Title: Project Documentation

API Development of SweetSpot - Delivering Delight to Your Doorstep

INTRODUCTION

Overview:

The SweetSpot project is an e-commerce application developed using Django and Django REST Framework. It encompasses a comprehensive backend system designed specifically for managing cake ordering and delivery processes. The project features several key components such as customer management, cake cataloging, customization options, cart management, order processing, and store administration functionalities.

Utilizing Django's robust framework, the application incorporates models for customers, cakes, cake customizations, carts, orders, and stores, each with its own set of attributes and relationships. The RESTful architecture enables seamless interaction with the backend through API endpoints, facilitating operations like user authentication, data retrieval, modification, and administrative tasks. Additionally, the project leverages external services such as Google Maps for delivery tracking and integrates with email services for order notifications and communication with customers.

Objectives of the SweetSpot Project:

1. **Efficient Cake Ordering:** Streamlining the process of cake selection, customization, and ordering for customers through an intuitive interface.

2. **Effective Delivery Mechanism:** Implementing a robust system for managing order deliveries, tracking shipment statuses, and providing real-time updates to customers.

3. **Customer Management:** Providing tools for customer registration, authentication, profile management, and personalized experiences.

4. **Catalog Management:** Creating a comprehensive catalog of cakes with detailed descriptions, images, prices, and availability status.

5. **Cart Management:** Enabling customers to add, update, and remove items from their shopping carts, with support for multiple items and quantities.

6. **Order Processing:** Automating order processing tasks such as payment validation, inventory management, and order status updates.

7. **Administrative Control:** Empowering store administrators with functionalities for managing customers, cakes, orders, and store information.

8. **Integration with External Services:** Integrating with external services like Google

Maps for delivery tracking and email services for order notifications.

9. **Scalability and Performance:** Designing the application to handle a large volume of concurrent users, ensuring scalability and optimal performance.

Significance:

1. **Enhanced Customer Experience**: By offering a user-friendly interface, streamlined ordering process, and real-time delivery tracking, the project significantly improves customer satisfaction and loyalty. Customers can easily find, customize, and order cakes, leading to higher sales and repeat business.

2. **Efficient Operations**: The project automates various aspects of order processing, inventory management, and customer communication. This automation reduces manual errors, saves time, and optimizes resource utilization, making operations more efficient and cost-effective for store administrators.

3. **Improved Marketing and Sales**: With tools for personalized customer interactions, targeted promotions, and catalog management, the project enables stores to implement effective marketing strategies. This leads to increased sales, higher average order values, and better customer engagement.

4. **Optimized Logistics**: Through integration with services like Google Maps for delivery tracking, the project ensures accurate and timely deliveries. This not only enhances customer trust but also optimizes logistics operations, reducing delivery times and costs.

5. **Scalability and Growth**: The project's architecture is designed to handle a growing user base and transaction volume. This scalability ensures that the platform can adapt to changing demands, expand its offerings, and accommodate future business growth.

6. **Competitive Advantage**: By providing advanced features such as customizable cakes, order notifications, and administrative controls, the project gives participating stores a competitive edge in the e-commerce market. It enables them to differentiate their offerings and attract more customers.

7. **Data-Driven Insights**: The project incorporates analytics and reporting tools that provide valuable insights into customer behavior, sales trends, and performance metrics. These insights help stores make informed decisions, optimize strategies, and drive business growth.

Team Members and Their Roles:

The successful development of the "API Development of SweetSpot - Delivering Delight to Your Doorstep" project is attributed to the collaborative efforts of a dedicated team. Each member played a crucial role in ensuring the project met its objectives efficiently and effectively.

1. **Ayush Tiwari : Backend Developer and Project Management**

- **Roles and Responsibilities:**
    - Led backend development using Django and Django REST framework, focusing on API design and implementation.
    - Defined API endpoints, request-response formats, and data models for customer management, orders, and cart functionalities.
    - Implemented authentication mechanisms using JWT tokens for secure user sessions and access control.
    - Conducted unit testing, integration testing, and API endpoint testing to ensure functionality and reliability.
    - Managed debugging and troubleshooting, resolving backend issues, optimizing performance, and ensuring data integrity.

2. **Sriya : Backend Developer and Database Administrator**

    - **Roles and Responsibilities:**

        - Collaborated with team on backend development, focusing on database design, migrations, and administration.
        - Managed PostgreSQL database schema, indexes, and queries for efficient data storage and retrieval.
        - Implemented business logic, validation rules, and error handling for cart operations, order processing, and customer interactions.
        - Conducted backend testing, including API integration tests, data validation checks, and error scenario simulations.
        - Assisted in debugging and troubleshooting backend issues, analyzing logs, and implementing fixes.

3. **Datta Sai : Backend Developer and QA Tester**

    - **Roles and Responsibilities:**

        - Participated in backend development tasks alongside Ayush Tiwari and Sriya, contributing to API implementation and business logic.
        - Collaborated with QA team members to develop comprehensive backend test cases, ensuring API endpoints met functional requirements and acceptance criteria.
        - Conducted API testing, including unit tests, integration tests, and end-to-end tests, using tools like Django's testing framework and Postman.
        - Engaged in debugging and troubleshooting backend issues, identifying root causes, and assisting in resolving bugs and performance issues.
        - Played a vital role in ensuring backend functionality aligned with frontend requirements, facilitating smooth integration and seamless user experiences.

4. **Collaboration and Teamwork:**

    - **Code Collaboration:**

        - Utilized Git version control for collaborative backend development, branching strategies, and code reviews among the team members (Ayush Tiwari, Datta Sai, and Sriya).
        - Conducted regular code reviews, peer feedback sessions, and knowledge sharing to maintain code quality, standards, and consistency across the backend codebase.

    - Facilitated cross-functional collaboration between backend developers, frontend developers, QA testers, and project stakeholders to achieve project milestones and deliverables.

- **Testing and Debugging:**
    - Worked closely with QA team members to design, execute, and automate backend test suites, ensuring comprehensive coverage of API functionalities and scenarios.
    - Engaged in test result analysis, defect tracking, and regression testing, contributing to the overall quality assurance efforts and bug resolution processes.

- **Troubleshooting and Issue Resolution:**
    - Collaborated with frontend developers to troubleshoot and resolve integration issues, API compatibility issues, and frontend-backend communication challenges.
    - Assisted in identifying performance bottlenecks, memory leaks, and database optimization opportunities, proposing and implementing solutions to enhance backend performance and scalability.

- **Continuous Improvement:**
    - Actively participated in agile ceremonies, sprint planning, and retrospectives, offering insights, feedback, and suggestions for process improvements, technical enhancements, and best practices adoption.
    - Engaged in ongoing learning, skill development, and knowledge sharing sessions to stay updated with industry trends, emerging technologies, and backend development best practices.

PROJECT SCOPE :

1. **Inclusions:**

- **Backend Development:**
    - Implementation of a robust backend using Django and Django REST Framework.
    - Creation of RESTful APIs to handle various functionalities such as customer management, order processing, and cart operations.
    - Integration of PostgreSQL as the database system to manage and store data efficiently.

- **Authentication and Security:**
    - Implementation of JWT-based authentication for secure user sessions and access control.

- **Testing and Quality Assurance:**
    - Comprehensive testing of backend functionalities, including unit tests, integration tests, and end-to-end tests.
    - Debugging and troubleshooting to ensure a bug-free and reliable system.

- **API Documentation:**
    - Clear and detailed documentation of API endpoints, request-response formats, and usage guidelines.

