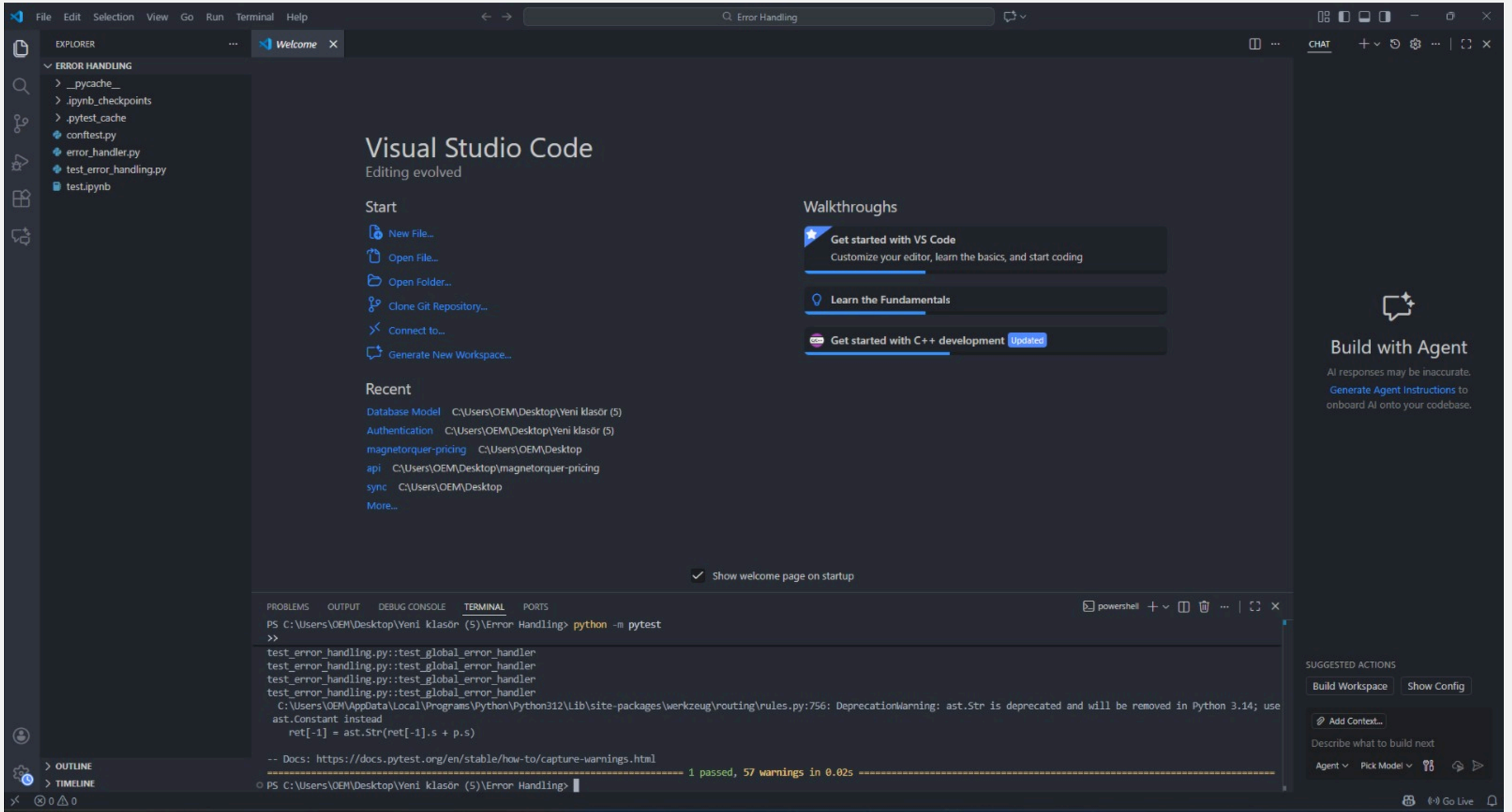
Robust error handling is implemented across the application to ensure stability, reliability, and a consistent user experience. The system uses a combination of **global exception handlers and endpoint-specific error management** to gracefully handle unexpected situations without causing application crashes or exposing sensitive internal details.

Global error handling intercepts common issues—such as invalid requests, authentication failures, missing data, or server-side exceptions—and transforms them into clear, standardized JSON responses. This prevents raw errors or stack traces from being returned to the client, improving both user experience and security.

