

Infosys Springboard Internship 4.0

Project Report on

“API development of SweetSpot - Delivering Delight to Your Doorstep”

Submitted By

Ishani Bhowmick

Coworkers

Subham Singh

Tushar Gavali

Kolli Varshita

Under the supervision of

S. Lalitha

Maha Lakshmi

Certificate

This is to certify that Ishani Bhowmick with team members Subham Singh, Tushar Gavali, Kolli Varshita at Infosys Springboard Internship 4.0, have submitted a synopsis of the proposed project entitled “API development of SweetSpot - Delivering Delight to Your Doorstep” under the guidance of S. Lalitha and Maha Lakshmi.

The synopsis provides a comprehensive overview of the project, and it is well-structured and well-written, and is clearly presented in a professional manner.

The synopsis has been found satisfactory and is recommended for evaluation.

Date: 23. 05. 2024

Abstract

The goal of this project is to design and implement a computerized e-commerce platform, SweetSpot, dedicated to cake delivery services using Python. SweetSpot offers customers the convenience of ordering customized cakes online and tracking their deliveries in real-time, revolutionizing the cake delivery industry. The platform simplifies the process of ordering customized cakes and ensures timely delivery, enhancing customer satisfaction. By offering features such as online ordering, customization, and real-time delivery tracking, SweetSpot aims to streamline business operations and establish a strong brand presence. The automation of processes like order management and inventory tracking improves operational efficiency, allowing for a seamless user experience. SweetSpot aspires to become a leading player in the market, driven by its innovative approach and commitment to customer delight. Ultimately, the platform seeks to transform the cake delivery industry by providing a comprehensive, user-friendly solution that meets the evolving needs of customers and businesses alike.

Contents

| | |
|-----------------------------------------------------------|----|
| Introduction | 5 |
| Project Scope | 7 |
| Requirements..... | 8 |
| Functional and Non-Functional Requirements | 8 |
| User Stories Behind Development of Sweetspot..... | 9 |
| Technical Stack..... | 11 |
| Architecture/Design..... | 13 |
| Overview Of The System Architecture | 13 |
| UML Diagrams..... | 15 |
| Design Decisions | 21 |
| Trade-offs and Alternative Design Considerations | 21 |
| Development..... | 23 |
| Technologies And Frameworks Used In The Development | 23 |
| Coding Standards and Best Practices Followed | 24 |
| Challenges Encountered and Solutions..... | 25 |
| Testing..... | 26 |
| Testing Approach | 26 |
| Results of the Testing Phase | 27 |
| Deployment..... | 29 |
| Deployment Process: A Step-by-Step Guide..... | 29 |
| Configuring the Environment..... | 29 |
| Automation with Deployment Scripts | 30 |
| Environment-Specific Deployment Instructions | 30 |
| Sweetspot API: Setup and Configuration..... | 31 |
| Usage Instructions..... | 32 |
| API testing screenshots..... | 35 |
| Conclusion..... | 41 |
| Appendices..... | 42 |
| Appendix A | 42 |
| Appendix B | 42 |
| Appendix C | 55 |

Introduction

In today's world of changing lifestyle, our social lives are increasingly fast-paced and digitally connected. Sweetspot perfectly complements this trend, offering a convenient and personalized way to celebrate life's special moments. Busy schedules often leave little time for elaborate cake-making or trips to the bakery. It is an online cake delivery system that eliminates all hassles. With a few clicks, anybody can browse a vast selection of cakes, customize flavours and decorations, and have a beautiful cake delivered right to your doorstep, saving precious time and energy.

Sweetspot goes beyond mere convenience. It empowers customers with personalized choices. They can discover unique offerings from talented local bakers, catering to specific dietary needs and preferences. This not only supports local businesses and artisans but also injects a sense of community into the online cake-ordering experience. Social gatherings have evolved with a focus on smaller, more intimate settings. Sweetspot cake delivery allows everybody to celebrate remotely with friends and family. Surprise a loved one with a virtual celebration and cake delivery, spreading joy even when miles apart. Catering to diverse dietary preferences is no longer a challenge. Sweetspot offers a wider range of allergy-friendly and specialty cakes compared to traditional bakeries. This inclusivity ensures everyone can enjoy a delicious slice at the celebration.

Life throws unexpected moments to celebrate. Online cake delivery empowers you to react with joy. A surprise cake delivery can boost morale. This spontaneity fosters stronger connections and creates lasting memories. This online cake delivery system connects you with talented local bakeries, allowing you to discover hidden gems and support your community. This fosters a sense of connection and ensures your celebrations contribute to the local economy. Sweetspot creates a win-win situation for everyone involved. Consumers enjoy the ease and personalization of online shopping while supporting local businesses and their communities. Local cake makers gain access to a wider customer base, allowing them to showcase their creativity and expand their reach. This cycle fosters a vibrant local economy and strengthens the connection between consumers and passionate local artisans. Sweetspot addresses the growing demand for convenience and personalization in the food industry. Customers can now enjoy the ease of online shopping with the advantage of supporting local businesses. This not only benefits consumers and local economies but also harbours a sense of community by connecting people with the passion and creativity of local cake makers.

This online cake delivery system - Sweetspot caters to the needs of our fast-paced, socially connected generation. It offers convenience, inclusivity, and the chance to celebrate life's moments spontaneously, all while supporting local businesses. So, the next time a celebration arises, anybody can skip the baking stress and embrace the delightful world of online cake delivery of Sweetspot.

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.

| MEMBER NAME | CONTRIBUTION |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ISHANI BHOWMICK | Contributed to backend development tasks, including API design, authentication, and database management; ensured seamless integration between different modules and assisted in identifying and resolving technical issues |

**SUBHAM
SINGH**

Participated in backend development tasks, including database design, model creation, and API integration, conducted testing procedures to ensure the reliability, security, and performance of the application.

**TUSHAR
GAVALI**

Worked closely with team members to understand requirements, implement features, and address challenges effectively; provided ongoing support for the application, addressed user inquiries, and resolved issues in a timely manner.

**KOLLI
VARSHITA**

Supervised the overall project, coordinated team efforts; provided ongoing support for the application, addressed user inquiries, and resolved issues in a timely manner.

Project Scope

The Sweetspot project scope outlines the boundaries of what will be included and excluded when building this delightful online cake delivery platform.

What's Included:

For our customers, Sweetspot offers a seamless experience from start to finish. Users can register and login to browse a curated selection of cakes from nearby stores. They can then customize their dream cake with a variety of flavours, sizes, and toppings to suit their preferences. A shopping cart allows them to easily add, remove, or modify their selections before securely processing payment online. Automated email notifications keep them informed throughout the process, from order confirmation to payment success and delivery updates. Additionally, customers can track their order status for complete peace of mind.

Stores benefit from Sweetspot as well. They can register and manage their profiles, uploading detailed cake information including descriptions, pictures, and pricing. They have the flexibility to manage the customization options available to customers and receive and manage orders through a dedicated dashboard. Updating order status allows them to keep customers informed about the progress of their cakes.

For overall platform management, the admin functionality provides comprehensive user management for both stores and customers. This includes the ability to Create, Read, Update, and Delete (CRUD) information for stores, cakes, and customer orders. Additionally, optional access to overall platform analytics can provide valuable insights for future development.

What's Not Included:

While Sweetspot offers a range of customization options, the initial scope might not include highly intricate cake designs or features requiring extensive back-and-forth communication with customers. Similarly, integrating with a third-party delivery service may be considered for a later phase. Initially, the platform could focus on order confirmation and leaving delivery arrangements to the individual stores. Implementing complex search filters or cake recommendation algorithms could also be deferred for future development iterations.

Limitations and Constraints:

The chosen payment gateway may have limitations on supported regions, transaction fees, or security protocols. To ensure scalability, initial development might prioritize a smaller user base with plans for expansion in the future. Finally, the development team's skills and experience will influence the complexity of features implemented initially.

Requirements

Functional and Non-Functional Requirements

Functional Requirements:

These requirements define the specific functionalities Sweetspot must deliver to fulfil its purpose. They can be categorized based on user types:

Customer:

- **User Registration and Login:** Users should be able to create accounts with secure login credentials.
- **Cake Browsing:** Customers can browse a curated selection of cakes from nearby stores, including filtering by category (e.g., birthday, wedding) or dietary needs (e.g., gluten-free).
- **Cake Customization:** A user-friendly interface allows customers to personalize cakes with various flavours, sizes, toppings, and inscriptions.
- **Shopping Cart Management:** Customers can add, remove, and modify cake selections in a virtual shopping cart before checkout.
- **Secure Online Payment Processing:** Integration with a secure payment gateway ensures safe and convenient online payment for order placement.
- **Automated Email Notifications:** Customers receive automated emails for order confirmation, payment success, and delivery updates.
- **Order Tracking:** A dedicated section allows customers to track the status of their orders (e.g., "placed," "in progress," "delivered").

Store:

- **Store Registration and Profile Management:** Stores can create accounts and manage their profiles with basic information (e.g., location, contact details).
- **Cake Information Upload:** Stores can upload details of their cake offerings, including descriptions, high-quality pictures, and pricing.
- **Customization Options Management:** Stores can manage the customization options available for their cakes (e.g., flavours, sizes) within the platform's limitations.
- **Order Receiving and Management:** Stores receive incoming customer orders through a dedicated dashboard, allowing them to review, accept, and manage them efficiently.
- **Order Status Update:** Stores can update the status of customer orders (e.g., "confirmed," "baking," "out for delivery") to keep customers informed.

Admin:

- **Comprehensive User Management:** Admin has access to manage user accounts for both stores and customers (e.g., create, edit, deactivate).
- **CRUD Operations for Data:** Admin can perform Create, Read, Update, and Delete (CRUD) operations on data related to stores, cakes, and customer orders.
- **Optional Platform Analytics Access:** Admin may have access to overall platform analytics (e.g., order trends, user demographics) for future development decisions.

Non-Functional Requirements:

These requirements define the overall qualities and characteristics of Sweetspot:

- **Performance:** The platform should load pages quickly and respond to user actions efficiently, ensuring a smooth user experience.
- **Scalability:** The system should be able to handle increasing user base and order volume without significant performance degradation.
- **Security:** Secure storage of user data, integration with secure payment gateways, and protection against vulnerabilities are crucial.
- **Availability:** The platform should be highly available with minimal downtime to ensure reliable service.
- **Usability:** An intuitive and user-friendly interface is essential for all user types (customer, store, admin) to navigate the platform easily.
- **Maintainability:** The codebase should be well-structured and documented for efficient maintenance and future development.
- **Reliability:** The platform should function consistently without errors or crashes, ensuring a trustworthy user experience.

Additional Considerations:

- **Mobile Responsiveness:** Optimizing the platform for mobile devices is crucial for user convenience in today's mobile-first world.
- **Accessibility:** The platform should be accessible to users with disabilities, adhering to accessibility guidelines.
- **Compliance:** The platform should comply with relevant data privacy regulations and security standards.

User Stories Behind Development of Sweetspot

The development of Sweetspot, a user-centric online cake delivery platform, was guided by a series of well-defined user stories. These stories served as critical roadmaps, ensuring the platform catered to the core needs of both customers and store owners.

Customer-Centric Functionality:

- **Enhanced Discovery:** Customers expressed a desire to explore a curated selection of cakes from nearby stores. This user story led to the development of location-based search functionalities and partnerships with local bakeries, ensuring customers discover unique and delicious options within their vicinity.
- **Customization for All Occasions:** The ability to personalize cakes with various flavours, sizes, and toppings was a significant user request. This translated into the development of a user-friendly interface for cake customization, empowering customers to create cakes that perfectly match the occasion and any necessary dietary restrictions.
- **Transparency through Tracking:** Maintaining order visibility was paramount for customers. This user story resulted in the creation of a robust order tracking system with automated email notifications, keeping customers informed about the progress of their cake, from order placement to delivery.
- **Seamless Payment Integration:** Secure and familiar online payment processing was a customer priority. This user story necessitated integration with a secure payment gateway, ensuring a smooth and hassle-free checkout experience.

Streamlining Store Operations:

- **Showcasing Expertise:** Store owners emphasized the need to upload detailed cake information, including descriptions, pictures, and pricing. This user story translated into the development of a dedicated store dashboard for managing cake details, allowing store owners to effectively showcase their unique offerings and entice customers.
- **Efficient Order Management:** Managing incoming customer orders efficiently was a crucial requirement for stores. This user story involved building a comprehensive dashboard specifically for managing customer orders, allowing store owners to receive and manage them through a centralized and user-friendly interface.
- **Customer Communication:** Keeping customers updated on order status was important for stores. This user story resulted in the development of a functionality for updating order status (e.g., "in progress," "delivered"), fostering transparency and managing customer expectations.

These user stories served as the foundation for Sweetspot's development. Our team referred to other pan-India companies that work similarly over online cake delivery system. By prioritizing the needs of both customers and store owners, the platform offers a user-friendly and efficient solution for online cake delivery. Future iterations will incorporate additional user stories based on continuous feedback and evolving market demands.

Technical Stack

The technical stack is the foundation of a web application, it consists of programming languages (like Python), frameworks (like Django that help build things faster), databases (like PostgreSQL to store information), and tools (like code editors and testing tools). We use a technical stack to efficiently build web applications. Each part works together: languages write the code, frameworks provide pre-built structures, databases store data, and tools help us write, test, and manage the code. Choosing the right ingredients (languages, frameworks, etc.) ensures a stable, secure, and scalable application that meets the project's needs. Here's a streamlined tech stack to create Sweetspot that empowered its building.

- Programming Languages

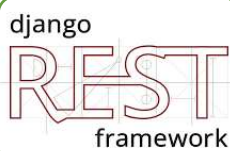


Python (version 3.9): This well-developed, general-purpose, high-level language offers excellent readability and boasts a vast ecosystem of libraries. Python's versatility and well-established web development frameworks make it an ideal choice for building the robust backend of Sweetspot.

- Frameworks/ Libraries used



Django (version 3.2) (<https://docs.djangoproject.com/en/5.0/>) : This high-level web development framework expedites common tasks such as user authentication, database access, and security implementation. Django's pre-built functionalities allow developers to focus on crafting unique features for Sweetspot, streamlining the development process.



Django REST Framework (version 3.14.0) (<https://www.django-rest-framework.org/>) : This powerful extension for Django facilitates the creation of RESTful APIs. These APIs will enable seamless communication between Sweetspot's web application and potential mobile apps or external services, ensuring a flexible and scalable architecture.



Django Channels (for real-time web functionality): This library enables features like live chat or order updates, allowing for a more interactive and dynamic user experience on Sweetspot.



REST Framework Simple JWT (for JWT-based authentication): This package streamlines user login and authorization by implementing JSON Web Tokens (JWT). JWTs offer a secure and efficient way to manage user sessions within Sweetspot.



Google Maps Platform (for distance matrix API): The distance matrix API specifically calculates the distance and travel time between various locations. This functionality can be used to display estimated delivery times for cakes ordered from nearby stores.

- Databases



PostgreSQL: This open-source relational database management system (RDBMS) is renowned for its robust features. PostgreSQL's capabilities in data integrity, scalability, and security make it a trustworthy choice for storing critical information within Sweetspot. This includes user data, intricate cake details, and order information.

- Tools/Platforms:



Visual Studio Code (IDE): This feature-rich Integrated Development Environment (IDE) empowers developers with functionalities like syntax highlighting, code completion, and debugging tools. These features enhance the development experience and promote efficient code creation for Sweetspot.



Postman (API testing tool): This API client simplifies the crucial process of testing and debugging Sweetspot's APIs. Developers can leverage Postman to send requests, meticulously examine responses, and analyze data. This ensures the APIs function as intended, delivering a smooth user experience.