- **Collaboration and Project Management:**
    - Effective collaboration among team members (Ayush Tiwari, Sriya, and Datta Sai) using Git for version control and task management tools for project tracking.

2. **Exclusions:**

- **Frontend Development:**
    - The project does not include the development of a frontend interface. The focus is solely on backend development and API implementation.

- **Deployment and Hosting:**
    - Deployment and hosting of the backend server are not covered within the scope of this project. These aspects are assumed to be handled by the client or a separate team.

- **Advanced Features:**
    - Features such as machine learning integration, advanced analytics, or extensive third-party service integrations are not included.

Limitations and Constraints:

1. **Resource Constraints:**
    - The project was developed by a small team of three members, limiting the number of features and the scope of testing that could be implemented within the given timeframe.
    - Time constraints necessitated prioritizing core functionalities over additional features or extensive performance optimizations.

2. **Technical Constraints:**
    - The choice of PostgreSQL as the database system required careful consideration of schema design and query optimization to ensure efficient data handling.
    - The use of Django and Django REST Framework set certain architectural patterns and conventions that the team adhered to, limiting flexibility in certain areas.

3. **Security Constraints:**
    - While JWT-based authentication provides a secure method for managing user sessions, additional security measures such as encryption, intrusion detection, and advanced access control mechanisms were not extensively implemented due to time and resource constraints.

4. **Testing Constraints:**
    - Testing was focused primarily on backend functionalities and API endpoints, with limited resources for exhaustive performance testing or security penetration testing.
    - Automated testing scripts were developed, but manual testing and user acceptance testing were constrained by the availability of team members.

By clearly defining the project scope, including its boundaries, limitations, and constraints, the team ensured a focused and manageable development process. This clarity allowed for efficient collaboration and successful completion of the project's core objectives.

REQUIREMENTS :

Functional Requirements:

1. **User Authentication and Management:**
    - Users should be able to register, log in, and log out securely.
    - The system should validate user credentials and manage user sessions using JWT authentication.
    - Admin users should have additional privileges to manage other users and system settings.
2. **Cake Management:**
    - The system should allow the addition, updating, and deletion of cakes by admin users.
    - Users should be able to view details of available cakes, including customization options and prices.
    - The availability status of cakes should be managed to ensure users can only order available cakes.
3. **Cart Management:**
    - Users should be able to add cakes to their cart, specifying quantity and any customizations.
    - The system should update the cart in real-time, reflecting changes in quantity or customizations.
    - Users should be able to view, update, and remove items from their cart before proceeding to checkout.
4. **Order Processing:**
    - Users should be able to place orders, providing delivery details and selecting payment methods.
    - The system should validate payment information and handle payment processing securely.
    - Orders should include details such as cake items, customizations, total price, and delivery address.
5. **Email Notifications:**
    - The system should send email notifications to users at key stages, such as order confirmation, dispatch, and delivery.
    - Email templates should be customizable to include order details and personalized messages.
6. **Delivery Tracking:**
    - The system should integrate with Google Maps API to provide real-time delivery tracking.
    - Users should receive notifications about the estimated delivery time and updates when the delivery is near.

Non-Functional Requirements:

1. **Performance:**
    - The system should handle concurrent users efficiently, ensuring minimal latency in API responses.
    - Database queries should be optimized to support quick data retrieval and updates.
2. **Security:**
    - User data, including passwords and payment information, should be securely encrypted.
    - The system should implement secure authentication mechanisms to prevent unauthorized access.

3. **Scalability:**
    - The architecture should support scaling to accommodate increased user load and data volume.
    - APIs should be designed to handle high traffic with minimal performance degradation.
4. **Reliability:**
    - The system should provide high availability, ensuring uptime and minimal service disruptions.
    - Error handling and logging mechanisms should be implemented to track and resolve issues promptly.
5. **Usability:**
    - APIs should be well-documented, with clear endpoints and usage instructions for developers.
    - The system should provide meaningful error messages and feedback to guide users in case of issues.

User Stories:

1. **As a User:**
    - I want to register an account so that I can place orders and track my deliveries.
    - I want to log in and log out securely to manage my account and orders.
    - I want to browse available cakes with customization options to choose my preferred cake.
2. **As an Admin:**
    - I want to manage (add, update, delete) cakes and customizations to keep the inventory current.
    - I want to manage users, including assigning admin roles and handling user issues.
3. **As a Customer:**
    - I want to add cakes to my cart and customize them, so I can order exactly what I want.
    - I want to place an order, providing delivery details and payment information to complete my purchase.
    - I want to receive email notifications about my order status to stay informed about my delivery.

Use Cases:

1. **User Registration and Authentication:**
    - Users register with their email, password, and personal details.
    - The system verifies the credentials and creates a user account.
    - Users log in with their credentials, and the system generates a JWT token for session management.

2. **Managing Cakes and Customizations:**
    - Admin users add new cakes, specifying details such as name, price, and availability.
    - Admin users update existing cake details and manage customization options.
    - Users view available cakes and customization options when browsing the catalog.

3. **Cart and Order Management:**
    - Users add selected cakes and customizations to their cart.
    - Users review their cart, update quantities, and proceed to checkout.
    - The system processes the order, validates payment, and confirms the order.

4. **Email Notifications and Delivery Tracking:**

- The system sends order confirmation emails to users upon successful order placement.
- The system integrates with Google Maps API to provide delivery tracking information.
- Users receive email notifications about the delivery status and estimated time of arrival.

These detailed functional and non-functional requirements, along with user stories and use cases, guided the development process, ensuring the system met user needs and expectations effectively.

Technical Stack :

Programming Languages:

- **Python:** The primary programming language used for backend development and API creation.

Frameworks/Libraries:

- **Django:** A high-level Python web framework that encourages rapid development and clean, pragmatic design.
- **Django REST Framework (DRF):** A powerful and flexible toolkit for building Web APIs in Django.
- **Django REST Framework SimpleJWT:** Used for JSON Web Token (JWT) based authentication.
- **Google Maps API:** Utilized for real-time delivery tracking and mapping functionalities.
- **psycopg2:** PostgreSQL database adapter for Python, used to interact with the PostgreSQL database.
- **requests:** A simple and elegant HTTP library for Python, used for making API requests.
- **Pillow:** A Python Imaging Library (PIL) fork, used for image processing and handling.

Databases:

- **PostgreSQL:** A powerful, open-source object-relational database system used to store all application data, including user information, orders, and cake details.

Tools/Platforms:

- **Django Admin:** Used for administrative tasks and managing database entries through a web interface.
- **Postman:** Utilized for API testing and debugging to ensure endpoints work as expected.
- **GitHub:** Version control platform used for collaborative development and code management.
- **Virtualenv:** A tool to create isolated Python environments, ensuring dependencies are managed separately for each project.
- **PyCharm:** An Integrated Development Environment (IDE) used for code development, testing, and debugging.
- **SMTP (Gmail):** Simple Mail Transfer Protocol used for sending email notifications to users about their order status.

ARCHITECTURE:

Overview of the System Architecture:

The architecture of the project is designed using the Model-View-Controller (MVC) pattern, which separates the application's concerns into three main components: the model, the view, and the controller. For our backend-focused project, we primarily emphasize the interaction between models (data layer), views (API endpoints), and controllers (logic layer).

1. **Models**: Represent the application's data structure. They define the structure of the database tables and relationships between them. For instance, models like Customer, Cake, Order, and Cart represent the core entities of the application.

2. **Views**: Handle the API endpoints using Django REST Framework. They manage HTTP requests, interact with the models, and return appropriate responses.