Additionally, endpoint-level error handling provides granular control for operations that require more precise validation. For example, attempting to delete a non-existent menu item, placing an order with an empty cart, or accessing a restricted admin route triggers meaningful, descriptive error messages. These custom responses help clients understand what went wrong and how to correct the request.

Overall, this structured approach to exception handling enhances system robustness, simplifies debugging, and ensures consistent error formatting across all API endpoints. As a result, both developers and end users benefit from a predictable and user-friendly interaction workflow, even when errors occur.

Error Handling - Output



Logging

System logs are generated to support debugging, monitoring, and error tracking.
Example:

```python
# logger.py
import logging

def start_logging(log_file="app.log"):
    logging.basicConfig(
        filename=log_file,
        level=logging.INFO,
        force=True   # ← TESTLERİN ÇALIŞMASI İÇİN ŞART
    )
    logging.info("Application started successfully")
```

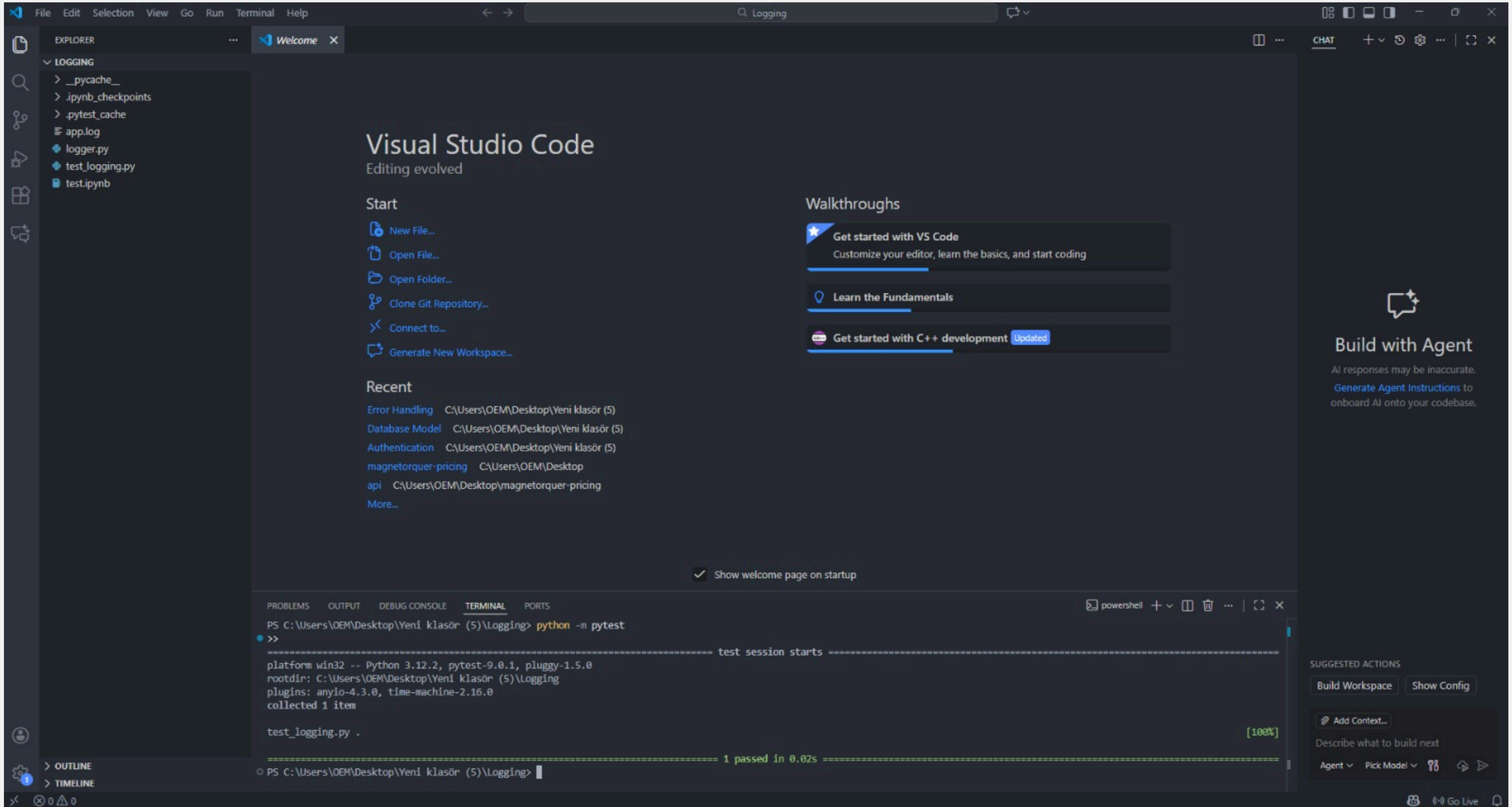
A comprehensive logging mechanism is integrated into the application to support effective debugging, monitoring, and long-term maintenance. The logging system captures essential runtime information, including incoming requests, system events, database interactions, and error messages. This allows developers to efficiently diagnose issues, trace execution flows, and monitor overall system health.