Architecture/Design

Overview Of The System Architecture

Sweetspot's backend architecture is designed for scalability and efficiency, ensuring a seamless experience for both customers and stores. Here's a breakdown of the key components and their interactions:

Backend

- **Django Web Framework:** The heart of the backend, Django handles user authentication, authorization, database interactions, and business logic. It acts as the central processing unit, receiving requests and coordinating responses.
- **Django REST Framework (Optional):** This extension facilitates the creation of well-structured APIs. These APIs serve as the communication channels between the frontend user interface and the backend functionalities.
- **PostgreSQL Database:** This robust database serves as the single source of truth, storing all crucial information for the platform. This includes user data, cake details, order information, and potentially store locations (if using Google Maps integration).

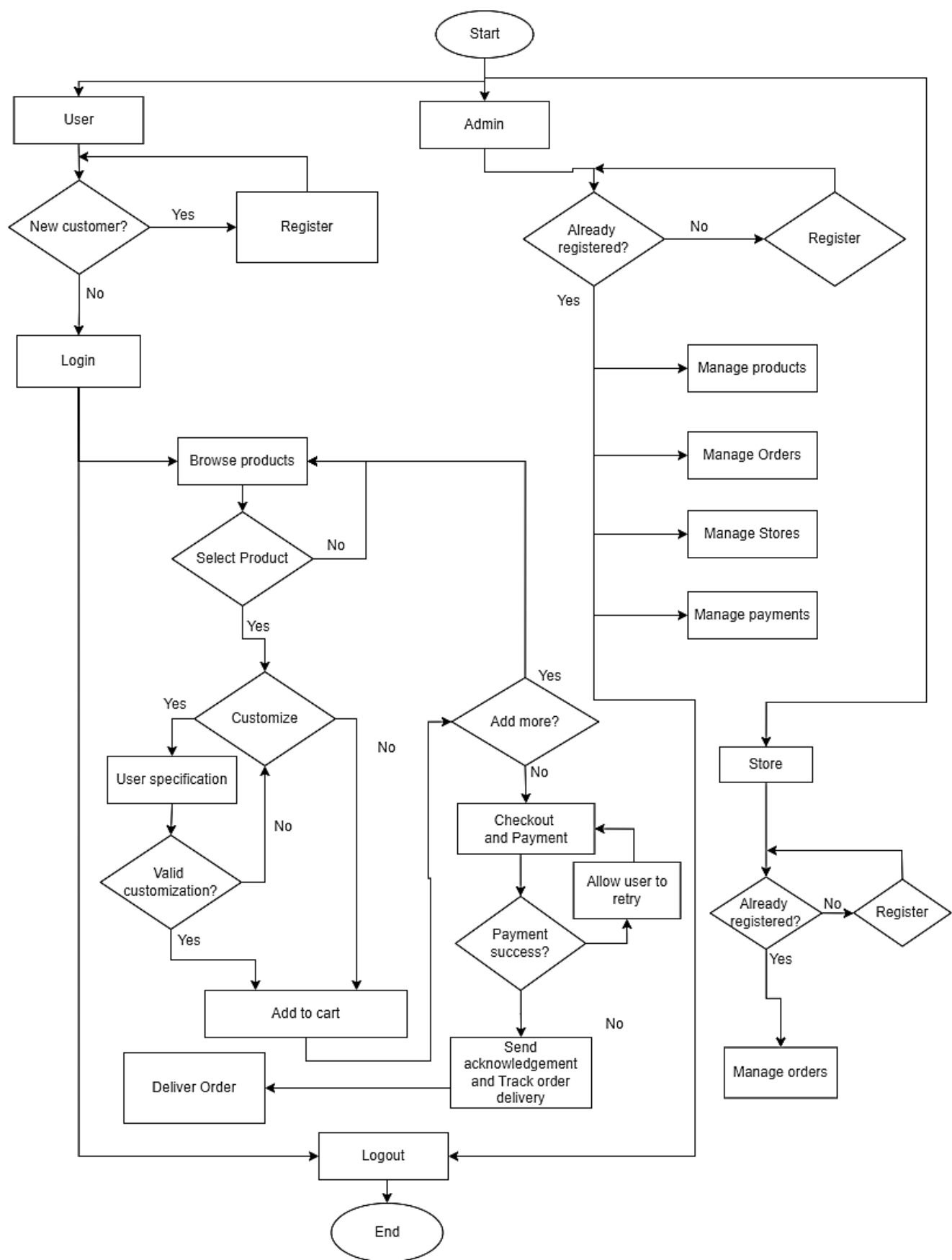
External Services

- **Payment Gateway:** A secure payment gateway PayPal integrates with Sweetspot to process online payments for cake orders. Communication between Django and the payment gateway ensures secure transactions.
- **Email Service:** An SMTP email service allows Sweetspot to send automated email notifications for order confirmation, payment success, and delivery updates. Django triggers emails based on specific events within the system.
- **Google Maps Platform (Optional):** Utilizing the Google Maps Distance Matrix API can calculate delivery distances and estimated times, enhancing transparency for customers. Django interacts with the Google Maps API to retrieve this data.

Communication Flow

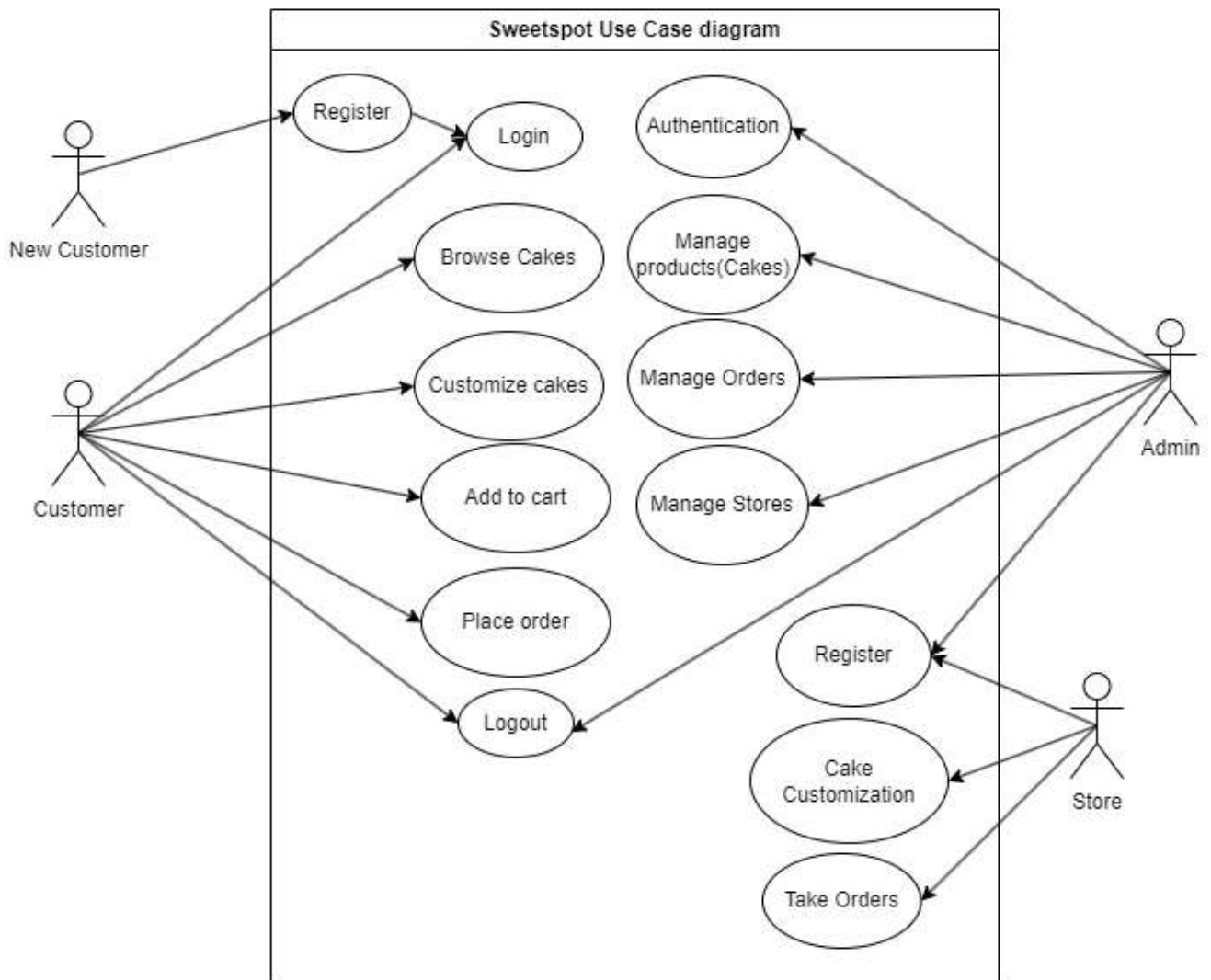
- Step 1. **API Requests:** The UI sends API requests to the Django backend. The frontend user interface sends API requests to dedicated endpoints within the Django framework.
- Step 2. **Django Processing:** Django receives the API requests, utilizes user authentication and authorization to verify access, and interacts with the PostgreSQL database to retrieve or update relevant information. Based on the request type, Django might also interact with external services.
- Step 3. **External Services:** Django communicates with external services like payment gateways or email services to complete specific tasks (e.g., processing payments or sending emails).
- Step 4. **API Response:** Django formulates an API response based on the processing results and sends it back to the frontend user interface.
- Step 5. **Frontend Update:** The UI receives the API response and updates itself accordingly, reflecting changes or displaying relevant information to the user.

Design Flowchart:



UML Diagrams

Use Case UML Diagram



A Use Case Diagram is a type of Unified Modeling Language (UML) diagram that represents the interaction between actors (users or external systems) and a system under consideration to accomplish specific goals. It provides a high-level view of the system's functionality by illustrating the various ways users can interact with it. Here's a breakdown of the elements depicted in the diagram:

Actors:

- **Customer** - The primary user of the application who can register, login, browse cakes, manage orders, and place new orders.
- **Admin** (Optional) - A secondary user who can manage stores, register (if applicable), and potentially manage other administrative tasks within the application.

Use Cases:

- **Register** - This use case allows a new customer to create an account on Sweetspot.

- **Login** - This use case enables existing customers to access their accounts using their email address and password.
- **Browse Cakes** - This use case allows customers to browse through the available cakes offered by different stores.
- **Manage Products (Cakes)** (Optional) - This use case might be associated with the Admin actor and would allow them to manage the cake offerings within the system.
- **Customize Cakes** - This use case allows customers to select a cake, choose a size, flavor, fillings, and any special messages for personalization.
- **Manage Orders** - This use case allows customers to view their order history, track the status of ongoing orders, and potentially reorder cakes from past purchases.
- **Place Order** - This use case encompasses the entire ordering process, including selecting a cake, customization, choosing a delivery time slot, and finalizing the order with secure payment processing.
- **Logout** - This use case allows customers to securely log out of their accounts.

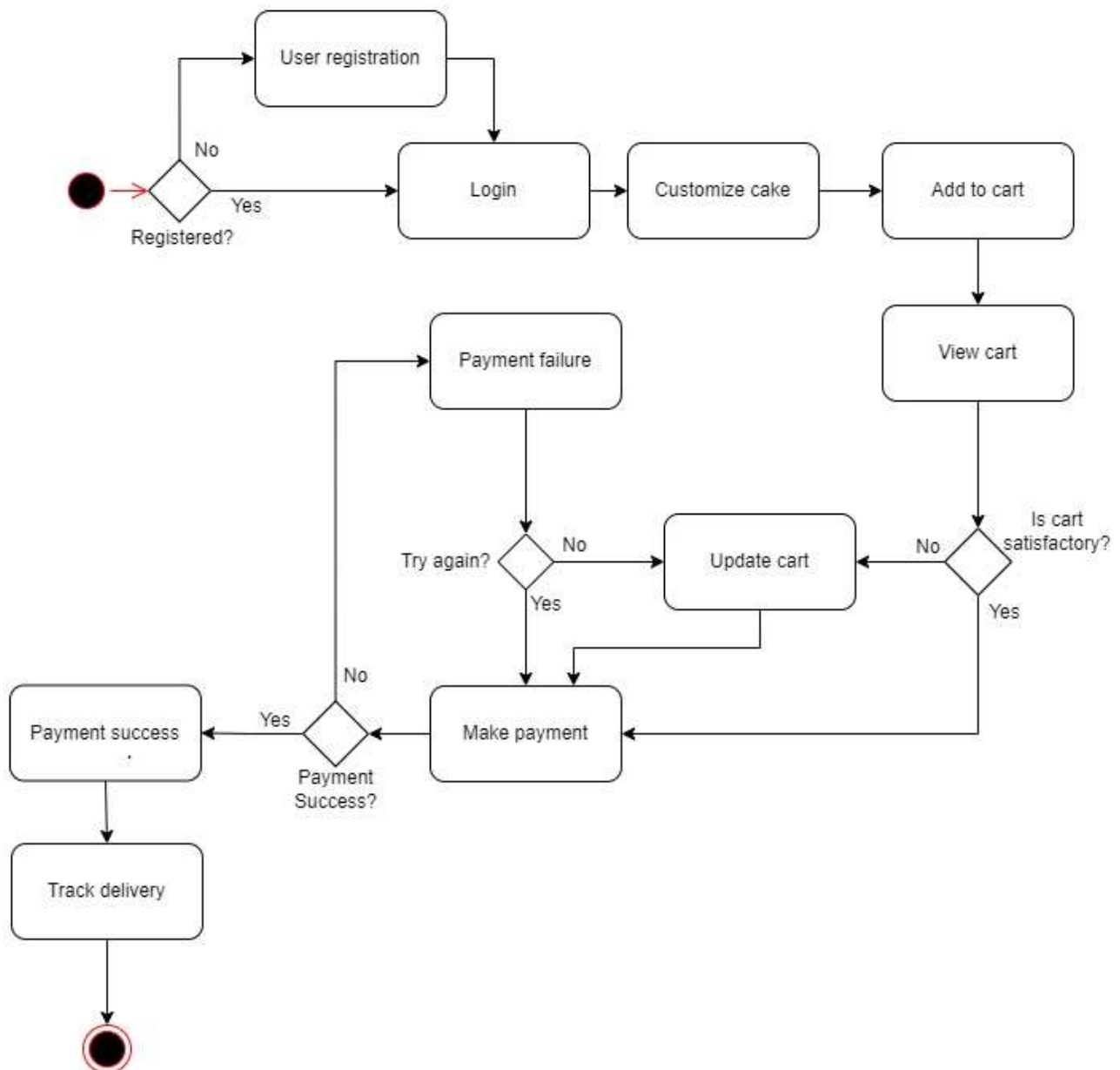
System Entities:

- **Cake** - Represents the various cake offerings available on the platform, including details like description, size, flavors, and prices.
- **Order** - Represents a customer's cake order, containing details like the chosen cake, customization options, delivery information, and order status.
- **Store** (Optional) - This entity might represent the bakery or cake shops that provide the cakes offered on the platform. It could be relevant if the Admin actor has the ability to manage stores or if additional information about the stores is presented to the Customer actor (e.g., location, store details).