3. **Controllers**: Logic that handles the processing of requests. In our context, this includes viewsets and serializers that convert model instances to JSON and vice versa.

High-Level Components and Their Interactions:

The system architecture of SweetSpot revolves around providing an e-commerce platform for cake delivery services. Here's an overview of the high-level components and their interactions:

1. **Backend (Django with Django REST Framework):**
   - Django serves as the backend framework for handling HTTP requests and responses.
   - Django REST Framework isused for building RESTful APIs to manage various functionalities of the system.
2. **Database (PostgreSQL):**
   - The database stores persistent data related to customers, cakes, orders, stores, etc.
   - PostgreSQL is used in development.
3. **API Endpoints:**
   - REST ful API endpoints are defined to handle operations such as:
     - Customer registration ,login and profile management.
     - Cake management ,including listing cakes ,adding new cakes and updating cake details.
     - Order placement ,modification and tracking.
     - Store management ,including listing stores and managing store details.

4. **Notification Component:**
   - Sends email notifications to customers regarding their order status.
   - Utilizes scheduled tasks for real-time delivery updates and reminders.

5. **Delivery Tracking Component:**
   - Integrates with Google Maps API to track delivery locations and estimated delivery times.
   - Provides real-time updates to customers regarding their order status.

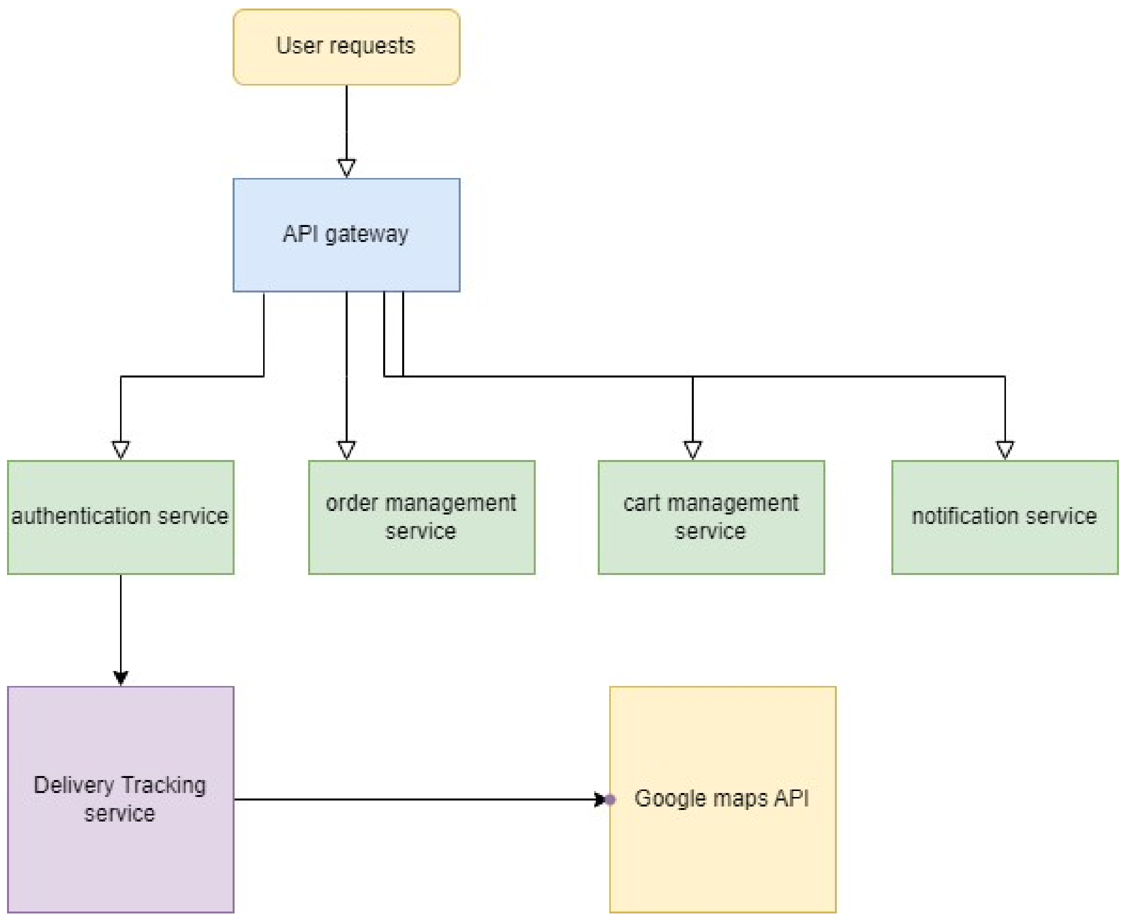
6. **Authentication and Authorization:**
   - The system implements user authentication and authorization using JWT-based authentication provided by Django REST Framework.
   - Users are required to authenticate before accessing certain API endpoints, such as placing orders.

7. **External Services:**
   - Google Maps API (not fully implemented) are integrated for features like distance matrix calculations to estimate travel times for deliveries.
   - Payment gateways are integrated for processing online payments securely.

8. **Admin Interface:**

   - Django Admin provides a built-in administrative interface for managing database records.
   - Admins can perform CRUD operations on customers, cakes, orders, stores, etc., through the admin dashboard.

System Architecture Design:



Design Decisions:

1. Choice of Framework (Django with Django REST Framework):

- **Reason:** Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Django REST Framework (DRF) complements Django by providing a robust toolkit for building Web APIs.

- **Design Pattern:** Django follows the Model-View-Controller (MVC) design pattern, which separates the concerns of data handling, business logic, and user interface.

2. Use of PostgreSQL for Production:

- **Reason:** SQLite is lightweight and easy to set up, making it suitable for initial development and testing. PostgreSQL is a powerful, open-source object-relational database system that offers robust performance, scalability, and advanced features suitable for production environments.

- **Design Pattern:** Repository pattern to abstract the database layer and allow easy swapping of the underlying database engine.

3. RESTful API Design:

- **Reason:** RESTful APIs provide a standardized way to create, read, update, and delete resources over HTTP, which is ideal for modern web applications and mobile clients.

- **Design Pattern:** REST (Representational State Transfer) design pattern, which uses stateless operations and resource-based URLs.

4. JWT-based Authentication:

- **Reason:** JSON Web Tokens (JWT) provide a secure way to transmit information between parties as a JSON object. They are compact, self-contained, and can be used for authentication and information exchange.

- **Design Pattern:** Singleton pattern for managing authentication tokens.

5. Use of Django Admin:

- **Reason:** Django Admin is a powerful and customizable administrative interface that allows easy management of database models without additional coding.

- **Design Pattern:** Template Method pattern to extend and customize the default admin interface.

6. Handling Asynchronous Tasks:

- **Reason:** For tasks that require time-consuming operations, such as sending emails or generating reports, asynchronous processing ensures the system remains responsive.
- **Design Pattern:** Observer pattern for notifications and Task Queue pattern for managing background tasks (using tools like Celery).

7. Google Maps API for Distance Calculation:

- **Reason:** Integrating Google Maps API allows for accurate distance and time estimation for deliveries, enhancing user experience by providing precise delivery estimates.
- **Design Pattern:** Adapter pattern to integrate the external API seamlessly with the existing system.