Logs serve as a critical source of insight when unexpected behavior occurs. For example, failed authentication attempts, invalid API requests, or exceptions thrown during order processing are recorded with timestamps and severity levels. This makes it easier to identify root causes and detect recurring patterns or potential security threats.

In addition to error tracking, logging provides valuable operational feedback—such as performance bottlenecks, slow queries, or unusual activity—that can guide future improvements. The structured logging format ensures consistency across components and simplifies integration with external monitoring tools if needed.

Overall, the logging module enhances transparency, reliability, and maintainability of the backend, ensuring that the system remains stable and easier to troubleshoot as it evolves.

Logging - Output



Testing Framework

The system was thoroughly tested using **pytest** in combination with **Flask's built-in test client**, providing a reliable and automated approach to validating application behavior. This testing framework enabled simulation of real API requests without running the server, ensuring that both functional logic and endpoint responses work as expected.

The test suite covers a wide range of core functionalities essential to the food ordering workflow:

- **User registration and login:**

Verifies correct handling of new users, JWT token generation, and authentication validation.

- **Adding items to the cart:**

Ensures that items are correctly added, validated, and stored in user-specific sessions or database records.

- **Placing orders:**

Tests order creation, price calculation, database persistence, and response formatting.

- **Admin menu management:**

Confirms that only authorized admin users can add, delete, or update menu items, including proper permission checks.

With approximately 80% code coverage, the testing process ensures the reliability of implemented features and reduces the likelihood of hidden bugs. The comprehensive nature of the tests provides confidence that the core functionality of the backend behaves consistently under various scenarios, supporting a stable and maintainable system.

Future Improvements

Several enhancements are planned to expand the system's capabilities and improve overall performance, usability, and maintainability. These future improvements aim to bring the application closer to real-world production standards and provide a more feature-rich experience for both users and administrators.

- **Payment Gateway Integration:**
- A secure payment module—such as a Stripe API mock—will be integrated to simulate real online transactions. This will allow users to complete orders with card payments and provide a more realistic checkout experience.
- **Enhanced Mobile Responsiveness:**
- The frontend will be optimized for mobile and tablet devices to ensure a seamless user experience across different screen sizes. Improvements in layout, touch interactions, and loading performance will increase accessibility and usability.
- **Docker Compose Deployment:**
- Introducing Docker Compose will streamline the deployment process by containerizing the backend, frontend, and database services. This will simplify environment setup, improve portability, and ensure consistent behavior across development and production systems.
- **Expanded Admin Analytics:**
- Additional analytics tools—such as sales reporting, order statistics, and heatmaps—will be developed to provide administrators with deeper insights into system performance and customer behavior. These features will support better decision-making and operational optimization.

Overall, these planned enhancements aim to increase system robustness, scalability, and user satisfaction while offering improved administrative oversight and smoother deployment workflows.

Below is the finished, report-ready chapter.

Architecture Pattern – MVC (Model–View–Controller)

The application follows the **MVC (Model–View–Controller)** architecture pattern, which separates the system into three independent layers:
Models handle data, **Views** present information to users, and **Controllers** manage the application logic and communication between components.
This architecture was selected because it improves maintainability, supports modular development, and ensures a clean separation of responsibilities across the backend services.

1. Controllers (Request Handlers)

Controllers receive incoming HTTP requests, validate input, call the appropriate model functions, and return the output in JSON format.