Activity UML Diagram

The below Activity UML Diagram focuses on the customer's journey through the checkout process, likely for an online shopping website. While it shares some similarities with the functionalities of Sweetspot, the online cake delivery platform, there are some key differences. Here's a breakdown of the activities and flows in the image you sent:

- **Activities:**
 - User registration (optional)
 - Login (optional)
 - View cart: Customer can review the items in their shopping cart.
 - Update cart: Customer can modify the quantity of items or remove items from the cart.
- **Decision:** "Is cart satisfactory?" Yes/No selection determines if the customer proceeds to checkout or continues shopping.
- **Activities:**
 - Make payment: Customer enters payment information and completes the payment process.
 - Payment success/failure: The system determines the outcome of the payment transaction.
- **Conditional flows:**
 - Based on the payment outcome (success/failure), the user either receives an order confirmation or is prompted to try again.
- **Activity (outside the main flow):** Track delivery: This might be available after successful payment, allowing the customer to track the delivery of their order.



DFD

DFD is the abbreviation for Data Flow Diagram. The flow of data in a system or process is represented by a Data Flow Diagram (DFD). It also gives insight into the inputs and outputs of each entity and the process itself. Data Flow Diagram (DFD) does not have a control flow and no loops or decision rules are present. Specific operations, depending on the type of data, can be explained by a flowchart. It is a graphical tool, useful for communicating with users, managers and other personnel. It is useful for analyzing existing as well as proposed systems.

0-Level DFD:

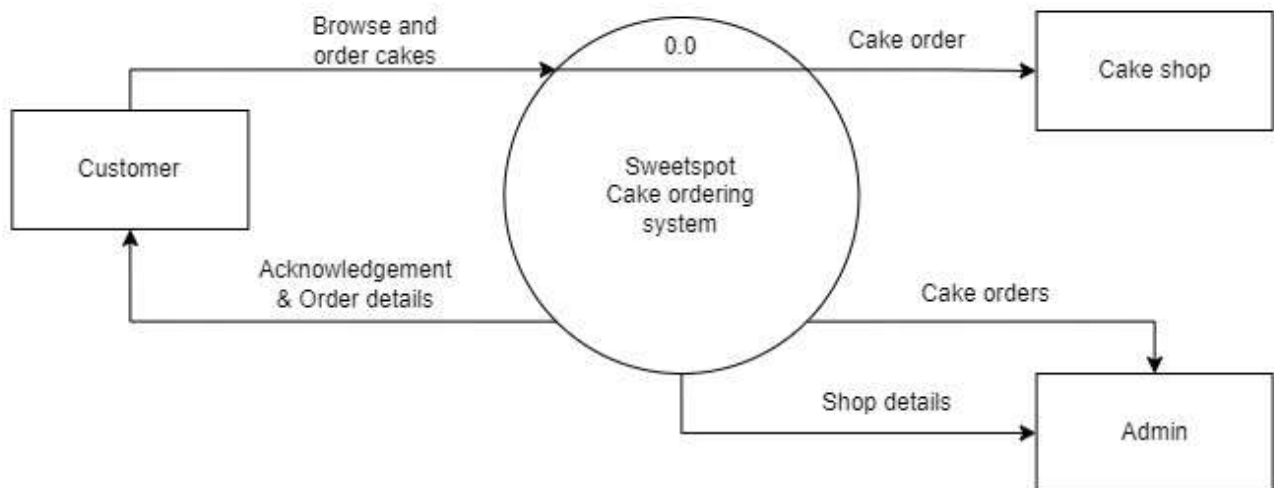
It would depict the main internal processes within the system and how they interact with each other and the external entities (customer and cake shop). Here's a breakdown of a level 0 DFD for Sweetspot:

- **Main Processes:**

- Process Customer Account: Handles customer registration, login, and account management.
- Manage Cake Catalog: Enables adding, updating, and removing cakes offered by the cake shops (potentially through a separate bakery portal).
- Process Cake Orders: Handles receiving and processing customer orders, including customization options and order fulfillment.
- Manage Order Delivery: Facilitates communication with cake shops regarding order fulfillment and potentially tracks delivery status (if integrated with a delivery service).
- Generate Reports (Optional): Generates reports on sales, customer behavior, or other relevant data.

- **Data Flows:**

- Customer interacts with the system to browse cakes, place orders, manage accounts (data flows to Process Customer Account and Process Cake Orders).
- Cake shops interact with the system to manage their cake offerings and receive order information (data flows to Manage Cake Catalog and Process Cake Orders).
- Process Customer Account communicates with Process Cake Orders to fulfill orders placed by customers.
- Process Cake Orders communicates with Manage Cake Catalog to retrieve cake information and potentially Manage Order Delivery to track delivery status.
- Process Orders communicates with the customer to send order confirmations and updates (data flows to customer).
- Process Orders communicates with the cake shop to provide order details (data flows to cake shop).
- Generate Reports (Optional) utilizes data from other processes to create reports.



1-Level DFD

The level 1 DFD expands on the context diagram by illustrating the main internal processes within Sweetspot and how they interact with each other and the external entities (customer and cake shop). Here's a breakdown of the key elements:

External Entities:

- **Customer:** Interacts with the system to browse cakes, place orders, and manage their account.
- **Shop:** Represents the bakery or cake shops that provide the cakes. These shops interact with the system (potentially through a separate web interface) to manage their cake offerings, receive orders, and communicate with customers.

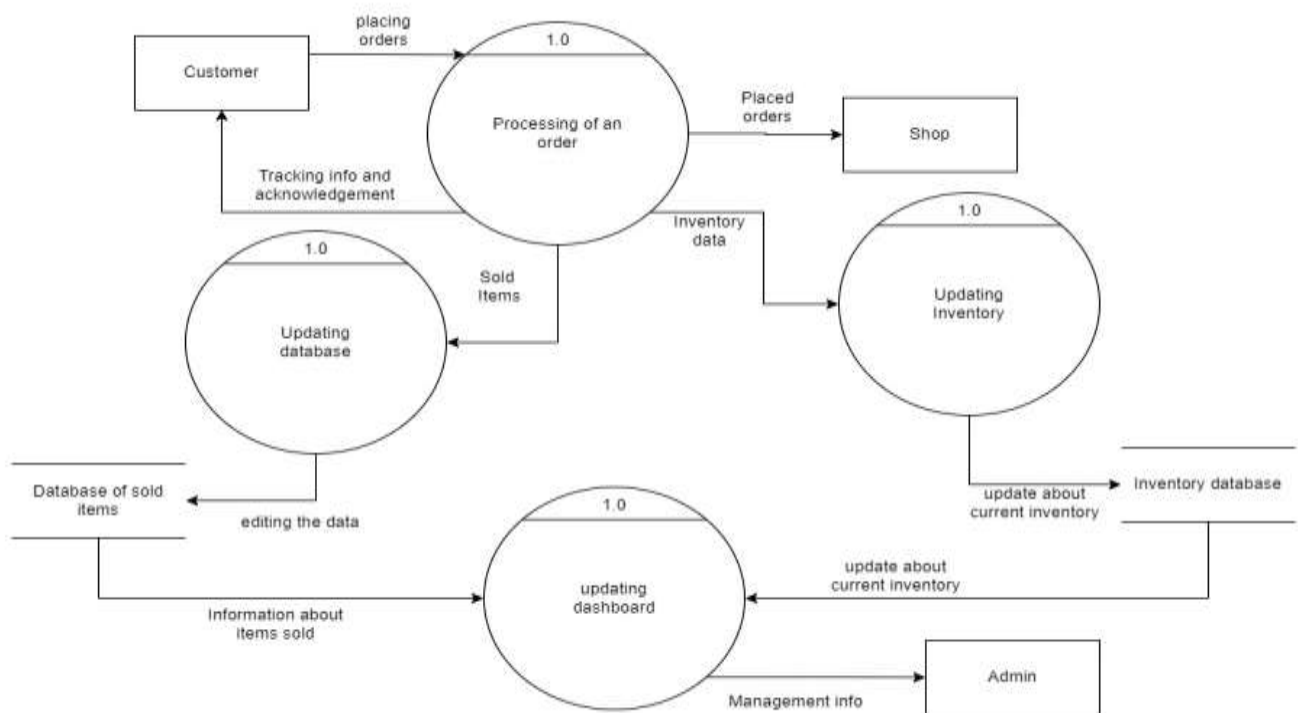
Processes:

1. **Process Customer Request:**
 - Handles customer interactions like browsing cakes, searching for specific cakes, and filtering based on criteria (e.g., category, price).
 - Receives customer input for placing a new order, including cake selection, customization options (size, flavor, fillings, message), and delivery details (time slot, address).
 - Interacts with the Process Account Management subsystem to verify customer login and potentially retrieve customer information for pre-filling order forms.
2. **Process Account Management:**
 - Handles customer account creation, login, and account updates (e.g., profile information, payment methods).
 - Communicates with the customer to provide account-related information or confirmation messages.
3. **Manage Cake Catalog:**
 - Enables adding, updating, and removing cakes offered by the cake shops (potentially through a separate bakery portal).
 - Interacts with the shop to receive cake information (description, images, prices) and updates.
4. **Process Cake Order:**
 - Validates the customer's order details (e.g., selected cake availability, delivery time slot) and communicates any errors or conflicts.
 - Calculates the order total price based on cake selection, customization options, and any applicable taxes or delivery fees.
 - Interacts with the Payment Processing subsystem to securely process customer payments.
 - Creates a new order record in the system database, storing order details, customer information, and cake information.
 - Sends order information to the Manage Order Fulfillment subsystem.
5. **Manage Order Fulfillment:**
 - Communicates with the cake shop to relay the order details (cake, customization, delivery information).
 - Tracks the order status (received, in progress, completed) and updates the customer accordingly.
 - (Optional) Integrates with a delivery service provider to track delivery status and provide updates to the customer.
6. **Generate Reports (Optional):**
 - Creates reports on sales, customer behavior, or other relevant data (not directly involved in order processing but utilizes data from other processes).

Data Flows:

- **Customer interacts with the system:**

- Browsing cakes and searching for specific cakes sends data to Process Customer Request.
- Placing a new order sends data (cake selection, customization options, delivery details) to Process Customer Request.
- Account management interactions (login, create account, update information) send data to Process Account Management.
- **Process Customer Request communicates with other processes:**
 - Retrieves customer information from Process Account Management (during order placement).
 - Sends order details to Process Cake Order for validation and processing.
- **Process Account Management communicates with the customer:**
 - Sends account-related information or confirmation messages (e.g., login success, password reset confirmation).
- **Shop interacts with the system:**
 - Sends cake information (description, images, prices) to Manage Cake Catalog.
- **Manage Cake Catalog communicates with the shop:**
 - Sends updates or requests for cake information.
- **Process Cake Order communicates with other processes:**
 - Verifies inventory with Manage Cake Catalog (optional).
 - Interacts with the Payment Processing subsystem to handle payments.
 - Sends order information to Manage Order Fulfillment.
- **Manage Order Fulfillment communicates with the shop:**
 - Sends order details (cake, customization, delivery information) to the shop.
- **Manage Order Fulfillment communicates with the customer:**
 - Sends order status updates (received, in progress, completed).
- **Generate Reports (Optional):**
 - Utilizes data from other processes (e.g., Process Cake Order, Process Customer Request) to generate reports.



Design Decisions

The design of Sweetspot's backend architecture prioritizes user experience, scalability, and maintainability. These design decisions and potential design patterns contribute to a robust, scalable, and maintainable backend for Sweetspot. The chosen patterns can be implemented as needed based on project complexity and future feature requirements. The focus remains on delivering a secure and efficient platform for online cake delivery, catering to both customers and stores. Here's a breakdown of key design decisions and the implemented design patterns:

Design Decisions

- **Model-View-Controller (MVC) Pattern:** Django inherently follows the MVC pattern, separating the backend logic (Model) from user interaction (View) and API communication (Controller). This promotes code organization, maintainability, and easier testing of individual components.
- **RESTful API Design:** By utilizing Django REST Framework (optional), Sweetspot adheres to RESTful principles for API design. This ensures a standardized approach to data access and manipulation, facilitating communication with the frontend and potential future mobile apps.
- **Database Schema Design:** The PostgreSQL database schema is carefully designed to efficiently store and retrieve cake details, order information, and user data. Normalization techniques help minimize data redundancy and ensure data integrity.
- **Scalable Infrastructure:** The backend is designed to be deployed on a scalable cloud platform, allowing for future growth and handling increased user traffic and order volume without significant performance degradation.

Design Patterns

- **Repository Pattern (Optional):** This pattern might be implemented to encapsulate logic for interacting with the PostgreSQL database. A dedicated repository class can handle data access concerns, separating them from the core business logic within Django models.
- **Factory Pattern (Optional):** For complex cake object creation with various customization options, the factory pattern can be employed. This pattern allows for centralized creation logic, promoting code reuse and maintainability.
- **Decorator Pattern (Optional):** Advanced features like user permissions or order tracking could utilize the decorator pattern. This pattern allows for dynamically adding functionalities to existing objects without modifying their core behavior.

Trade-offs and Alternative Design Considerations

These trade-offs prioritize efficient development for a Minimum Viable Product (MVP) of Sweetspot. The initial focus is on core functionalities and a user-friendly platform. Future iterations can incorporate more features based on user feedback and business needs. The chosen technologies can also be revisited and upgraded as the platform evolves.

API Development

- **Trade-off:** Django REST Framework vs. Custom API development
- **Decision:** For the initial set of functionalities, custom API development might be considered to save time on setting up Django REST Framework.

- **Future Consideration:** As the API evolves and requires more complex functionalities, transitioning to Django REST Framework can streamline development and maintenance.

Google Maps Integration

- **Trade-off:** Google Maps Integration vs. Static Delivery Estimates
- **Decision:** Initially, static estimates based on average delivery times could be used. This avoids the setup and potential costs associated with Google Maps.
- **Future Consideration:** Integrating Google Maps can provide dynamic delivery estimates, enhancing user experience. This can be added in a later phase.

Authentication

- **Trade-off:** JWT-based authentication vs. Alternatives (session-based, social login)
- **Decision:** While JWT-based authentication with Django REST Framework Simple JWT is included, alternatives can be explored based on evolving needs.
- **Future Consideration:** Session-based authentication might be simpler for less complex user management. Social login options can be integrated for wider user reach.

Development

Technologies And Frameworks Used In The Development

By leveraging the following technologies, adhering to coding standards, and addressing challenges methodically, the development of the Sweetspot project was able to progress smoothly, resulting in a robust and maintainable online cake delivery system. The project's structure and practices ensure scalability, security, and ease of future development, setting a strong foundation for continued growth and enhancement.

Programming Language: Python (version 3.9)

- Reason for Choice: Python is known for its simplicity and readability, which accelerates the development process. Its extensive standard library and large ecosystem of third-party packages make it highly versatile for web development.
- Advantages: Python's dynamic typing and concise syntax contribute to faster development times. Additionally, its support for multiple programming paradigms (procedural, object-oriented, and functional) provides flexibility in solving various programming problems.

Web Framework: Django (version 3.2)

- Reason for Choice: Django is a high-level web framework that encourages rapid development and clean, pragmatic design. It's particularly well-suited for applications that require robust database interaction and user authentication.
- Features: Django comes with built-in tools for ORM, authentication, admin interface, routing, and templating. It also adheres to the DRY (Don't Repeat Yourself) principle, which reduces redundancy and increases code maintainability.
- Scalability: Django's architecture supports scaling both horizontally and vertically, making it a good choice for growing applications.

API Framework: Django REST Framework (DRF) (version 3.14.0)

- Reason for Choice: DRF simplifies the creation of RESTful APIs by providing a flexible and powerful toolkit for building Web APIs. It integrates seamlessly with Django, allowing the reuse of models and views.
- Features: DRF offers serialization, authentication, and permission classes, which help in managing and securing API endpoints effectively. It also provides an interactive API browser, which enhances the developer experience during testing and debugging.