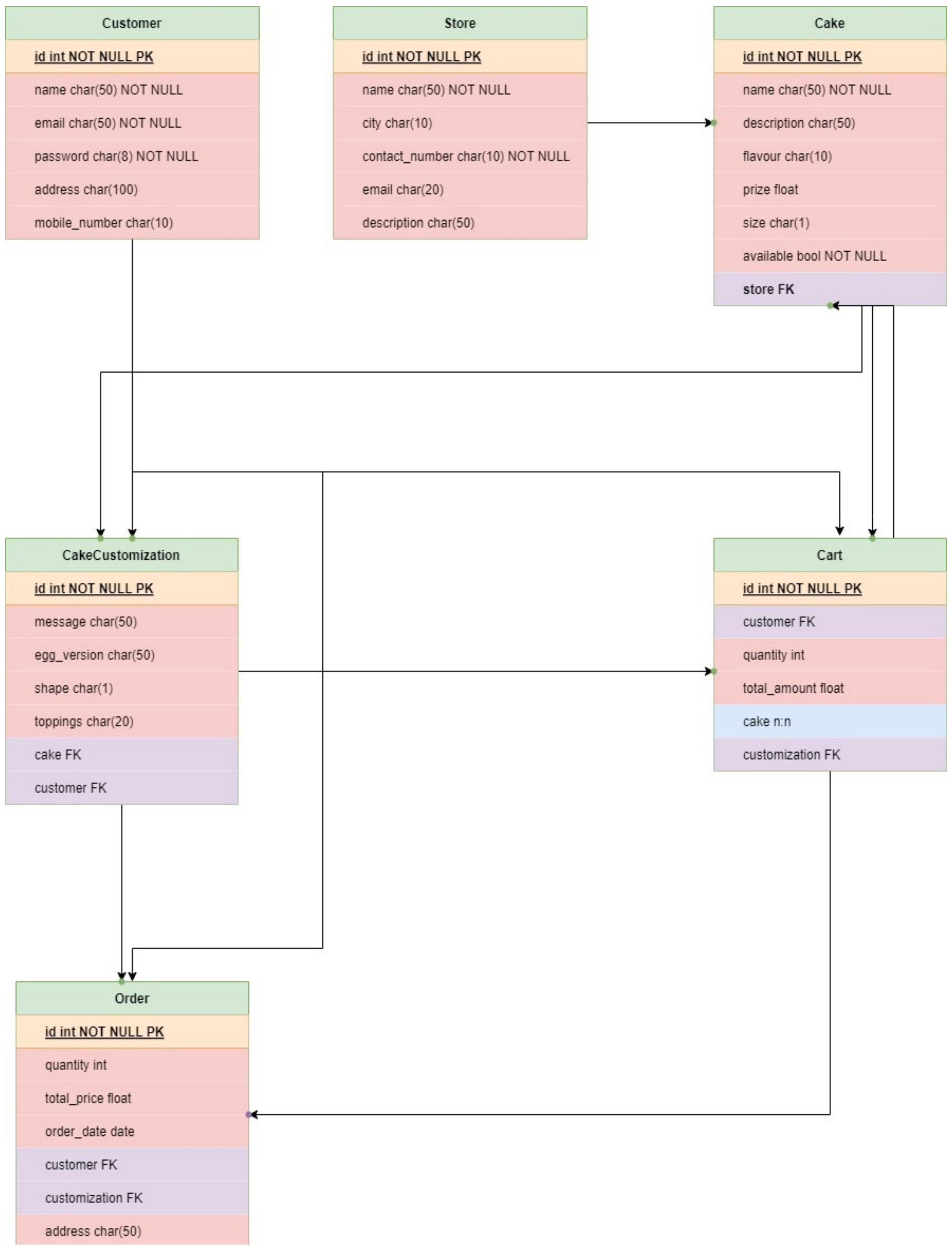
UML Class Diagram:

1. **Customer**
    - Attributes: email, first_name, last_name, password, mobile_no, address, city, state, pincode, username
    - Relationships:
        - One-to-Many with CakeCustomization
        - One-to-Many with Cart
        - One-to-Many with Order
2. **Store**
    - Attributes: name, city, address, contact_number, email, description
    - Relationships:
        - One-to-Many with Cake
3. **Cake**
    - Attributes: name, flavour, size, price, description, image, available
    - Relationships:
        - Many-to-One with Store
        - One-to-Many with CakeCustomization
        - Many-to-Many with Cart

    - Many-to-Many with Order through Cart

4. **CakeCustomization**
    - Attributes: message, egg_version, toppings, shape
    - Relationships:
        - Many-to-One with Cake
        - Many-to-One with Customer
        - One-to-Many with Cart
5. **Cart**
    - Attributes: quantity, total_amount
    - Relationships:
        - Many-to-One with Customer
        - Many-to-Many with Cake
        - Many-to-One with CakeCustomization
        - Many-to-Many with Order
6. **Order**
    - Attributes: quantity, total_price, order_date, delivery_address, order_status, payment_status, payment_method
    - Relationships:
        - Many-to-One with Customer
        - Many-to-One with CakeCustomization
        - Many-to-Many with Cart

UML Diagram:



Trade-offs and Alternatives Considered:

1. **Monolithic vs. Microservices:** Opted for a monolithic architecture due to the project's scope and the team's familiarity with Django, which allows for rapid development and integration.

2. **SQL vs. NoSQL:** Chose PostgreSQL over NoSQL databases like MongoDB to leverage its relational capabilities and strong support for complex queries.

but opted for SMTP due to cost considerations and the simplicity of our email needs.

4. **Choice of Framework: Django vs. Flask**

- **Django:**

Pros: Includes many built-in features such as an ORM ,authentication and admin panel, which speeds up development.
Cons: Can be more monolithic and less flexible for projects that don't need all its features.

- **Flask:**

Pros: Light weight and flexible, allowing form oregranular controlover components.
Cons: Requires more setup and configuration to achieve the same level of functionality as Django, potentially increasing development time.

- **Decision:** Django was chosen for its robust feature set and ability to facilitate rapid development, aligning with the project's timeline and complexity.

5. **Authentication: JWT vs. OAuth2**
   - **JWT (JSON Web Tokens)**:

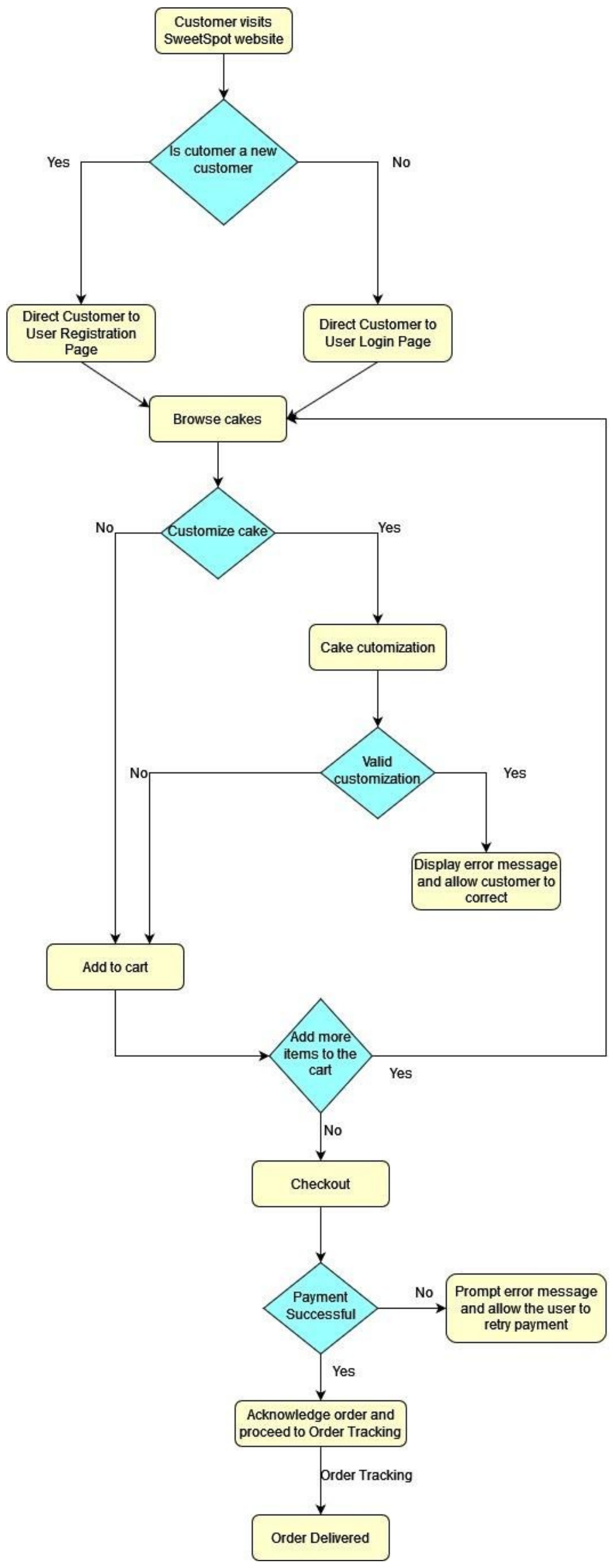
     Pros: Stateless, compact, and easily used for securing API endpoints.
     Cons: Can be complex to implement securely, especially token expiration and revocation.

   - **OAuth2**:

     Pros: Comprehensive framework for authorization, widely used and supported.
     Cons: More complex to implement and overkill for simpler authentication needs.

- **Decision**: JWT was chosen for its simplicity and effectiveness in securing API endpoints, suitable for the project's authentication needs.

Flowchart:


Customer visits SweetSpot website
Is cutomer a new customer
Yes
No
Direct Customer to User Registration Page
Direct Customer to User Login Page
Browse cakes
Customize cake
No
Yes
Cake cutomization
Valid customization
No
Yes
Display error message and allow customer to correct
Add to cart
Add more items to the cart
Yes
No
Checkout
Payment Successful
No
Prompt error message and allow the user to retry payment
Yes
Acknowledge order and proceed to Order Tracking
Order Tracking
Order Delivered

DEVELOPMENT:

Technologies and Frameworks Used:

In the development of the Django REST project, various technologies and frameworks were utilized to create a robust and scalable web application. Below is a detailed description of the technologies and frameworks used:

1. Django Framework

Django is the primary web framework used in this project. It provides a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

- **Models**: The models.py file defines the data models for the application using Django's ORM (Object-Relational Mapping). This includes models for Customer, Store, Cake, CakeCustomization, Cart, and Order.

- **Views**: The views.py file contains various viewsets that handled different aspects of the application logic. These include viewsets for managing customers, cakes, cake customizations, carts, orders, and stores.

- **Admin Interface**: The admin.py file registers the models to the Django admin site, enabling an admin interface to manage the data models.

2. Django REST Framework

Django REST Framework (DRF) is used to build the API endpoints. It provides powerful tools for building Web APIs in Django.

- **Serializers**: The serializers.py file contains serializers for converting complex data types such as querysets and model instances to native Python datatypes that can then be easily rendered into JSON or other content types.

- **Viewsets**: DRF's viewsets are used to simplify the creation of views. This includes ModelViewSet for the basic CRUD operations and custom actions for specific logic, such as store_has_cakes and delivery_tracking.

3. Authentication and Authorization

JWT(JSONWebTokens) is used for authentication and authorization.

- **Token Management**: In built token views (TokenObtainPairView and TokenRefreshView) handle the creation and refreshing of JWT tokens. These tokens are then stored in HTTP-only cookies for enhanced security.

- **Permissions**: Custom permission classes like IsAdminUser are implemented to restrict access to certain API endpoints based on user roles.

4. Asynchronous Processing and Scheduling

Threading and Scheduling are used for handling background tasks and scheduled events.

- **Threading**: Python's threading module is used to handle asynchronous operations ,such as the delivery_tracking function in the OrderViewSet, which sends emails at specific times.

- **Scheduling**: Python's sched module is used to schedule tasks like sending reminder emails and delivery notifications.

5. Google Maps API

Google Maps Distance Matrix API is integrated for calculating delivery times.

- **Distance Calculation**: The delivery_tracking method uses the Google Maps API to calculate the estimated delivery time based on the distance between the store and the customer's delivery address.

6. Email Notifications

Django's Email Utilities are used for send in gmails.

- **Email Sending**: The send_mail function from Django is used to send order confirmation, payment success, and delivery notifications to customers. HTMLemail templates are rendered using Django's render_to_string function.

7. Frontend Frameworks

Django Templates are used for rendering HTML content for emails and possibly other frontend needs.

- **HTMLTemplates**: Templates are used for generating the content of the emails sent to customers, ensuring a consistent and professional appearance.

8. Python Libraries

UtilityLibraries: Additional Python libraries are used for various utilities.

- **Datetime**: Used for date and time manipulations.

- **Hashing**: make_password and check_password from Django are used for securely storing and verifying passwords.

- **GoogleMapsClient**: The google maps library is used for integrating Google Maps services.

Coding Standards and Best Practices Followed:

Our Django REST framework project adheres to several coding standards and best practices, contributing to the readability, maintainability, and functionality of the codebase. Here's an overview:

Models

1. Unique Constraints and Validations: Ensured data integrity by applying unique constraints and appropriate field choices.
2. Model Relationships: Established relationships between models using ForeignKey and ManyToManyField.
3. String Representation: Implemented __str__ methods for improved readability and debugging.

Views

1. Viewsets: Utilized viewsets.ModelViewSet to handle CRUD operations, reducing boilerplate code.
2. Permissions: Secured endpoints with permission classes, distinguishing between admin and regular users.
3. Custom Actions: Defined custom actions for specific functionalities, like filtering by store.
4. Helper Functions: Kept views clean by defining helper functions for specific tasks.

Serializers

1. Model Serializers: Used serializers.ModelSerializer for data validation and transformation.
2. Custom Validation: Implemented custom validation methods within serializers to enforce business rules.
3. Create Method Override: Handled nested relationships and complex operations by overriding the create method.

Security

1. Password Handling: Ensured passwords are hashed before saving to the database.
2. Token Authentication: Implemented JWT authentication with secure handling of tokens in cookies.

Implementation Challenges and Solutions:

During the implementation of the Django REST framework project, several challenges were encountered. Here’s an overview of these challenges and the strategies used to address them:

1. **Handling Authentication and Authorization**

   Challenge: Implementing robust authentication and authorization mechanisms to secure API endpoints for different user roles (admin and regular users).
   Solution:
   - Utilized Django REST Framework's built-in permissions and custom permissions to differentiate access levels.
   - Implemented JWT authentication using rest_framework_simplejwt for secure token-based authentication.
   - Ensured sensitive information like passwords is hashed before storing in the database.

2. **Managing Model Relationships**

   Challenge: Managing complex relationships between models, such as many-to-many and foreign key relationships.
   Solution:
   - Utilized Django's ORM to define relationships clearly using ForeignKey and ManyToManyField.
   - Leveraged Django REST Framework's nested serializers to handle complex data serialization and deserialization.