Example – Order Controller

```
from flask import Blueprint, request, jsonify
from models.order import Order
from models.order_item import OrderItem
from extensions import db

order_bp = Blueprint('order', __name__)

@order_bp.route('/order', methods=['POST'])
def place_order():
  data = request.json
  new_order = Order(user_id=data["user_id"], status="Pending")
  db.session.add(new_order)
  db.session.commit()
  return jsonify({'message': 'Order placed', 'order_id': new_order.id}), 201
```

Explanation:

The controller:

- Receives API calls,
- Performs validation and business logic,
- Communicates with Models to store or retrieve data,
- Sends JSON responses to the client.

This structure keeps the business logic external to the Model and maintains a clean flow from View → Controller → Model.

2. Models (Data Layer)

Models represent database structures and handle all interactions with persistent storage.
They use SQLAlchemy ORM for relational mapping.

Example – MenuItem Model

```
class MenuItem(db.Model):
  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(100))
  price = db.Column(db.Float)
```

Explanation:

- Represents the menu_item table,
- Encapsulates all data-related behavior,
- Ensures consistent database operations,
- Is reusable across controllers.

In the MVC architecture, Controllers never write SQL directly.
Instead, they call Models to handle database operations, ensuring a clean separation between logic and storage.

3. Views (API Responses)

Since this is a backend REST API, the Views are JSON responses returned to the client.

Example – Sample JSON Response

```
{
  "id": 1,
  "username": "nurkyz",
  "orders": []
}
```

Explanation:
In typical web applications, Views consist of rendered HTML templates.
However, in MVC-based REST systems, the View is represented by JSON outputs, which are consumed by the frontend or mobile application.
This approach keeps the backend lightweight, scalable, and reusable.

4. REST API Endpoints (Controller Layer)

Functionality	Endpoint	Method	Controller Function
Register User	/register	POST	auth_controller.register()
Login	/login	POST	auth_controller.login()
View Menu	/menu	GET	menu_controller.get_all()
Add to Cart	/cart/add	POST	cart_controller.add_item()
Place Order	/order	POST	order_controller.place_order()
View User Orders	/orders/<id>	GET	order_controller.get_orders()
Add Menu Item (Admin)	/admin/add_item	POST	admin_controller.add_item()

Explanation:

Each endpoint maps to a specific controller function, ensuring a structured flow of: **Client → Controller → Model → View (JSON).**

5. Database Model (Model Layer)

Examples of core models used in the architecture:

User Model

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True)
    password = db.Column(db.String(255))
```

Order Model

```
class Order(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    status = db.Column(db.String(50))
```

Explanation:

In MVC:

- Models represent database entities,
- Controllers call Model methods instead of writing SQL,
- Views return Model data as JSON.

6. Authentication (Part of Controller Layer)

JWT authentication is implemented inside dedicated controllers:

```
def create_jwt_token(user_id):
  payload = {'user_id': user_id}
  return jwt.encode(payload, SECRET, algorithm="HS256")
```

Explanation:

- Controllers validate user login,
- Models fetch user data from the DB,
- Views return the JWT to the client,
- The architecture keeps each responsibility separated.

7. Error Handling

```
@app.errorhandler(Exception)
def handle_error(e):
  return jsonify({'error': str(e)}), 500
```

Explanation:
Error handling belongs partly to the View layer because it returns formatted error responses.
Controllers raise exceptions, and the error handler transforms them into readable JSON outputs.

8. Logging

```
import logging
logging.basicConfig(filename='app.log', level=logging.INFO)
logging.info("API started")
```

Explanation:
Logging is a cross-cutting architectural feature that helps monitor controller actions and model operations.

9. Testing Framework