Database: PostgreSQL

- Reason for Choice: PostgreSQL is a powerful, open-source relational database that offers advanced features like full-text search, JSON support, and complex querying capabilities.
- Advantages: PostgreSQL's ACID compliance ensures data integrity, and its extensibility allows the use of custom functions and data types. It's well-suited for handling the complex queries and transactional operations required by the Sweetspot application.

Tools/Platforms:

- Visual Studio Code (IDE): Chosen for its robust feature set, including debugging, integrated terminal, and support for a wide range of extensions. It enhances productivity with features like IntelliSense, code navigation, and version control integration.
- Postman: Utilized for API testing and debugging. Postman's ability to create and run collections of requests allows thorough testing of API endpoints, ensuring they function correctly and efficiently.

Coding Standards and Best Practices Followed

The development team adopted a pragmatic approach by prioritizing a Minimum Viable Product (MVP). This focused version of Sweetspot included core functionalities like user registration, cake browsing and ordering, and basic order management. This strategy enabled a faster launch, allowing the team to gather valuable user feedback early on. With real-world user data in hand, the team could then prioritize future features and development efforts to create a product that truly resonates with the target audience. The selection of technologies was a crucial step in ensuring Sweetspot's scalability and long-term success. Python, with its well-established web development libraries and focus on readability, provided a solid foundation for the backend. On top of this, Django's pre-built functionalities like user authentication and admin panels streamlined development, allowing the team to focus on crafting unique features for Sweetspot. PostgreSQL, a robust and secure open-source database, was chosen to store critical user and order information. By opting for these well-regarded technologies, the team laid a groundwork that could handle increasing user traffic and order volume as Sweetspot gained traction.

Code Organization and Structure

- **Project Layout:** Followed Django's recommended project layout with separate apps for different functionalities (e.g., customers, stores, orders, cakes). This modular approach enhances code maintainability and scalability.
- **MVT Pattern:** Adhered to the Model-View-Template (MVT) pattern to ensure a clear separation of concerns, making the codebase easier to understand and manage.

Version Control

- **Git:** Used Git for version control, enabling collaborative development and version tracking. Followed a branching strategy (e.g., feature branches, development branch, main branch) to organize and manage changes.
- **Commit Messages:** Maintained clear and descriptive commit messages to document the purpose and context of changes, facilitating easier navigation and understanding of the project's history.

Testing

- **Unit Testing:** Implemented unit tests using Django's built-in testing framework to ensure individual components function as expected. Focused on testing models, views, and serializers.
- **Integration Testing:** Used Postman to create collections of API tests, validating that different parts of the application work together seamlessly.
- **Continuous Integration:** Integrated CI tools (e.g., GitHub Actions, Travis CI) to automate the testing process, ensuring that code changes do not introduce regressions.

Security Best Practices

- **Authentication and Authorization:** Leveraged Django's built-in mechanisms for user authentication and DRF's permission classes to secure API endpoints.
- **Data Protection:** Used environment variables to manage sensitive information such as database credentials and secret keys, preventing them from being hard-coded in the source code.
- **Preventive Measures:** Implemented measures to prevent common security vulnerabilities like SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).

Documentation

- **API Documentation:** Used tools like Swagger or DRF's built-in documentation to generate and maintain up-to-date API documentation. This made it easier for developers to understand and interact with the APIs.

- **Code Documentation:** Included comprehensive docstrings and comments to explain complex logic and functionality, aiding future developers and collaborators.

Challenges Encountered and Solutions

Database Migrations

- **Challenge:** Ensuring smooth and error-free database migrations, especially when making significant schema changes.
- **Solution:** Carefully planned and tested migrations in a development environment before applying them to production. Used Django's migration tools to manage changes incrementally and rollback strategies to handle potential issues.

API Versioning

- **Challenge:** Managing changes to the API without breaking existing clients.
- **Solution:** Implemented API versioning using URL patterns (e.g., /api/v1/) to allow multiple versions of the API to coexist. This approach provided clients with the flexibility to migrate to newer versions at their own pace.

Handling Customizations

- **Challenge:** Implementing a flexible customization system for cakes that could handle various customer requests.
- **Solution:** Designed a robust data model with a Customization entity linked to cakes, allowing dynamic handling of different customization options. Used serializers to process and validate customization data efficiently, ensuring it integrates seamlessly with the order processing system.

Email Notifications

- **Challenge:** Ensuring reliable email delivery for payment success and delivery tracking notifications.
- **Solution:** Integrated Django's email backend with a reliable email service provider of SMTP services. Implemented asynchronous email sending using Celery, improving performance and reliability by offloading email tasks from the main application thread.

Performance Optimization

- **Challenge:** Optimizing the performance of the application to handle high traffic and large volumes of data.
- **Solution:** Employed caching strategies using Django's caching framework (e.g., Memcached, Redis) to reduce load times. Optimized database queries by indexing key fields and using Django's select-related and prefetch-related methods to minimize the number of queries. Conducted load testing and profiling using tools like Apache JMeter and Django Debug Toolbar to identify and resolve bottlenecks.

Testing

Testing Approach

The development of Sweetspot wasn't just about building features; it was also about ensuring a delightful and bug-free user experience. To achieve this, a multi-layered testing approach was implemented, encompassing unit tests, integration tests, and system tests. This layered approach ensured that Sweetspot was built on a solid foundation of well-functioning individual components, that these components worked together seamlessly, and that the overall system functioned as expected from a user's perspective.

Unit Testing: Building Blocks of Reliability

Individual functions, modules, and classes within the Sweetspot codebase were subjected to rigorous unit testing. These unit tests served as the foundation of the testing pyramid, verifying the functionality of each building block in isolation. Testing frameworks like Django's built-in testing framework or additional options like pytest were utilized to write comprehensive unit tests that covered various scenarios and edge cases.

- **Purpose:** To validate the functionality of individual components in isolation, such as models, views, serializers, and utilities.
- **Tools Used:** Django's built-in testing framework (`django.test`), along with Python's `unittest` module.
- **Scope:**
 - Models: Tested the integrity and relationships of the database schema.
 - Views: Verified that each view function or class-based view returned the expected HTTP status codes and responses.
 - Serializers: Ensured that data serialization and deserialization processes worked correctly.
 - Utilities: Checked any helper functions or custom logic for correctness.

Integration Testing: Bridging the Gaps

After the individual components passed their unit tests, integration testing ensured they functioned harmoniously when working together. This phase focused on verifying how different modules interacted with each other, particularly how data flowed between them. For instance, integration tests might check if user input from the View layer is correctly processed by the Model layer and saved to the PostgreSQL database.

- **Purpose:** To test the interaction between different components and ensure they work together as expected.
- **Tools Used:** Django's testing framework, Postman for API testing, and pytest for more advanced testing features.
- **Scope:**
 - API Endpoints: Tested the end-to-end functionality of REST API endpoints, including authentication, CRUD operations, and data flow between models.
 - User Scenarios: Simulated real-world user scenarios, such as placing an order, customizing a cake, and tracking delivery.

System Testing: Simulating the Real World

System testing served as the final checkpoint, simulating real-world user interactions with Sweetspot. These tests assessed the overall functionality of the system from a user's perspective. Manual testing

played a crucial role in this phase, with testers mimicking user actions like registering, searching for cakes, placing orders, and processing payments. Additionally, automated system tests might be implemented to simulate user workflows and identify any potential issues.

- **Purpose:** To validate the system as a whole and ensure that the entire application meets the requirements.
- **Tools Used:** Selenium for browser automation and UI testing, and manual testing to verify the application's functionality from an end-user perspective.
- **Scope:**
 - End-to-End Scenarios: Tested complete workflows, such as user registration, login, adding cakes to the cart, placing orders, making payments, and tracking delivery.
 - UI/UX: Verified the user interface for usability and responsiveness across different devices and browsers.

Results of the Testing Phase

The testing phase revealed several bugs and issues, which were systematically addressed and resolved. The thorough testing approach, encompassing unit, integration, and system tests, ensured the Sweetspot application was robust, reliable, and user-friendly. All identified bugs and issues were resolved, leading to a stable release. Regular testing and continuous integration practices were crucial in maintaining the quality of the application and ensuring a smooth user experience. The comprehensive testing approach successfully unearthed a variety of bugs and potential issues within Sweetspot's codebase. These issues ranged from minor UI glitches to more critical errors in data processing or order management. The development team meticulously documented and addressed each identified bug, ensuring a smooth and functional user experience. Below is a summary of the key findings and resolutions:

Bug: Incorrect Order Total Calculation

- **Issue:** The total price for orders was sometimes calculated incorrectly due to floating-point arithmetic errors.
- **Resolution:** Implemented Django's DecimalField for price calculations and ensured that all monetary values were handled with the decimal module to maintain precision.

Bug: API Endpoint Authentication

- **Issue:** Some API endpoints were accessible without proper authentication, posing a security risk.
- **Resolution:** Added appropriate authentication and permission classes to all sensitive endpoints using DRF's IsAuthenticated and custom permission classes where necessary.

Issue: Slow Response Times for Complex Queries

- **Issue:** Some views with complex database queries had slow response times, affecting the user experience.
- **Resolution:** Optimized database queries by using select_related and prefetch_related to reduce the number of queries. Implemented caching for frequently accessed data.

Bug: Email Delivery Failures

- **Issue:** Emails for payment success and delivery tracking were not being reliably sent.
- **Resolution:** Integrated with a reliable email service provider and used Celery with Redis for asynchronous email processing, ensuring emails are sent reliably without affecting the main application flow.

Bug: Order Status Not Updating

- **Issue:** Order status was not updating correctly when payment was confirmed or when the order was delivered.

- **Resolution:** Fixed the logic in the order processing workflow to ensure the order status updates correctly based on the payment and delivery events.

Benefits of a Multi-Layered Approach:

- **Early Detection of Issues:** By implementing unit tests early in the development process, the team could identify and fix errors within individual components before they propagated to larger parts of the system. This saved time and effort compared to identifying issues later in the development cycle.
- **Improved Code Quality:** The focus on unit testing encouraged developers to write clean, modular, and well-documented code. Code that is easier to test is often easier to understand, maintain, and reuse in the future.
- **Enhanced System Stability:** Integration and system testing ensured that different parts of Sweetspot worked together seamlessly, minimizing the risk of unexpected crashes or malfunctions in a production environment.

Deployment

Deployment Process: A Step-by-Step Guide

This document outlines the deployment process for Sweetspot, the delightful online cake delivery platform. We'll delve into the steps involved, the automation employed to streamline deployment, and the environment-specific instructions to ensure a seamless transition from development to production.

1. Code Preparation:

- The development team merges all code changes into the master branch.
- Unit tests and integration tests are re-run to confirm everything functions as expected.
- Static code analysis tools can be employed to identify potential issues or areas for improvement.

2. Building for Deployment:

- Deployment scripts written in languages like Python or Bash can be used to automate tasks like collecting static files, running migrations to update the database schema (if applicable), and creating a production-ready build of the application.

3. Environment Selection:

- The deployment script prompts the user to select the target environment (e.g., development, staging, production).
- Environment-specific configurations (database credentials, API keys) are loaded based on the chosen environment.

4. Deployment to the Server:

- The deployment script securely transfers the application build files and configuration files to the target server.
- Secure methods like SFTP or rsync can be used for file transfer.

5. Service Restart and Verification:

- The script triggers a restart of the web server process (e.g., restarting Gunicorn for a Django application).
- Once the server restarts, the script can perform basic health checks to ensure the application is running and accessible.

Configuring the Environment

- Clone the Repository - Clone your Django project from your repository

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

- Set Up a Virtual Environment - Create and activate a virtual environment.

```
python3 -m venv venv
source venv/bin/activate
```

- Install Dependencies - Install the required Python packages

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

- Configure Django Settings - Update your Django settings for production. Ensure DEBUG is set to False, and configure your ALLOWED_HOSTS, database settings, and other production settings.

```
DEBUG = True
ALLOWED_HOSTS = []
# Application definition
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    "Sweetspot_pro",
    "rest_framework"
]
```

- Configure PostgreSQL into Django Settings - Update your Django settings for database insertion and linkage.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': '1234',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

- Apply Database Migrations - Apply any database migrations.

```
python manage.py makemigrations
python manage.py migrate
```

Automation with Deployment Scripts

- Deployment scripts automate repetitive tasks, reducing the risk of human error and ensuring consistency across deployments.
- These scripts can be integrated with continuous integration/continuous delivery (CI/CD) pipelines for automated deployments triggered by code merges or version bumps.

Environment-Specific Deployment Instructions

By following these guidelines and leveraging deployment scripts, the Sweetspot team ensures a smooth and efficient deployment process. This allows for swift updates and feature additions, keeping Sweetspot fresh and delightful for both customers and cake stores.

- **Development Environment:**

- The development environment is typically a local machine or a virtual machine accessible only to the development team.
- Deployment scripts might be configured to skip specific steps like database migrations, as the development environment might have a separate database for testing purposes.
- **Staging Environment:**
 - The staging environment mirrors the production environment as closely as possible.
 - It serves as a final testing ground before deploying to production.
 - Deployment to staging should follow the same process as production, but with appropriate environment-specific configurations.
- **Production Environment:**
 - The production environment is where SweetSpot is accessible to the public.
 - Security best practices like strong passwords and encryption should be strictly adhered to in the production environment.
 - Backups of the database and application codebase should be implemented for disaster recovery purposes.

SweetSpot API: Setup and Configuration

To use the SweetSpot API, users need to follow these steps:

Register

- If you are a new user, register an account on the SweetSpot platform by providing your email address, name, password, phone number, and address details.
- Endpoint: `POST/api/customers/register`
- Example Request Body:

```
{
  "email": "user@example.com",
  "name": "John Doe",
  "password": "password123",
  "phone": "123-456-7890",
  "address": "123 Sweet St, Candyland, CA 12345"
}
```

Login

- Once registered, login to your account using your email address and password.
- Endpoint: `POST /api/customers/login`
- Example Request Body:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

Explore Cakes

- Browse through the available cakes listed on the platform to view their names, flavors, sizes, prices, descriptions, and images.
- Endpoint: `GET /api/cakes`
- Example Request: `GET /api/cakes`

Customize Orders

- Select a cake and customize your order by specifying additional options such as message, egg version, toppings, and shape.
- Endpoint: `POST /api/cake-customizations`
- Example Request Body:

```
{  
  "cake_id": 1,  
  "message": "Happy Birthday!",  
  "egg_version": true,  
  "toppings": ["chocolate", "nuts"],  
  "shape": "heart"  
}
```

Manage Cart

- Add customized cakes to your cart, adjust quantities if necessary, and review the total price.
- Endpoint: `POST /api/carts`
- Example Request Body:

```
{  
  "customization_id": 1,  
  "quantity": 2  
}
```

- To update quantities or remove items, use the appropriate endpoints (`PUT /api/carts/{item_id}`, `DELETE /api/carts/{item_id}`).

Place Order

- Proceed to checkout, provide delivery address details, select a payment method, and confirm your order.
- Endpoint: `POST /api/orders`
- Example Request Body:

```
{  
  "cart_id": 1,  
  "delivery_address": "123 Sweet St, Candyland, CA 12345",  
  "payment_method": "credit_card",  
  "credit_card_details": {  
    "number": "4111111111111111",  
    "expiration_date": "12/25",  
    "cvv": "123"  
  }  
}
```

Usage Instructions

API Endpoints

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

- `/api/customers`: Manage customer accounts (registration, login, profile).
- Example: `POST /api/customers/register``, `POST /api/customers/login``
- `/api/cakes`: View and search available cakes.
- Example: `GET /api/cakes``
- `/api/cake-customizations`: Customize cakes with additional options.
- Example: `POST /api/cake-customizations``
- `/api/carts`: Manage shopping carts (add items, update quantities, remove items).
- Example: `POST /api/carts``, `PUT /api/carts/{item_id}``, `DELETE /api/carts/{item_id}``
- `/api/orders`: Place orders and track order status.
- Example: `POST /api/orders``, `GET /api/orders/{order_id}``
- `/api/stores`: View available stores and cakes by store.
- Example: `GET /api/stores``

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.