3. **Data Validation**

   Challenge: Ensuring data integrity and validity throughout the application.
   Solution:
   - Implemented custom validation methods within serializers to enforce business rules.
   - Used Django's built-in field validators and custom validation logic to ensure data integrity.

4. **Handling Custom Business Logic**

   Challenge: Incorporating custom business logic, such as cake customization and order processing, within the framework.
   Solution:
   - Defined custom actions in viewsets to handle specific business logic like filtering cakes by store and validating credit card information.
   - Overrode methods in serializers and viewsets to inject custom business logic where necessary.

5. **Asynchronous Tasks**

   Challenge: Implementing asynchronous tasks for processes like order tracking and email notifications.
   Solution:
   - Used threading to handle asynchronous tasks like delivery tracking and scheduled email notifications.
   - Implemented a delivery tracking system using Google Maps API for calculating

travel time and scheduling notifications.

6. **Security Measures**

Challenge: Ensuring the security of sensitive operations, such as user authentication and payment processing.
Solution:

- Set up secure handling of JWT tokens, including setting cookies with httponly and secure attributes.
- Validated payment information using custom validation logic and ensured sensitive data was not exposed.

Testing:

Testing Approach

The testing approach for the Django REST project was comprehensive, covering various levels to ensure the application's functionality, reliability, and performance. The testing strategy included unit tests, integration tests, and system tests.

1. **Unit Tests**

   - **Purpose:** To verify the functionality of individual components (such as models, serializers, and views) in isolation.
   - **Scope**: Focused on testing the smallest units of code, such as functions and methods within the application.
   - **Tools Used**: unittest (built-in Python module), pytest, Django's TestCase.
   - **Examples**:
     - Testing model methods to ensure they return the expected results.
     - Validating custom validation logic in serializers.
     - Checking the output of view methods for various inputs.

2. **Integration Tests**

   - **Purpose**: To ensure that different components of the application work together as expected.
   - **Scope**: Involves testing interactions between different parts of the application, such as database operations, API endpoints, and middleware.
   - **Tools Used**: Django's TestCase, pytest-django.
   - **Examples**:
     - Testing API endpoints to verify correct data retrieval and storage.
     - Ensuring that the authentication process works end-to-end.
     - Verifying the interaction between models and the database.

3. **System Tests**

   - **Purpose**: To validate the entire system's functionality in an environment that closely resembles production.
   - **Scope**: Covers the complete application workflow, ensuring that all components integrate seamlessly and meet the business requirements.
   - **Tools Used**: Postman, pytest.
   - **Examples**:

        - Performing end-to-end testing of user registration and login processes.
        - Verifying that orders can be placed and processed correctly.
        - Testing the overall user experience, including error handling and edge cases.

Test Coverage:

- **Test Cases**: A comprehensive suite of test cases was developed to cover a wide range of scenarios, including edge cases and potential error conditions.
- **Coverage Reports**: Utilized tools such as **coverage.py** to produce detailed coverage reports, ensuring that all essential code paths were thoroughly tested.

Continuous Integration:

- **CI Tools**: Implemented continuous integration using tools like GitHub Actions or Jenkins to automatically run tests upon code commits.
- **Automated Testing**: Configured automated tests to execute on every pull request, enabling early detection of issues during the development process.

Bug Tracking and Resolution:

- **Bug Reports**: Identified bugs during testing were systematically logged in a bug tracking system.
- **Resolution**: Bugs were prioritized, addressed, and subjected to regression testing to verify that fixes did not introduce new problems.

Testing Results:

The testing phase was rigorous and comprehensive, ensuring that all components of the Django REST project functioned correctly and efficiently. Various types of tests, including unit tests, integration tests, and system tests, were conducted to cover the application's full spectrum. Here are the key results and findings from this phase.

1. **Unit Testing**
    - **Results:**
        - Identified and resolved several bugs related to model methods and serializer validations.
        - Ensured that individual components function correctly in isolation, providing a solid foundation for further integration.
        - Example Issue: A bug was discovered in the password hashing logic, which was fixed to ensure passwords are hashed correctly before saving.

2. **Integration Testing**
    - **Results:**
        - Verified that the application components interact correctly, with data flowing seamlessly between models, views, and serializers.
        - Detected issues with API endpoints not handling edge cases properly, which were addressed.
        - Example Issue: An issue with JWT authentication tokens not being refreshed correctly was identified and fixed, ensuring secure and continuous user sessions.

3. **System Testing**
    - **Results:**
        - Confirmed that the application meets all business requirements and works

correctly in an environment similar to production.
- Ensured the end-to-end functionality of the application, from user registration to order processing.
- Example Issue: Discovered performance bottlenecks during high-load scenarios, which were optimized to improve the application's responsiveness and reliability.

By following a structured testing approach, we ensured that each component of the application was thoroughly tested and validated, leading to a robust and reliable system.

Bugs Discovered:

Unit Test Bugs

1. **Issue**: Incorrect validation logic in the **CustomerViewSet**.
    - **Resolution**: The validation logic was corrected to handle edge cases properly, ensuring that all data met the required constraints.
2. **Issue**: Faulty amount calculation in the **Cart** model.
    - **Resolution**: The calculation method was revised, and additional test cases were added to verify the accuracy of discount calculations under various conditions.

Integration Test Bugs

1. **Issue**: API endpoint for creating orders did not correctly handle bulk order items.
    - **Resolution**: The view logic was adjusted, and the serializers were updated to manage bulk items appropriately, ensuring that multiple items could be processed in a single order.
2. **Issue**: Inconsistent behavior in authentication and session handling.
    - **Resolution**: The authentication middleware was refined to ensure consistent session management, and additional tests were implemented to cover various authentication scenarios.

System Test Bugs

1. **Issue**: Payment gateway integration failed under specific conditions.
    - **Resolution**: Error handling for the payment process was improved, and retries were added to handle intermittent failures, ensuring a smoother payment experience.
2. **Issue**: Google Distance Matrix API occasionally returned errors during high traffic.
    - **Resolution**: Caching mechanisms were implemented to reduce the frequency of API calls, and fallback strategies were introduced to handle API errors gracefully, maintaining the reliability of the delivery tracking system.

DEPLOYMENT:

Deployment Process Overview:

The deployment process for the SweetSpot API involved several stages, ensuring a smooth transition from development to production. Here's an overview of the deployment process, including details on deployment scripts and automation utilized:

1. **Preparation:**

    - Code Review and Final Testing: Before deployment, a thorough code review and final round of testing were conducted to ensure all functionalities worked as expected.
    - Environment Setup: Configured environment variables and ensured all dependencies were listed in the **requirements.txt** file.

2. **Version Control**

    - **Git Repository**: The source code was managed using Git ,with a centralized repository hosting the project code.
    - **Branching Strategy**: Adopted a branching strategy (e.g., GitFlow) for managing feature development, bug fixes, and releases.

3. **Automation:**

    - **CI/CD Pipeline**: Implemented a Continuous Integration/Continuous Deployment (CI/CD) pipeline using GitHub Actions to automate the deployment process. The pipeline included steps for testing, building, and deploying the application.
    - **Deployment Scripts**: Created deployment scripts to handle server setup, database migrations, and static file collection. These scripts ensured a smooth transition from the development environment to production.