```
def test_register_user(client):
  res = client.post('/register', json={'username': 'test', 'password': '123'})
  assert res.status_code == 201
```

Explanation:
Tests cover:

- Controller logic flow,
- Model interactions,
- Returned JSON views.

This ensures that the MVC architecture functions consistently across all layers.

10. Future Improvements

- Add service layer to reduce controller logic
- Implement role-based access middleware
- Support API versioning
- Add caching to reduce model DB load
- Prepare MVC for microservice migration

Implementation Details

Technologies and Programming Languages

The Food Ordering System was implemented using **Python** as the primary programming language. The backend was developed with the **Flask** framework, which was used to build a **RESTful web service** that communicates with the client using JSON over HTTP. The system uses **PostgreSQL** as the relational database for persistent data storage, while **SQLAlchemy ORM** is employed to manage database interactions in an object-oriented manner.

The frontend of the application was implemented as a web interface using **HTML, CSS, and JavaScript / React. Js**, allowing users to browse the menu, manage their cart, and place orders through a browser. **Git and GitHub** were used for version control and collaborative development. The application can be deployed using **Docker and Docker Compose**, enabling consistent setup across different environments.

Backend Implementation

The backend follows the **Model–View–Controller (MVC)** architectural pattern.

- **Models** represent database entities such as users, menu items, orders, and order items.
- **Controllers** are implemented as Flask route handlers and are responsible for processing HTTP requests, executing business logic, and returning JSON responses.
- **Views** are represented by structured JSON responses sent to the frontend client.

Flask routes were implemented for key system functionalities, including user registration and login, menu retrieval, cart management, order placement, and administrative operations such as updating order status and managing inventory.

REST API and Communication

The system exposes a set of RESTful endpoints that follow standard HTTP methods:

- **GET** for retrieving resources (e.g., menu items, order status)
- **POST** for creating new resources (e.g., user registration, placing orders)
- **PUT** for updating existing resources (e.g., order status updates)
- **DELETE** for removing resources where applicable

All communication between the frontend and backend is performed using **JSON-formatted requests and responses**, ensuring platform-independent interaction.

Database Design and Persistence

The database schema includes tables for users, menu items, orders, and order items. SQLAlchemy ORM is used to map Python classes to database tables, simplifying database operations and improving maintainability. Relationships between entities, such as orders and their associated items, are handled using ORM relationships.

Testing and Quality Assurance

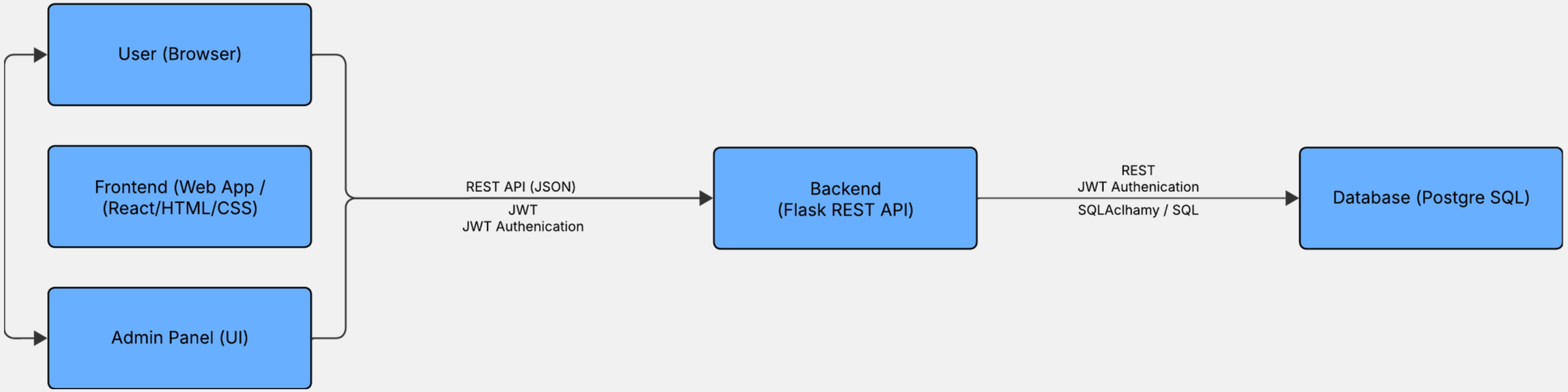
Automated testing was performed using pytest together with Flask's test client. Test cases were written to verify critical system functionalities, including user authentication, cart operations, order creation, and admin-specific actions. This approach helps ensure correctness, reliability, and easier maintenance of the system.

Deployment

The application supports containerized deployment using Docker Compose, which allows the backend service and PostgreSQL database to run as isolated containers. This setup simplifies configuration, improves portability, and ensures consistent behavior across development and production environments.