- Login to get JWT: `POST /api/customers/login``
- Authorization Header: `Authorization: Bearer <your_token_here>``

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

- Issue: Unable to Login
- Symptoms: Users are unable to log in to their accounts despite providing correct credentials.
- Troubleshooting Steps:
 1. Verify that the email address and password entered are correct.
 2. Ensure that the account is registered on the SweetSpot platform.
 3. Check for any typos or mistakes in the email address or password.
 4. Reset the password if forgotten, using the "Forgot Password" functionality.

Issue: Missing or Incorrect Data in Orders

- Symptoms: Users report missing items or incorrect details in their orders.
- Troubleshooting Steps:

1. Review the order confirmation email or order summary to identify discrepancies.
2. Check the cart and order history to verify the items and details submitted.
3. Contact customer support for assistance in resolving the issue and adjusting the order if necessary.

Issue: Payment Failure

- Symptoms: Users encounter errors or failures while attempting to make payments.
- Troubleshooting Steps:

1. Ensure that the payment method selected is valid and supported by the platform.
2. Verify that the payment details (e.g., credit card number, expiration date, CVV) are entered correctly.
3. Check for any restrictions or limitations imposed by the payment gateway or bank.
4. Try using an alternative payment method or contact customer support for further assistance.

Issue: Order Delivery Delay

- Symptoms: Users experience delays in receiving their orders beyond the estimated delivery time.
- Troubleshooting Steps:

1. Track the order status using the provided tracking information or order ID.
2. Contact the delivery service provider for updates on the delivery status and any potential delays.
3. Verify the delivery address provided during checkout for accuracy and completeness.
4. Reach out to customer support if the delay persists or for further assistance in resolving the issue.

Issue: Technical Errors or System Downtime

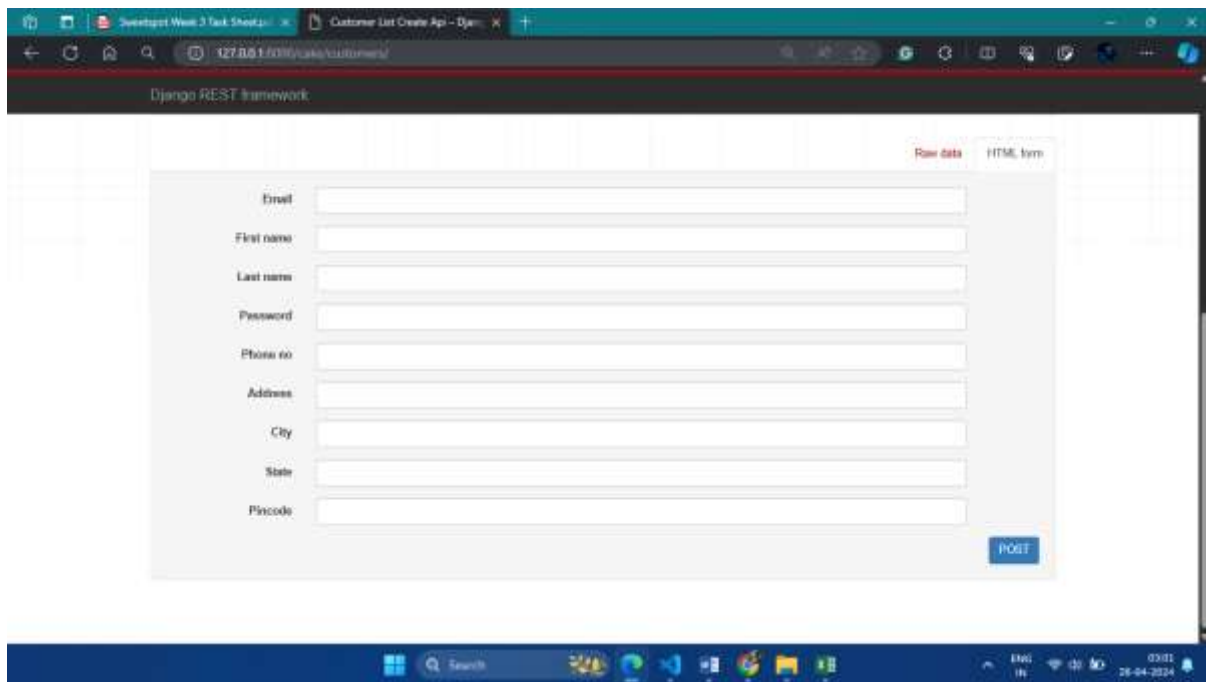
- Symptoms: Users encounter technical errors or disruptions while using the SweetSpot platform.
- Troubleshooting Steps:

1. Refresh the page or restart the application to attempt to resolve transient issues.
2. Check for any announcements or notifications regarding scheduled maintenance or system downtime.
3. Clear browser cache and cookies, or try accessing the platform from a different device or browser.
4. Report the issue to the platform's technical support team, providing details of the error encountered for investigation and resolution.

API testing screenshots

POST /users/register/ - User Registration and Login:

CREATE



Django REST framework

Row data HTML form

Email

First name

Last name

Password

Phone no

Address

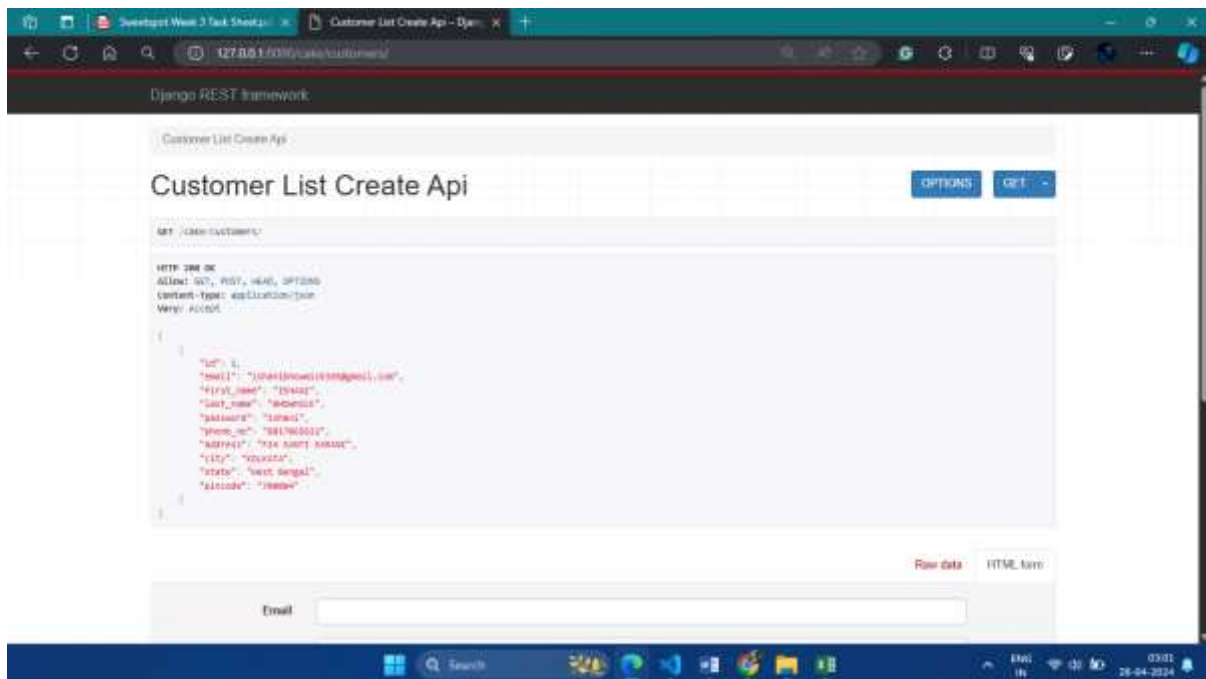
City

State

Pincode

POST

AVAILABLE DATABASE



Customer List Create Api

OPTIONS GET

GET /users/register/

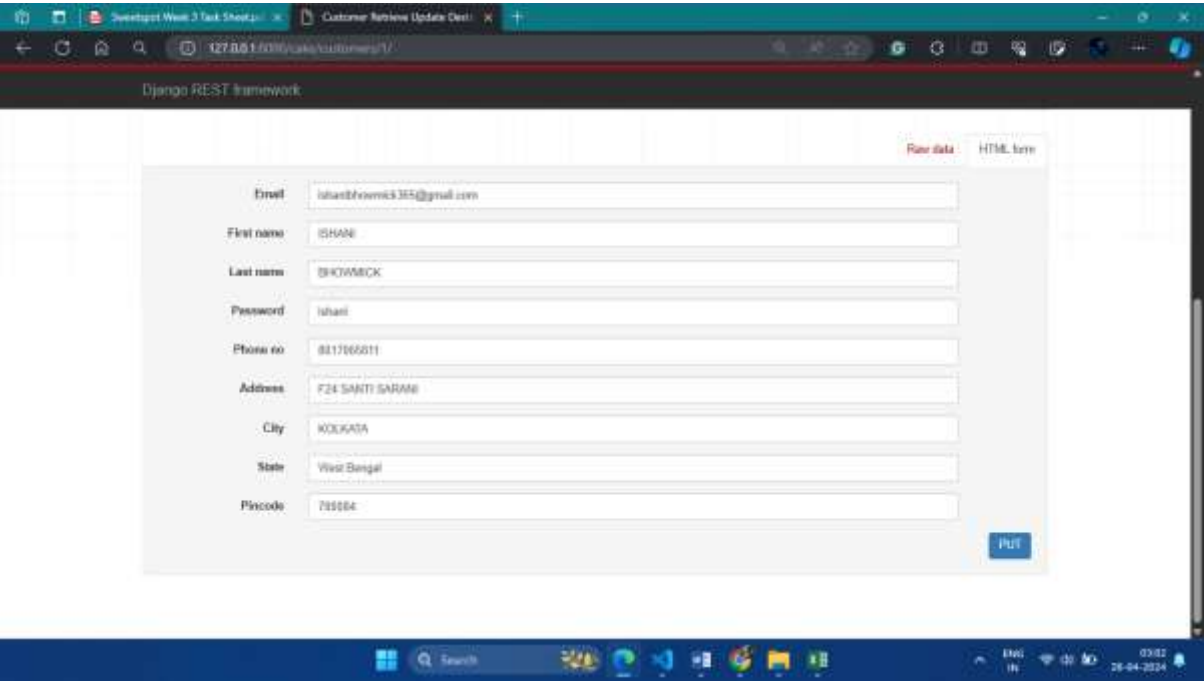
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
  "id": 1,
  "email": "john.doe@example.com",
  "first_name": "John",
  "last_name": "Doe",
  "password": "123456",
  "phone_no": "9876543210",
  "address": "123 Main Street",
  "city": "New York",
  "state": "New York",
  "pincode": "10001"
}
```

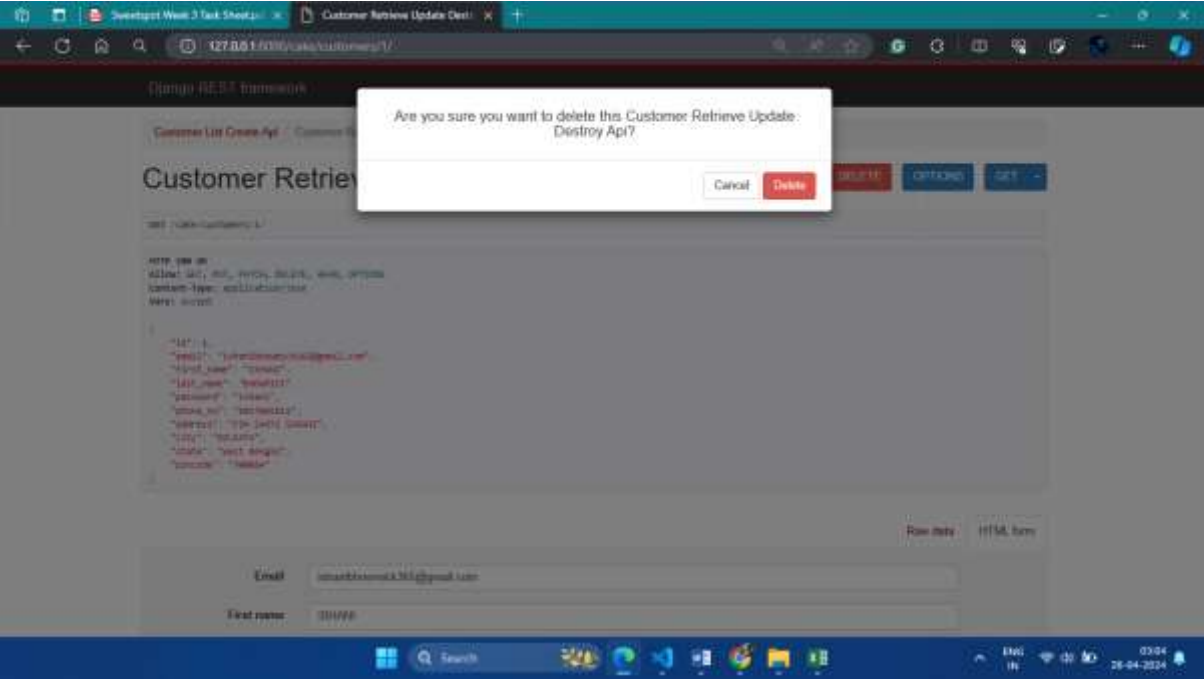
Row data HTML form

Email

UPDATE

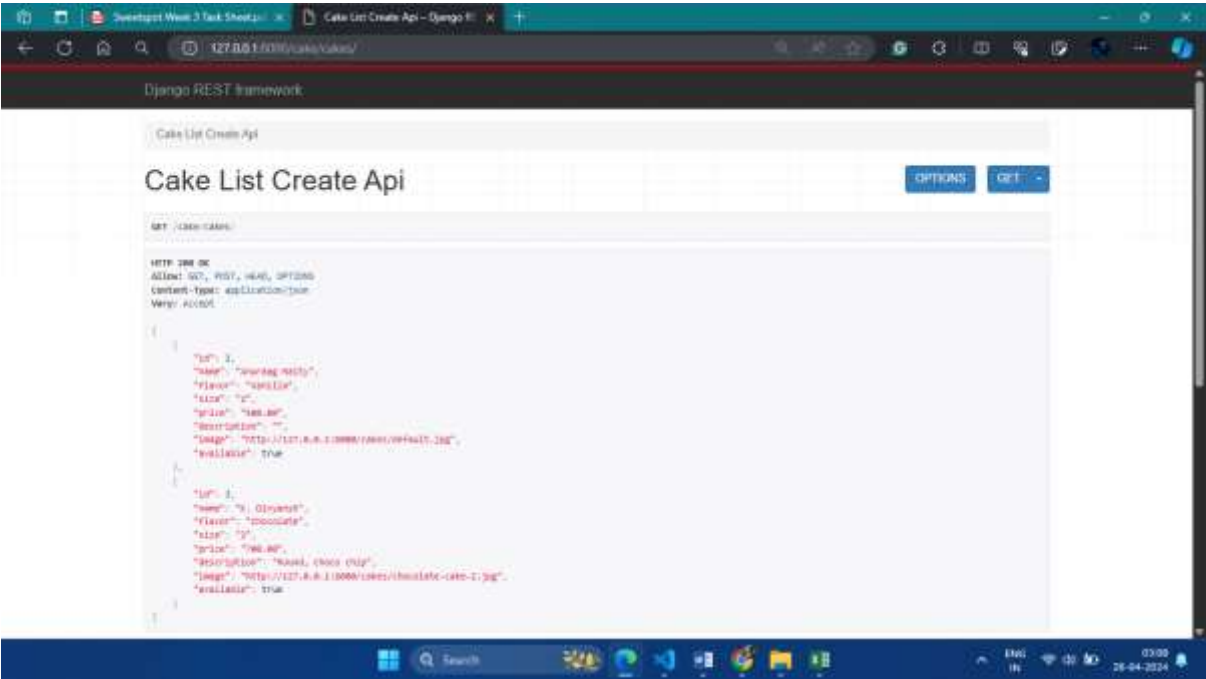


DELETE

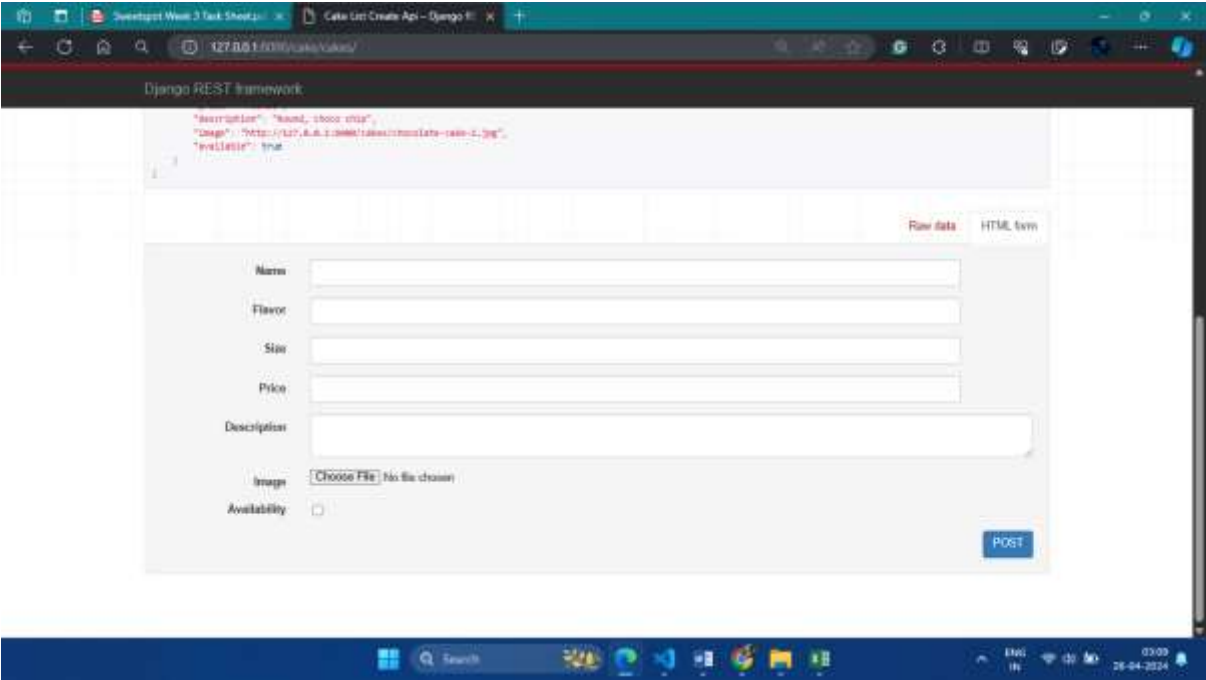


Cake CRUD APIs

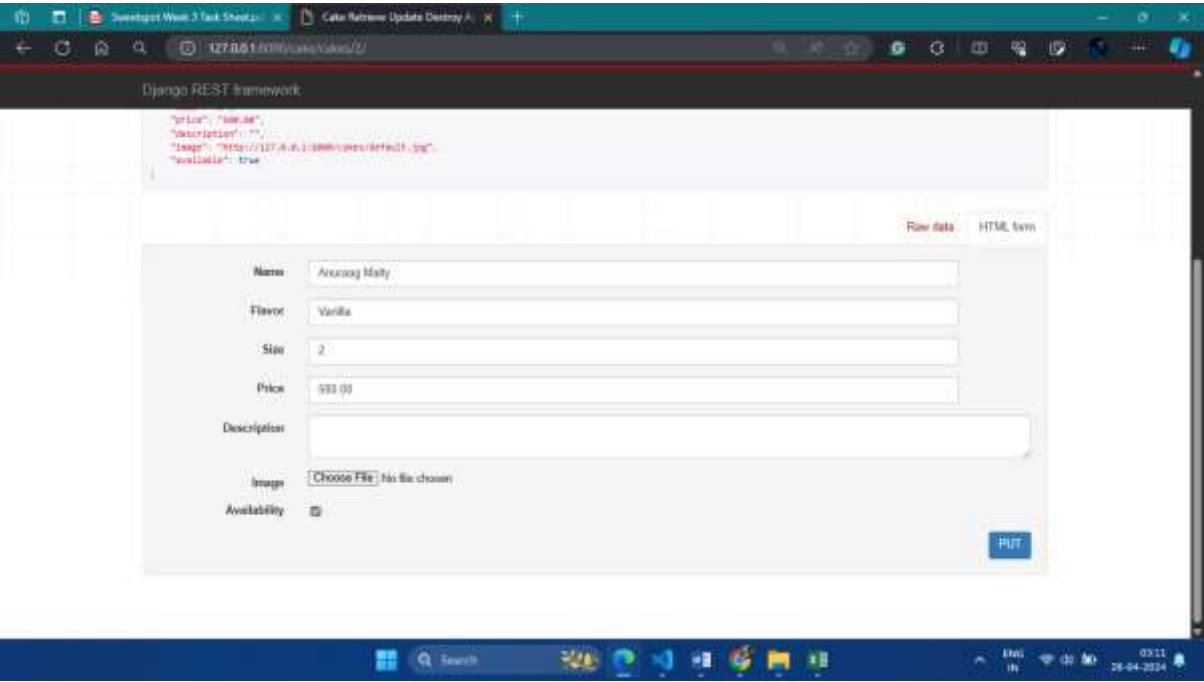
AVAILABLE DATABASE



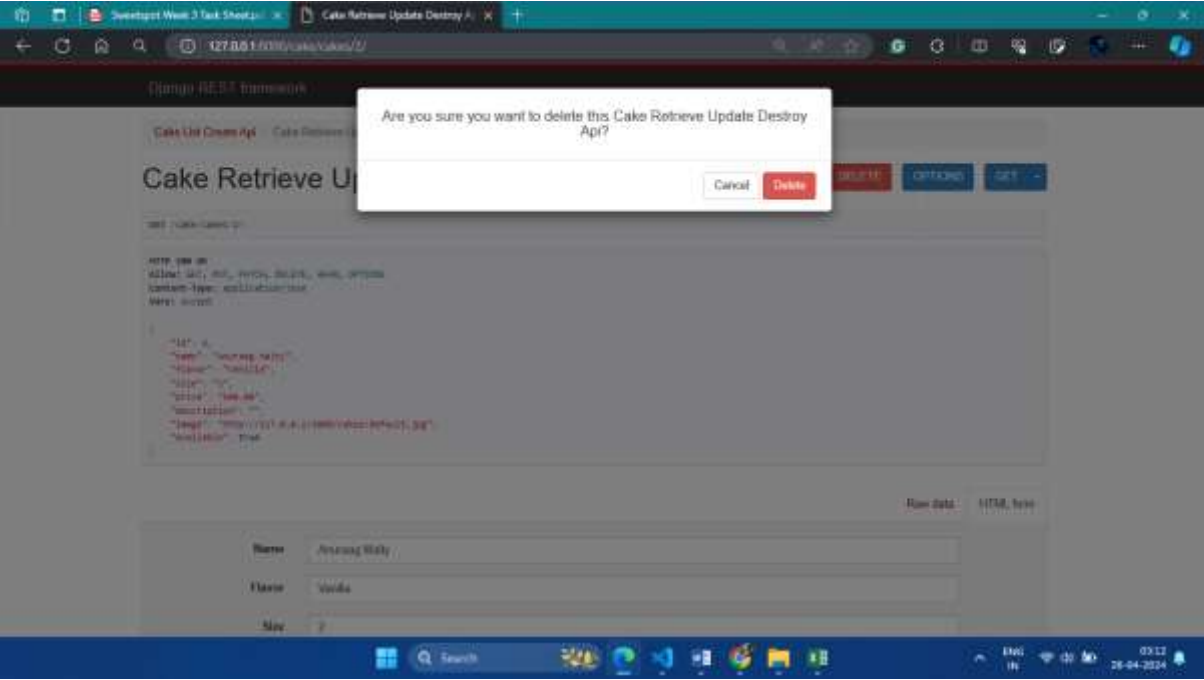
CREATE



UPDATE

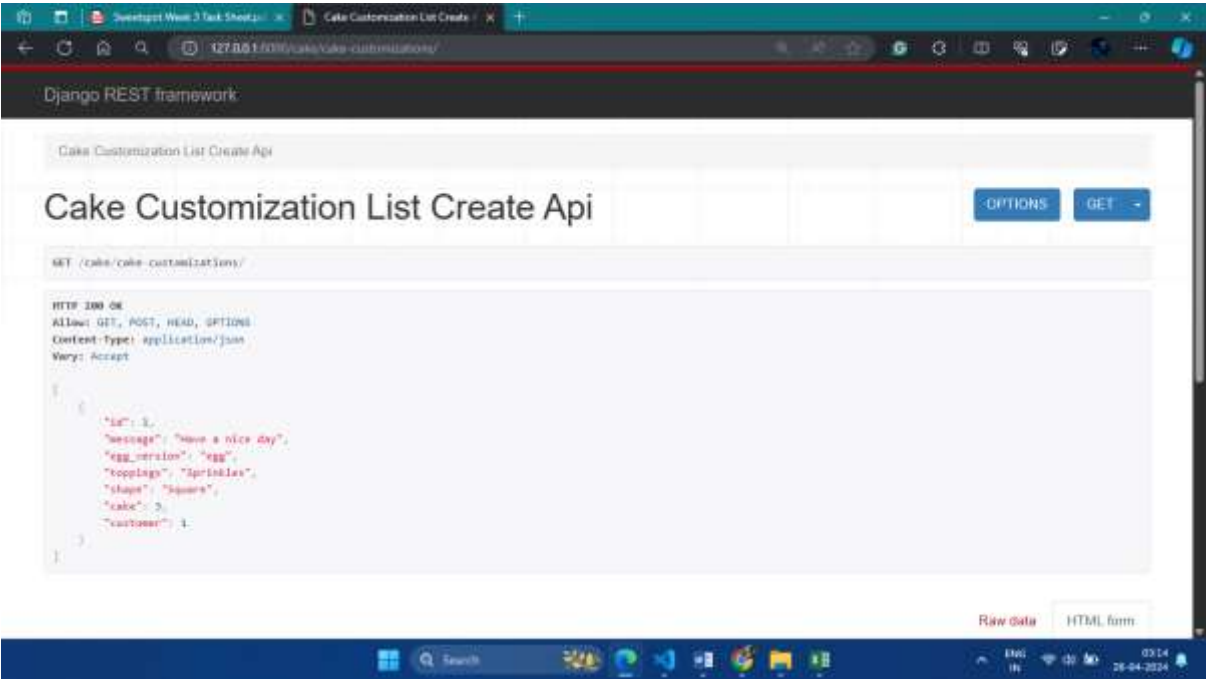


DELETE

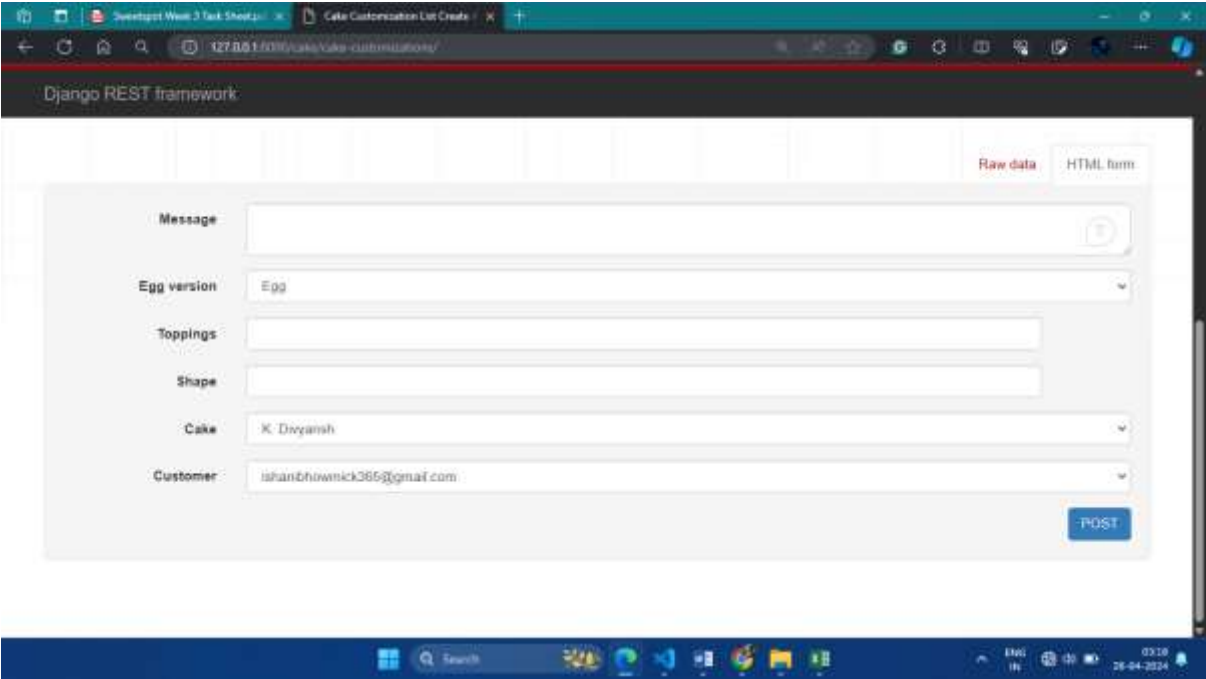


Cake Customization options CRUD APIs

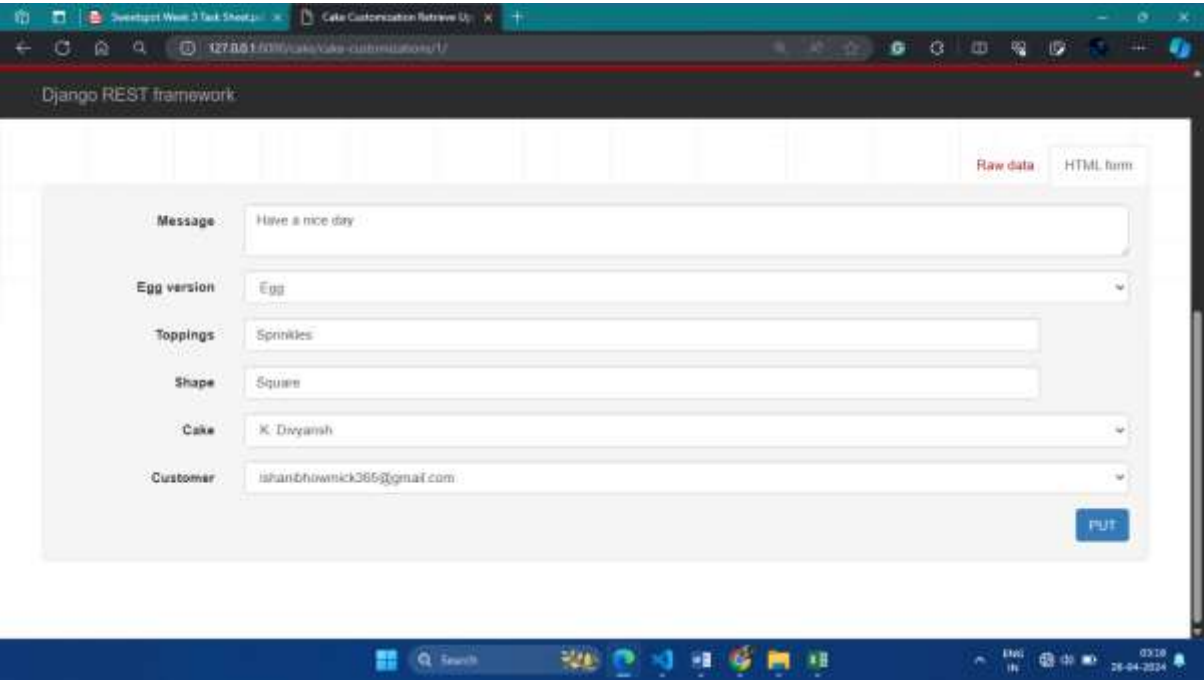
AVAILABLE DATABASE



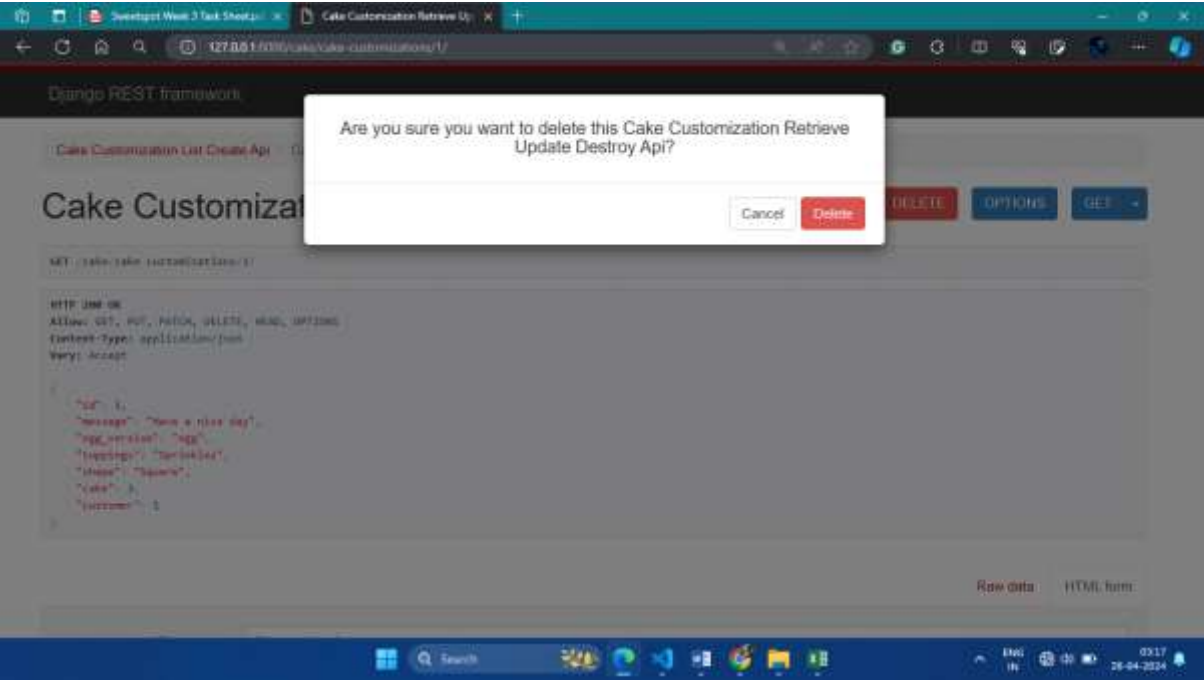
CREATE



UPDATE



DELETE



Conclusion

Sweetspot, the delightful online cake delivery platform, has successfully emerged from development. This project stands as a testament to the combined efforts of the development team, delivering a user-friendly platform that bridges the gap between customers with a craving for delicious cakes and the bakeries offering these delectable treats. From the initial stages of conceptualization to the final deployment, the team meticulously crafted Sweetspot to provide a seamless and enjoyable experience for all users. This user-centric approach ensures that both cake enthusiasts and bakery owners can leverage the platform to their fullest advantage.

Outcomes

- **A Functional and Scalable Platform:** Sweetspot boasts a robust backend built with Python, Django, and PostgreSQL, ensuring a secure and scalable foundation for future growth.
- **Seamless User Experience:** The platform offers a user-friendly interface for browsing cakes, placing orders, and tracking deliveries.
- **Focus on Security and Testing:** Rigorous coding practices, adherence to best practices, and a multi-layered testing approach ensure a secure and reliable user experience.
- **Automated Deployment:** Deployment scripts streamline the process of moving Sweetspot from development to production environments, minimizing human error and ensuring consistency.

Lessons Learned and Areas for Improvement

- **The Value of an MVP:** Focusing on a Minimum Viable Product (MVP) allowed for a faster launch and valuable user feedback. This iterative approach ensures the platform evolves in line with user needs.
- **Importance of Clear Documentation:** Well-documented code and comprehensive user guides are crucial for smooth development, onboarding new team members, and providing excellent user support.
- **Continuous Monitoring and Improvement:** Continuously monitoring user feedback, platform performance, and industry trends will guide future development efforts and ensure Sweetspot remains at the forefront of online cake delivery.

Looking Forward

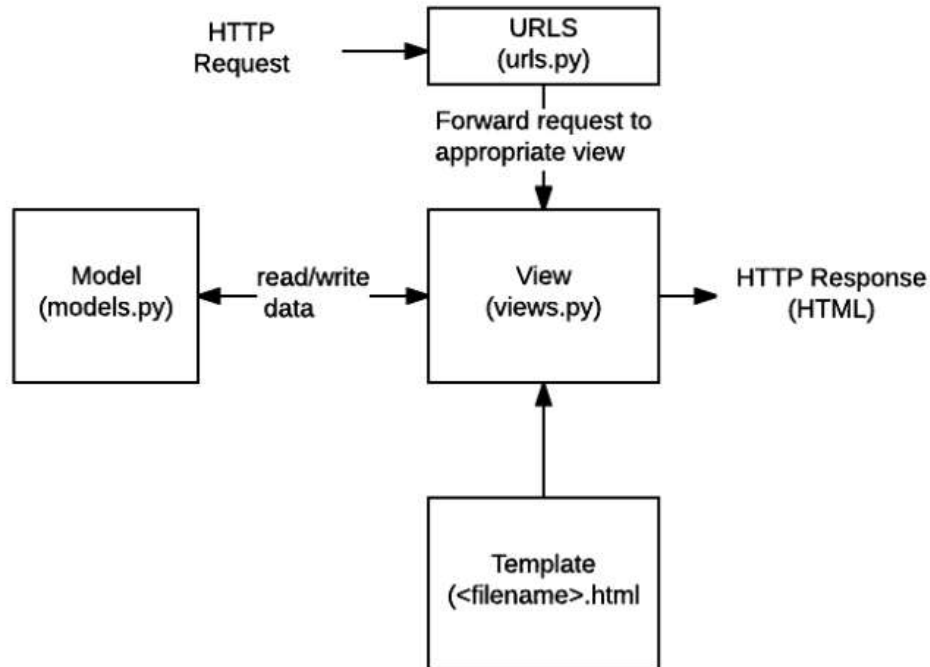
The launch of Sweetspot is just the beginning. The development team will leverage the lessons learned and the feedback received to continuously improve the platform. Future enhancements might include:

- **Mobile App Development:** A dedicated mobile app could provide an even more convenient way for users to order cakes on the go.
- **Real-Time Order Tracking:** Implementing real-time order tracking would allow users to see the progress of their delivery and enhance the overall user experience.
- **Social Login Options:** Integrating social login options could streamline the user registration process and potentially attract new customers.

Appendices

Appendix A

Additional Diagram:



Working Flow of Django RESTFramework Diagram

Appendix B

Code Snippets for models.py:

```
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, PermissionsMixin
from django.db import models
```

```
class CustomUserManager(BaseUserManager):
    def create_user(self, email, phone_no, first_name, last_name, city, state, address, pincode, password=None,
**extra_fields):
        if not email:
            raise ValueError('The Email field must be set')
        email = self.normalize_email(email)
        user = self.model(
            email=email,
            phone_no=phone_no,
            first_name=first_name,
            last_name=last_name,
            address=address,
            state=state,
            city=city,
            pincode=pincode,
            **extra_fields
        )
        user.set_password(password)
```

```

        user.save(using=self._db)
        return user

    def create_superuser(self, email, phone_no, first_name, last_name, city, state, address, pincode, password,
**extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True.')
        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')

        return self.create_user(email, phone_no, first_name, last_name, city, state, address, pincode, password,
**extra_fields)

class Customer(AbstractBaseUser,PermissionsMixin):
    email = models.CharField(unique=True,max_length=50)
    phone_no = models.CharField(max_length=10)
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=50)
    state = models.CharField(max_length=50)
    pincode = models.CharField(max_length=6)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    is_superuser = models.BooleanField(default=False)

    objects = CustomUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['phone_no', 'address', 'pincode','first_name','last_name','city','state']

    def __str__(self):
        return self.email
class Cake(models.Model):
    name = models.CharField(max_length=100)
    flavor = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    size = models.CharField(max_length=50)
    availability = models.BooleanField(default=True)
    image = models.ImageField(upload_to='cakes/', null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    store=models.ForeignKey('store',on_delete=models.CASCADE)
    def __str__(self):
        return self.name
class CakeCustomization(models.Model):
    Cake = models.ForeignKey('Cake', on_delete=models.CASCADE)

```

```

Customer=models.ForeignKey('Customer',on_delete=models.CASCADE)
message = models.CharField(max_length=50,blank=True)
toppings = models.CharField(max_length=100, blank=True)
shape = models.CharField(max_length=100, blank=True)
egg_version = models.BooleanField(default=True)
class Cart(models.Model):
    Customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
    Cake = models.ForeignKey(Cake,on_delete=models.CASCADE)
    Customization=models.ForeignKey(CakeCustomization,on_delete=models.SET_NULL,null=True,default=None)
    quantity = models.IntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    total_amount = models.DecimalField(max_digits=10, decimal_places=2, default=0)

class Order(models.Model):
    # Define choices for payment method
    PAYMENT_METHOD_CHOICES = [
        ('credit_card', 'Credit Card'),
        ('paypal', 'PayPal'),
        ('bank_transfer', 'Bank Transfer'),
        ('other', 'Other'),
    ]

    # Define choices for payment status
    PAYMENT_STATUS_CHOICES = [
        ('pending', 'Pending'),
        ('paid', 'Paid'),
        ('refunded', 'Refunded'),
        ('cancelled', 'Cancelled'),
    ]

    # Define choices for order status
    ORDER_STATUS_CHOICES = [
        ('pending', 'Pending'),
        ('processing', 'Processing'),
        ('shipped', 'Shipped'),
        ('delivered', 'Delivered'),
        ('cancelled', 'Cancelled'),
    ]

    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
    cake_customization = models.ForeignKey(CakeCustomization,
on_delete=models.SET_NULL,null=True,default=None)
    items = models.ForeignKey(Cake,on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField()
    total_price = models.DecimalField(max_digits=10, decimal_places=2)
    order_date = models.DateTimeField(auto_now_add=True)
    delivery_address = models.CharField(max_length=255)
    order_status = models.CharField(max_length=50, choices=ORDER_STATUS_CHOICES, default='pending')

```

```

    payment_status = models.CharField(max_length=50, choices=PAYMENT_STATUS_CHOICES,
default='pending')
    payment_method = models.CharField(max_length=100,
choices=PAYMENT_METHOD_CHOICES,default='other')
    def __str__(self):
        return f"Order {self.pk}"

```

```

class Store(models.Model):
    name = models.CharField(max_length=100)
    city = models.CharField(max_length=100)
    address = models.TextField()
    contact_number = models.CharField(max_length=15)
    email = models.EmailField(unique=True)
    description = models.TextField()
    def __str__(self):
        return self.name

```

Code Snippets for serializers.py:

```

from rest_framework import serializers
from .models import Cake, CakeCustomization, Cart, Order, Customer,Store
from django.shortcuts import get_object_or_404, get_list_or_404

```

```

class CustomerSerializer(serializers.ModelSerializer):

```

```

    class Meta:
        model = Customer
        fields = '__all__'

```

```

class CakeSerializer(serializers.ModelSerializer):

```

```

    class Meta:
        model = Cake
        fields = '__all__'

```

```

class CakeCustomizationSerializer(serializers.ModelSerializer):

```

```

    class Meta:
        model = CakeCustomization
        fields = "__all__"

```

```

class CartSerializer(serializers.ModelSerializer):

```

```

    class Meta:
        model = Cart
        fields = '__all__'

```

```

    def create(self, validated_data):
        cakes = validated_data.pop('cake')
        # customer_id = validated_data['customer']
        inst = Cart.objects.create(**validated_data)
        for cake_id in cakes:
            inst.cake.add(cake_id)
        return inst

```

```

class OrderSerializer(serializers.ModelSerializer):
    class Meta:
        model = Order
        fields = '__all__'

class StoreSerializer(serializers.ModelSerializer):
    class Meta:
        model=Store
        fields='__all__'

def validate_name(self, value):
    if len(value) < 3:
        raise serializers.ValidationError("Name is too short. It must be at least 3 characters long.")
    if len(value) > 100:
        raise serializers.ValidationError("Name is too long. It must be less than or equal to 100 characters.")
    return value

def validate_city(self, value):
    if len(value) < 2:
        raise serializers.ValidationError("City name is too short. It must be at least 2 characters long.")
    if len(value) > 100:
        raise serializers.ValidationError("City name is too long. It must be less than or equal to 100 characters.")
    return value

def validate_contact_number(self, value):
    if not value.isdigit():
        raise serializers.ValidationError("Contact number must contain only digits.")
    if len(value) < 10 or len(value) > 15:
        raise serializers.ValidationError("Contact number must be between 10 and 15 digits.")
    return value

def validate_email(self, value):
    if not serializers.EmailField().run_validation(value):
        raise serializers.ValidationError("Invalid email format.")
    return value

def validate_description(self, value):
    if len(value) > 500:
        raise serializers.ValidationError("Description is too long. It must be less than or equal to 500 characters.")
    return value

```

Code Snippets for urls.py:

```

from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import CakeViewSet, CakeCustomizationViewSet, CartViewSet, CustomerViewSet,
OrderPlacementViewSet,StoreViewSet
from .views import AdminCustomerViewSet, AdminCakeViewSet, AdminCakeCustomizationViewSet,
AdminOrderViewSet, AdminStoreViewSet
from .views import MyCusomToken, MyCustomRefreshToken

```

```

from .views import CakeViewSet, CakeCustomizationViewSet, CartViewSet, CustomerViewSet,
OrderPlacementViewSet
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

router = DefaultRouter()
# Regular user viewsets
router.register(r'customers', CustomerViewSet, basename='customer')
router.register(r'cakes', CakeViewSet, basename='cake')
router.register(r'cake-customizations', CakeCustomizationViewSet, basename='cake-customization')
router.register(r'carts', CartViewSet, basename='cart')
router.register(r'orders', OrderPlacementViewSet, basename='order')
router.register(r'stores', StoreViewSet, basename='stores')