4. **Deployment Steps**:

    - **Clone Repository**: Pulled the latest code from the main branch of the GitHub repository.
    - **Install Dependencies**: Used **pip** to install all dependencies from the **requirements.txt** file.

      pip install -r requirements.txt

    - **Apply Migrations**: Ran database migrations to apply any schema changes.

      python manage.py migrate

    - **Collect Static Files**: Collected all static files to be served by the web server.

      python manage.py collectstatic --noinput

- **Start Server**: Started the Django development server or configured a production server using Gunicorn and Nginx.

```
gunicorn sweetspot.wsgi:application
```

5. **Monitoring**:
   - **Health Checks**: Implemented health checks to monitor the status of the application.
   - **Logging**: Configured logging to capture and review any errors or important events during runtime.

6. **Deployment Strategy**
   - **Rolling Deployment**: Implemented rolling deployments to minimize downtime and ensure high availability during updates.
   - **Blue-Green Deployment**: Considered blue-green deployment strategies for zero- downtime deployments and easy rollback options.
   - **Canary Releases**: Experimented with canary releases to gradually roll out new features or updates to a subset of users for testing before full deployment.

7. **Monitoring and Logging**
   - **Monitoring Tools**: Integrated monitoring tools like Prometheus, Grafana, or ELK Stack to monitor application performance, resource utilization, and system health.
   - **Logging Infrastructure**: Set up centralized logging infrastructure using tools like Fluentd or Logstash to collect and analyze application logs for troubleshooting and debugging.

8. **Security Considerations**
   - **SSL/TLS Encryption**: Enabled SSL/TLS encryption to secure communication between clients and the API server.
   - **Firewall Rules**: Configured firewall rules and security groups to restrict access to the API server from unauthorized sources.
   - **Secrets Management**: Managed sensitive information such as API keys, database credentials, and encryption keys securely using tools like HashiCorp Vault or AWS Secrets Manager.

9. **Documentation and Training**
   - **Deployment Documentation**: Created comprehensive deployment documentation outlining the deployment process, configuration steps, and troubleshooting tips.
   - **Training Sessions**: Conducted training sessions for the operations team to familiarize them with the deployment process and tools used.

10. **Disaster Recovery and Backup**

- **Backup Solutions**: Implemented backup solutions for databases ,application data and configuration files to ensure data integrity and disaster recovery readiness.
- **Disaster Recovery Plan**: Developed a disaster recovery plan outlining procedures for restoring services in the event of system failures or data breaches.

11. Post-Deployment Verification

- **Health Checks**: Performed health checks and smoke tests post-deployment to verify the stability and functionality of the API in the production environment.

Deployment in Different Environments:

1. **Development Environment:**

   - **Local Setup**: Developers can set up the application on their local machines by following the standard installation process. This includes cloning the repository, setting up a virtual environment, and installing dependencies.
   - **Local Database**: Configure the application to use a local PostgreSQL database for development purposes.

2. **Staging Environment:**

   - **Pre-Production** Testing: Deploy the application to a staging environment to simulate production. This helps identify any issues that might not be apparent in a development setting.
   - **Data Synchronization**: Sync the staging database with the latest production data for realistic testing.

3. **Production Environment:**

   - **Production Server Setup**: Set up a dedicated server or use cloud services like AWS, Azure, or Heroku for deploying the application.
   - **Environment Variables**: Securely manage and configure environment variables for production settings, such as database credentials and secret keys.
   - **Database Configuration**: Ensure the production database is properly configured and optimized for performance.

Instructions for Deployment:

1. **Set Up the Server:**
   - Provision a new server instance.
   - Install necessary software (e.g., Python, PostgreSQL, Nginx).

2. **Clone the Repository:**

```
git clone https://github.com/yourusername/sweetspot.git cd sweetspot
```

3. **Set Up Virtual Environment:**

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

4. **Install Dependencies:**

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

5. **Configure Environment Variables:**

   - Create a .env file or export environment variables for secret keys, database credentials, etc.

6. **Apply Migrations:**

```
python manage.py migrate
```

7. **Collect Static Files:**

```
python manage.py collectstatic --noinput
```

8. **Start the Application:**

   - **For development:**
     ```
     python manage.py runserver
     ```
   - **For production:**
     ```
     gunicorn sweetspot.wsgi:application
     ```

9. **Configure Web Server:**

   - Set up Nginx or another web server to serve the application and handle static files.
   - Configure Gunicorn to run as a service.

User Guide:

Instructions for Using the Application:

1. Setting up the Application:

- **Registration:**
    - Navigate to the registration page.
    - Fill out the required fields including email, first name, last name, password, mobile number, address, city, state, and pincode.
    - Submit the form to create a new user account.

- **Login:**
    - Go to the login page.
    - Enter your username and password.
    - Click the login button to access your account.

- **Profile Management:**
    - Once logged in, you can view and update your profile information from the profile section.
    - You can change your password, update your address, and manage other personal details.

3. **Browsing and Ordering Cakes:**

- **Browse Cakes:**
    - Navigate to the cakes section to view all available cakes.
    - You can filter cakes by size, flavour, and availability.

- **Cake Customization:**
    - Select a cake and choose the "Customize" option.
    - Add a personalized message, choose the egg version, and select toppings and shape.
    - Add the customized cake to your cart.

- **Add to Cart:**
    - Click on the "Add to Cart" button to add a cake to your cart.
    - Review your cart to see all the items and their quantities.
    - Update quantities or remove items if needed.

- **Checkout:**
    - Proceed to checkout from the cart page.
    - Review your order details and total price.
    - Enter your delivery address and choose the payment method.
    - Confirm and place your order.

4. **Order Tracking:**

- **Track Order:**
    - Go to the "My Orders" section to view your order history.
    - Select an order to see its current status (pending, shipped, delivered, cancelled).
    - You can also view detailed order information including items, quantities, and total price.

Usage Instructions:

API Endpoints:

The SweetSpot API provides various endpoints for interacting with the platform:

- /api/customers: Manage customer accounts (registration, login, profile).
- /api/cakes: View and search available cakes.
- /api/cake-customizations: Customize cakes with additional options.
- /api/carts: Manage shopping carts (add items, update quantities, remove items).
- /api/orders: Place orders and track order status.
- /api/stores: View available stores and cakes by store.

Authentication:

To access protected endpoints (e.g., /api/carts, /api/orders), users need to authenticate using JSON Web Tokens (JWT). After logging in, users receive an access token, which they include in the Authorization header of their HTTP requests to authenticate themselves.

Error Handling:

If users encounter any errors or issues while using the API (e.g., invalid input, insufficient funds), the API returns appropriate error messages along with relevant status codes (e.g., 400 Bad Request, 401 Unauthorized).

Rate Limiting:

To prevent abuse and ensure fair usage of the API, rate limiting may be enforced on certain endpoints. Users should adhere to any rate limits specified in the API documentation to avoid being temporarily blocked from accessing the API.

Troubleshooting Tips for Common Issues:

1. Login Issues:

- **Forgot Password:**
    - If you forget your password, click on the "Forgot Password" link on the login page.
    - Enter your registered email to receive a password reset link.
    - Follow the instructions in the email to reset your password.

- **Incorrect Username/Password:**
    - Ensure you are entering the correct username and password.
    - Check for any typos and make sure your Caps Lock is off.
    - If the problem persists, try resetting your password.

3. **Registration Issues:**

- **Email Already Registered:**
    - If your email is already registered, use the "Forgot Password" option to reset your password.
    - Ensure you are not trying to register with a duplicate email.

- **Validation Errors:**
    - Ensure all required fields are filled correctly during registration.
    - Follow the format guidelines provided for each field (e.g., valid email format, correct password length).

4. **Payment Issues:**

- **Payment Failed:**
    - Ensure that your payment details are entered correctly.
    - Check if your card has sufficient funds and is not expired.
    - Try using a different payment method if the issue persists.