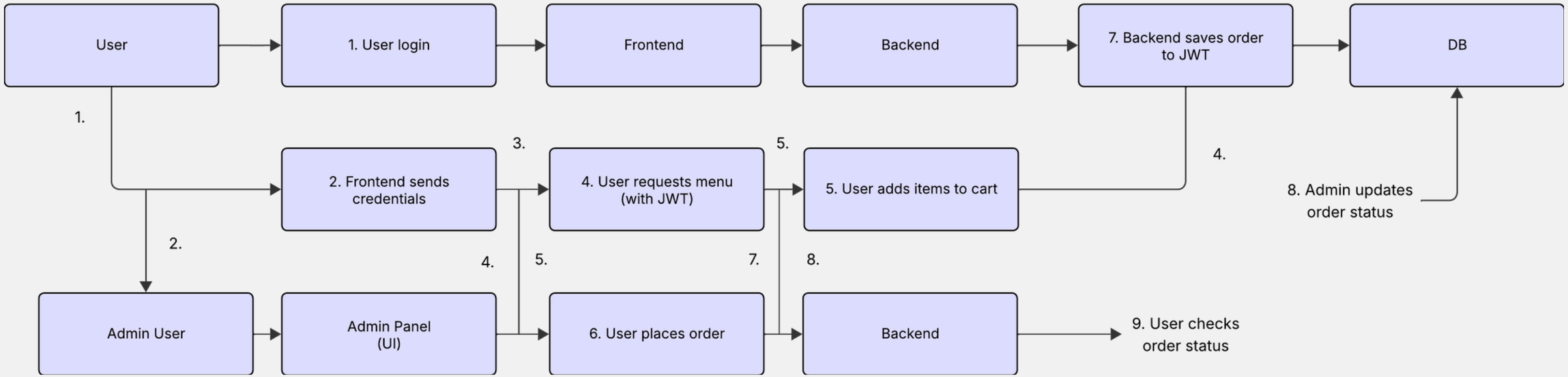
High-Level System Architecture Diagram - Explanation

The high-level system architecture diagram illustrates the technical structure of the Food Ordering System. Users interact with the system through a web browser using the frontend web application or the admin panel. The frontend communicates with the backend via a RESTful API using JSON, while JWT is used for secure authentication and authorization. The backend, implemented using Flask, processes business logic and interacts with the PostgreSQL database through SQLAlchemy to store and retrieve data.



Data Flow Diagram (Order Placement Flow) - Explanation

The data flow diagram presents the main steps involved in the order placement process. It shows how a user logs in, requests menu data, adds items to the cart, and places an order through the frontend interface. The backend validates requests using JWT, stores order data in the database, and allows the admin to update the order status. Finally, the user can retrieve the updated order status from the system.



Conclusions

The main objective of this project was to design and implement a web-based Food Ordering System that allows users to browse a menu, manage a cart, place orders, and track order status through a secure and user-friendly interface. This goal was successfully achieved by developing a RESTful backend using Flask, integrating JWT-based authentication, and storing data persistently in a PostgreSQL database.

The system demonstrates a clear separation of concerns through the use of the MVC architectural pattern, ensuring maintainability and scalability. Core functionalities such as user authentication, order processing, and administrative management were implemented and tested successfully. Automated testing further improved the reliability of the system, while containerized deployment using Docker enhanced portability and consistency across environments.

Future Work

Although the current implementation fulfills the core functional requirements, the system can be further improved and extended in several practical ways.

The first important future step would be the **design and evaluation of a complete UX/UI interface.** This includes creating wireframes and visual designs for both the user-facing interface and the admin panel, followed by usability testing to ensure the system is intuitive and easy to use. Feedback from test users could then be used to refine the interface and improve overall user experience.

- **Enhanced Frontend Functionality:** Completing and polishing the frontend interface to fully cover all backend features, such as order tracking and cart management.
- **Improved Validation and Error Handling:** Adding clearer error messages and input validation on both the client and server sides.
- **Extended Admin Features:** Enhancing the admin panel with better order management tools and basic reporting functions.
- **Performance Optimization:** Optimizing database queries and API responses to improve system responsiveness.
- **Security Improvements:** Strengthening authentication and authorization mechanisms and adding further security checks.

These future improvements focus on making the system more usable, stable, and ready for real-world usage before introducing advanced features.

References

- **Flask Documentation.** *Flask Web Framework*.
- **pytest Documentation.** *pytest – Simple Powerful Testing with Python*.
- **REST API Design Guide.** *RESTful API Design*.
- **Microsoft Visual Studio Code.** *Code Editor Documentation*.
- **YouTube.** *Educational Tutorials on Flask and REST API Development*.
- **Canva.** *Online Design and Presentation Tool*.
- **Microsoft Word.** *Word Processing Software*.