# Admin-only viewsets
router.register(r'admin/customers', AdminCustomerViewSet, basename='admin-customer')
router.register(r'admin/cakes', AdminCakeViewSet, basename='admin-cake')
router.register(r'admin/cake-customizations', AdminCakeCustomizationViewSet, basename='admin-cake-
customization')
router.register(r'admin/orders', AdminOrderViewSet, basename='admin-order')
router.register(r'admin/stores', AdminStoreViewSet, basename='admin-store')

urlpatterns = [
    path('token/', MyCusomToken.as_view(), name='token_obtain_pair'),
    path('token/refresh/', MyCustomRefreshToken.as_view(), name='token_refresh'),
    path("", include(router.urls)),
    path('order/<int:pk>/track-delivery/', OrderPlacementViewSet.as_view({'post': 'track_delivery'}), name='order-
track-delivery') ]

```

Code Snippets for views.py:

```

from rest_framework.response import Response
from rest_framework import status
from django.forms import model_to_dict
from rest_framework.decorators import action
from django.contrib.auth.hashers import check_password
from rest_framework import viewsets
from django.utils.decorators import method_decorator
from django.views.decorators.csrf import csrf_exempt
from .models import Cake, CakeCustomization, Cart, Customer, Order, Store
from .models import Cake, CakeCustomization, Cart, Customer, Order
from .serializers import (
    CakeSerializer,
    CakeCustomizationSerializer,
    CartSerializer,
    CustomerSerializer,
    OrderSerializer,
    StoreSerializer
)
from django.contrib.auth.hashers import make_password
from django.core.mail import send_mail
from django.template.loader import render_to_string

```

```
from django.utils.html import strip_tags
from django.shortcuts import get_object_or_404
import datetime
import threading
import sched
import time
from datetime import datetime, timedelta
from googlemaps import Client as GoogleMapsClient
from rest_framework.permissions import IsAuthenticated
from rest_framework import permissions
from .permissions import IsAdminUser
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView
from rest_framework.request import Request
from rest_framework.response import Response
```

```
class MyCusomToken(TokenObtainPairView):
    def post(self, request: Request, *args, **kwargs) -> Response:
        res = super().post(request, *args, **kwargs)
        if (res.status_code == 200):
            res.set_cookie(
                key='access',
                value=res.data['access'],
                httponly=True,
                secure=True,
                samesite='Lax',
            )
            res.set_cookie(
                key='refresh',
                value=res.data['refresh'],
                httponly=True,
                secure=True,
                samesite='Lax',
            )
            res.data = {
                "info": "Logged In Successfully"
            }
        return res
```

```
class MyCustomRefreshToken(TokenRefreshView):
    def post(self, request: Request, *args, **kwargs) -> Response:
        response = super().post(request, *args, **kwargs)
        if(response.status_code == 200):
            response.set_cookie(
                key='access',
                value=response.data['access'],
                httponly=True,
                secure=True,
                samesite='Lax',
            )
            response.data = {
```



```
        "info": 'Access token has been refreshed and set in cookies.'
    }
    return response
```

Admin Viewsets

```
class AdminCustomerViewSet(viewsets.ModelViewSet):
    queryset = Customer.objects.all()
    serializer_class = CustomerSerializer
    permission_classes = [IsAuthenticated, IsAdminUser]
```

```
class AdminCakeViewSet(viewsets.ModelViewSet):
    queryset = Cake.objects.all()
    serializer_class = CakeSerializer
    permission_classes = [IsAuthenticated, IsAdminUser]
    def get_permissions(self):
        if self.action in ['list', 'retrieve']:
            return [permissions.IsAdminUser()]
        return [IsAdminUser()]
    @action(detail=False, methods=['get'], url_path='by-store/(?P<store_id>\d+)')
    def get_cakes_by_store(self, request, store_id=None):
        try:
            cakes = Cake.objects.filter(store_id=store_id)
            if not cakes.exists():
                return Response({"error": "No cakes found for this store."}, status=status.HTTP_404_NOT_FOUND)
            serializer = CakeSerializer(cakes, many=True)
            return Response(serializer.data, status=status.HTTP_200_OK)
        except Store.DoesNotExist:
            return Response({"error": "Store not found"}, status=status.HTTP_404_NOT_FOUND)
```

```
class AdminCakeCustomizationViewSet(viewsets.ModelViewSet):
    queryset = CakeCustomization.objects.all()
    serializer_class = CakeCustomizationSerializer
    permission_classes = [IsAuthenticated, IsAdminUser]
    def get_permissions(self):
        if self.action in ['list', 'retrieve']:
            return [permissions.IsAdminUser()]
        return [IsAdminUser()]
```

```
class AdminOrderViewSet(viewsets.ModelViewSet):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer
    permission_classes = [IsAuthenticated, IsAdminUser]
    def get_permissions(self):
        if self.action in ['list', 'retrieve']:
            return [permissions.IsAdminUser()]
        return [IsAdminUser()]
```

```
class AdminStoreViewSet(viewsets.ModelViewSet):
    queryset = Store.objects.all()
    serializer_class = StoreSerializer
    permission_classes = [IsAuthenticated, IsAdminUser]
    def get_permissions(self):
```

```

    # Override to allow unrestricted access to 'store_has_cakes' action
    if self.action == 'store_has_cakes':
        return [permissions.AllowAny()]
    return super().get_permissions()
# Now any user can fetch all the cakes for a particular store.
@action(detail=True, methods=['get'], url_path='store-has-cakes')
def store_has_cakes(self, request, pk=None):
    try:
        store = Store.objects.get(pk=pk)
    except Store.DoesNotExist:
        return Response({"message": "Store not found"}, status=status.HTTP_404_NOT_FOUND)

    cakes = Cake.objects.filter(store=store)
    serializer = CakeSerializer(cakes, many=True)
    return Response(serializer.data, status=status.HTTP_200_OK)

def validate_card(data):
    card_number = data.get("card_number")
    expiration_date = data.get("expiration_date")
    cvv = data.get("cvv")
    f = 0
    if len(card_number.strip()) == 16:
        card_number = list(map(int, card_number))
        t = list(map(str, (map(lambda x: x * 2, card_number[-2::-2]))))
        t = sum([sum(list(map(int, list(i)))) for i in t])
        f = 1 if (t + sum(card_number[-1::-2])) % 10 == 0 else 0
    if not f:
        return False

    if len(cvv) not in [3, 4]:
        return False
    # Check expiration date format and validity
    try:
        expiration_date_obj = datetime.strptime(expiration_date, "%m/%y")
        if expiration_date_obj < datetime.now():
            return False # Expiration date is in the past
    except ValueError:
        return False # Invalid date format
    return True

class CustomerViewSet(viewsets.ModelViewSet):
    queryset = Customer.objects.all()
    serializer_class = CustomerSerializer

    def perform_create(self, serializer):
        serializer.validated_data["password"] = make_password(
            serializer.validated_data["password"]
        )
        serializer.save()
    def perform_update(self, serializer):
        if "password" in serializer.validated_data:

```

```

        serializer.validated_data["password"] = make_password(
            serializer.validated_data["password"]
        )
        serializer.save()
    @action(detail=False, methods=["POST"], url_path="customer-login")
    # def customer_login(self, request: Request) -> Response:
    def customer_login(self, request):
        email = request.data.get("email")
        password = request.data.get("password")
        if not email or not password:
            return Response(
                {"error": "Both email and password are required."},
                status=status.HTTP_400_BAD_REQUEST,
            )
        try:
            customer = Customer.objects.get(email=email)
        except Customer.DoesNotExist:
            return Response(
                {"error": "Invalid email."}, status=status.HTTP_404_NOT_FOUND
            )
        if not check_password(password, customer.password):
            return Response(
                {"error": "Invalid email or password."},
                status=status.HTTP_401_UNAUTHORIZED,
            )
        return Response({"message": "Login Successfull"}, status=status.HTTP_200_OK)

```

```

class CakeViewSet(viewsets.ModelViewSet):
    queryset = Cake.objects.all()
    serializer_class = CakeSerializer
    @action(detail=False, methods=['get'], url_path='by-store/(?P<store_id>\d+)')
    def get_cakes_by_store(self, request, store_id=None):
        try:
            cakes = Cake.objects.filter(store_id=store_id)
            serializer = CakeSerializer(cakes, many=True)
            return Response(serializer.data, status=status.HTTP_200_OK)
        except Cake.DoesNotExist:
            return Response(
                {"error": "Store or Cakes not found"}, status=status.HTTP_404_NOT_FOUND
            )

```

```

class CakeCustomizationViewSet(viewsets.ModelViewSet):
    queryset = CakeCustomization.objects.all()
    serializer_class = CakeCustomizationSerializer
    def create(self, request):
        cake_id = int(request.data.get("cake"))
        customer_id = int(request.data.get("customer"))
        d = request.data.copy()
        d["cake"] = cake_id
        d["customer"] = customer_id
        seria = CakeCustomizationSerializer(data=d)

```

```

if seria.is_valid():
    seria.save()
    return Response(seria.data, status=status.HTTP_200_OK)
return Response(
    {"error": "Erroring in customizing the cake"},
    status=status.HTTP_400_BAD_REQUEST,
)

```

```

class CartViewSet(viewsets.ModelViewSet):
    queryset = Cart.objects.all()
    serializer_class = CartSerializer
    @action(detail=False, methods=["get"], url_path="is-available/(?P<pk>\d+)")
    def cake_is_available(self, request, pk=None):
        try:
            obj = Cake.objects.get(id=pk)
            return Response(
                {"success": "Cake is available"}, status=status.HTTP_302_FOUND
            )
        except:
            return Response(
                {"error": "Cake is not available"}, status=status.HTTP_404_NOT_FOUND
            )
    @action(detail=False, methods=["get"], url_path="is-customize/(?P<pk>\d+)")
    def is_customize(self, request, pk=None):
        try:
            obj = CakeCustomization.objects.get(id=pk)
            return Response(
                "Cake customization exists!!!", status=status.HTTP_302_FOUND
            )
        except:
            return Response(
                "Cake customization not found", status=status.HTTP_404_NOT_FOUND
            )
    def create(self, request):
        post_data = request.data
        cake_id = int(post_data.get("cake_id"))
        try:
            customization_id = int(post_data.get("customization_id"))
        except:
            customization_id = None
        quantity = int(post_data.get("quantity"))
        user_id = int(post_data.get("user_id"))
        try:
            cake = Cake.objects.get(id=cake_id)
        except Cake.DoesNotExist:
            return Response("Cake not found", status=status.HTTP_404_NOT_FOUND)
        if not cake.available:
            return Response("Cake is not available", status=status.HTTP_400_BAD_REQUEST)
        cakeObj = get_object_or_404(Cake, id=cake_id)
        total_amount = cakeObj.price * quantity
        cart = {

```

```

        "customer": user_id,
        "total_amount": total_amount,
        "quantity": quantity,
        "customization": customization_id,
        "cake": [cake.id],
    }
    serializer = self.serializer_class(data=cart)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    else:
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

```

```

class OrderPlacementViewSet(viewsets.ModelViewSet):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer
    def create(self, request):
        data = request.data
        if "order_id" not in data:
            return Response("Order ID is required", status=status.HTTP_400_BAD_REQUEST)
        cake_id = int(data["order_id"])
        cart_item = Cart.objects.filter(cake=cake_id).first()
        cart_item = model_to_dict(cart_item)
        order_data = {
            "customer": cart_item["customer"],
            "cake_customization": cart_item["customization"],
            "items": [cart_item["id"]],
            "quantity": cart_item["quantity"],
            "total_price": cart_item["total_amount"],
            "delivery_address": data.get("delivery_address", "unknown"),
        }
        serializer = OrderSerializer(data=order_data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_200_OK)
        else:
            return Response(
                "Something happened while placing the order!!!",
                status=status.HTTP_403_FORBIDDEN,
            )
    def partial_update(self, request, pk=None):
        if not pk:
            return Response(
                "Error in updating the cake", status=status.HTTP_400_BAD_REQUEST
            )
        order = Order.objects.get(id=pk)
        if not validate_card(request.data):
            return Response(
                {"error": "Invalid credit/debit card information"},
                status=status.HTTP_400_BAD_REQUEST,
            )

```

```

cart = Cart.objects.filter(
    customer=order.customer, customization=order.cake_customization
)
context = {
    "order": order,
    "name": order.customer.first_name + " " + order.customer.last_name,
}
html_content = render_to_string("order_email_template.html", context)
send_mail(
    subject="Payment Successful for your order!",
    message=strip_tags(html_content),
    from_email="prashansa.eric@gmail.com",
    recipient_list=[order.customer.email],
    fail_silently=False,
    html_message=html_content,
)
order.payment_status = "PAID"
order.order_status = "PROCESSING"
cart.delete()
orderSerializer = OrderSerializer(order, data=request.data, partial=True)
if orderSerializer.is_valid():
    orderSerializer.save()
    return Response("Order successfully placed", status=status.HTTP_200_OK)
else:
    return Response(
        "Error in placing the Order!!!", status=status.HTTP_400_BAD_REQUEST
    )
def send_mail(self, subject, message, recipient):
    send_mail(
        subject=subject,
        message=message,
        from_email="prashansa.eric@gmail.com",
        recipient_list=[recipient],
    )
@action(detail=True, methods=["post"], url_path="track-delivery")
def track_delivery(self, request, pk=None):
    order = self.get_object()
    order_data = {
        "customer": order.customer,
        "cake_customization": order.cake_customization,
        "items": [order.id],
        "quantity": order.quantity,
        "total_price": order.total_price,
        "delivery_address": order.delivery_address,
    }
    scheduler = threading.Thread(target=self.delivery_tracking, args=(order_data,))
    scheduler.start()
    return Response("Delivery tracking initiated", status=status.HTTP_200_OK)
def delivery_tracking(self, order_data):
    order = self.get_object()
    # Initialize scheduler

```

```

scheduler = sched.scheduler(time.time, time.sleep)
# Use Distance Matrix API to calculate travel time
gmaps = GoogleMapsClient(key="AIzaSyBcRflkaeZg8jls3gaA53_rDShtdSQBLhg")
origin = "Connaught Place, New Delhi, Delhi, India"
destination = order.delivery_address
distance_matrix = gmaps.distance_matrix(origin, destination)
print(distance_matrix)
travel_time_seconds = distance_matrix["rows"][0]["elements"][0]["duration"]
    "value"
]
estimated_delivery_time = datetime.now() + timedelta(
    seconds=travel_time_seconds
)
# Schedule sending emails
self.send_mail(
    "Order Confirmation",
    "Order placed successfully.",
    order_data["customer"].email,
)
scheduler.enterabs(
    time.mktime((estimated_delivery_time - timedelta(minutes=5)).timetuple()),
    1,
    self.send_mail,
    (
        "Reminder: Order is almost here!",
        "Your order is almost here. Please be ready.",
        order_data["customer"].email,
    ),
)
scheduler.enterabs(
    time.mktime(estimated_delivery_time.timetuple()),
    1,
    self.send_mail,
    (
        "Your order has been delivered!",
        "Your order has been delivered. Enjoy your meal!",
        order_data["customer"].email,
    ),
)
# Run scheduler
scheduler.run()

```

```

class StoreViewSet(viewsets.ModelViewSet):
    queryset = Store.objects.all()
    serializer_class = StoreSerializer

```

Appendix C

Research References:

- Django Documentation: <https://docs.djangoproject.com/>
- Django Rest Framework 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/>