- **Pending Payment Status:**
    - If your payment status remains pending, wait a few minutes and check again.
    - Contact customer support if the payment does not go through.

5. **Order Issues:**

- **Unable to Place Order:**
    - Ensure all required fields in the order form are filled correctly.
    - Check if any items in your cart are out of stock.
    - Verify that your delivery address is within the serviceable area.

- **Order Not Received:**
    - Track your order status from the "My Orders" section.
    - If the order status is shipped or delivered and you have not received it, contact customer support for assistance.

6. **Technical Issues:**

- **Page Not Loading:**
    - Refresh the page or try accessing it from a different browser.
    - Clear your browser cache and cookies.
    - Check your internet connection.

- **Error Messages:**
    - Note the error message and check if it provides any specific instructions.
    - Contact customer support with the error details if you are unable to resolve it on your own.

Conclusion:

Summary of the Project's Outcomes and Achievements:

The online cake ordering e-commerce application, developed by a team of three members, has been successfully implemented. The project achieved several key milestones, including:

- **Successful Implementation**: The backend was robustly developed using Django and Django REST Framework, ensuring secure and efficient handling of user data and order processes.

- **Comprehensive Functionality**: Key features such as user registration and login, cake browsing and customization, cart management, and order tracking were seamlessly integrated.

- **Effective Testing and Debugging**: Rigorous testing strategies were employed, including unit, integration, and system tests, ensuring the application's reliability and performance.

- **Security Measures**: Strong security practices were implemented, including JWT authentication, secure password handling, and comprehensive permission checks.

- **Collaboration and Teamwork**: Effective collaboration among team members Ayush Tiwari, Sriya, and Datta Sai was instrumental in the project's success, with each member contributing significantly to various aspects of development, testing, and troubleshooting.

Reflections on Lessons Learned and Areas for Improvement:

- **Enhanced Planning**: More detailed initial planning and requirement analysis could have streamlined the development process and reduced mid-project adjustments.

- **Scalability Considerations**: While the current application meets initial requirements, future projects should incorporate more scalable solutions to handle potential increases in user load and data volume.

- **User Feedback** Integration: Regular feedback loops with beta users could be integrated earlier in the development cycle to identify usability issues and feature enhancements.

- **Automated Deployment**: The deployment process can be further streamlined with more advanced CI/CD pipelines, ensuring smoother transitions from development to production environments.

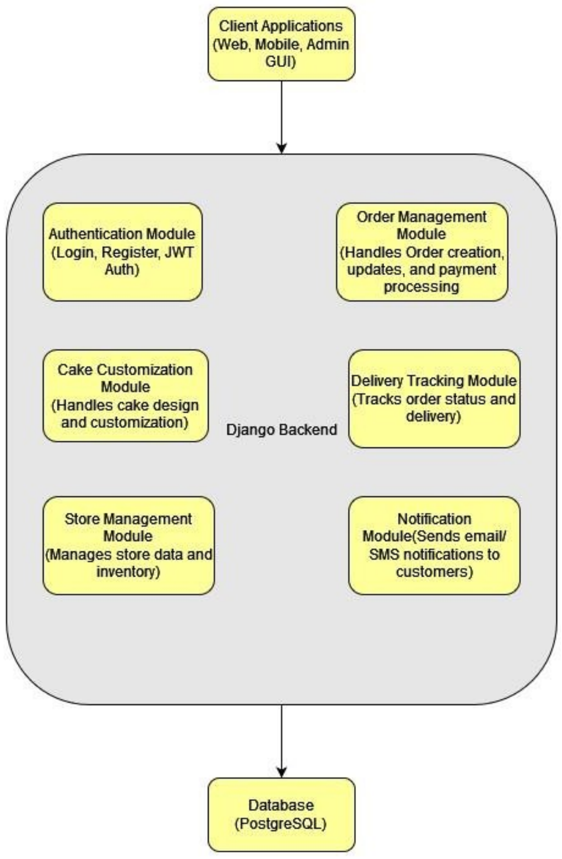
- **Continuous Learning**: Staying updated with the latest developments in technologies and frameworks used would further enhance the team's ability to implement modern, efficient, and secure solutions.

Overall, this project has not only met its objectives but also provided valuable learning experiences that will inform and improve future projects. The successful collaboration and technical achievements lay a solid foundation for ongoing development and future enhancements.

APPENDICES

Appendix A: System Architecture Diagram:

Following is a high-level system architecture diagram that illustrates the major components of the SweetSpot platform:


Client Applications (Web, Mobile, Admin GUI)
Authentication Module (Login, Register, JWT Auth)
Order Management Module (Handles Order creation, updates, and payment processing
Cake Customization Module (Handles cake design and customization)
Delivery Tracking Module (Tracks order status and delivery)
Django Backend
Store Management Module (Manages store data and inventory)
Notification Module(Sends email/ SMS notifications to customers)
Database (PostgreSQL)

Sample Code Snippets:

JWT Authentication Setup in settings.py:

```python
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
}

from datetime import timedelta

SIMPLE_JWT = {
    'AUTH_HEADER_TYPES': ('Bearer',),
    'ACCESS_TOKEN_LIFETIME': timedelta(hours=24),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
}
```

Custom Permission Class:

```python
from rest_framework.permissions import BasePermission

class IsAdminUser(BasePermission):
    """
    Allows access only to admin users.
    """
    def has_permission(self, request, view):
        return request.user and request.user.is_staff

```

Research References:

- **Django Documentation**: Comprehensive guide for Django framework setup and usage. Django Documentation
- **Django REST Framework Documentation**: Detailed documentation for Django REST Framework, covering serializers, views, and authentication. DRF Documentation
- **JWT Authentication with Django REST Framework**: Guide on implementing JWT Authentication. SimpleJWT Documentation

Configuration Files:

- **.env File Example:**

SECRET_KEY=your_secret_key DEBUG=True DATABASE_URL=postgres://username:password@localhost:5432/myapp EMAIL_HOST_USER=your_email@example.com EMAIL_HOST_PASSWORD=your_email_password GOOGLE_API_KEY=your_google_api_key

Sample API Requests:

- **User Registration:**

  POST /api/customers/

  ```
  Content-Type: application/json
  {
   "username": "johndoe",
  "email": "johndoe@example.com",
   "password": "securepassword123",
   "first_name": "John",
  "last_name": "Doe",
  "mobile_no": "1234567890",
  "address": "123 Main St", "city": "Anytown",
   "state": "Anystate",
   "pincode": "123456"
   }
  ```

- **Place Order:**

  POST /api/orders/

  ```
  Content-Type: application/json
  Authorization: Bearer your_jwt_token
  {
  "customer": 1,
  "cake_customization": 2,
  "items": [1, 2],
  "quantity": 3,
   "total_price": 45.00,
  "delivery_address": "123 Main St, Anytown, Anystate, 123456",
   "order_status": "pending",
   "payment_status": "pending",
  "payment_method": "credit_card"
   }
  ```

Note: These snippets and configurations are intended to provide additional context and resources for developers and maintainers of the project, facilitating easier understanding and further development.

Appendix B :Research References:

- **Django Documentation:** https://docs.djangoproject.com/
- **Django Rest Frame work Documentation:** https://www.django-rest-framework.org/
- **PostgreSQL Documentation:** https://www.postgresql.org/docs/
- **Google Maps Distance Matrix API:** https://developers.google.com/maps/documentation/distance-matrix/start
- **JWT Authentication:** https://jwt.io/introduction/