**FoodPickup Application**

The "Food in IIIT Hyderabad" application aims to revolutionize the dining experience within the IIIT Hyderabad campus by introducing an efficient, user-friendly mobile application that facilitates digital menu access, pre-ordering, real-time order tracking, secure payment, and feedback submission for the campus canteens and mess facilities.

**Requirements** –

• Digital Menus and Pre-ordering: Enables users to browse and order from up-to-date menus.

• Real-Time Order Tracking: Allows users to view the status of their orders.

• Payment Integration: Supports multiple secure payment methods.

• Feedback Mechanism: Facilitates submission of feedback on meals and services.

**Task 1: Requirements and Subsystems**

**Functional and Non-functional Requirements: Akanksha**

**Functional Requirements:**

**Digital Menus and Pre-ordering**

Description: Allows users to browse digital menus and place orders ahead of time.

Architectural Significance: This feature demands a dynamic, updatable content management system for menus, requiring robust database design and API integration to ensure real-time accuracy and efficiency in order processing.

**Real-Time Order Tracking**

Description: Enables users to view the status of their orders in real-time.

Architectural Significance: Necessitates the implementation of WebSocket or similar technology for maintaining a live connection between the server and client app, ensuring immediate updates to users.

**Payment Integration**

Description: Supports secure processing of payments through multiple methods.

Architectural Significance: Requires secure, reliable third-party payment gateway integration, emphasizing the need for secure transmission of sensitive information and compliance with financial data protection standards.

**Feedback Mechanism**

Description: Allows users to submit feedback on meals and services.

Architectural Significance: Calls for a responsive system that can collect, store, and analyse feedback, potentially requiring natural language processing for insights and prioritization of feedback for operational improvements.

**Non-Functional Requirements:**

**Performance**

The application must handle multiple simultaneous requests efficiently, ensuring fast loading times and swift order processing to improve user experience and manage peak load times effectively.

**Usability**

An intuitive, user-friendly interface is crucial for encouraging adoption among campus residents. This involves employing best practices in UI/UX design, accessible to all users, including those with disabilities.

**Reliability**

High reliability and uptime are essential, particularly during peak dining hours, requiring robust error handling, failover mechanisms, and possibly the use of cloud services for scalability and reliability.

**Security**

Given the handling of personal and payment information, adhering to security best practices and compliance with data protection regulations is paramount. This includes secure data storage, encrypted communications, and regular security audits.

**Scalability**

The system should be designed to easily accommodate growth, whether in terms of user numbers, menu items, or functionality, without significant rework. This could involve microservices architecture or similar strategies to ensure components can be scaled independently.

**Subsystem Overview:Akanksha**

**User Interaction Subsystem:**

This subsystem facilitates interactions between the users (students, faculty, and staff of IIIT Hyderabad) and the application through a user-friendly mobile interface. It allows users to browse digital menus, place pre-orders, track orders in real-time, and submit feedback on meals and services. This subsystem aims to enhance the dining experience by making it more convenient and efficient for the campus community.

**Order Management Subsystem:**

Central to the application, this subsystem manages the processing of orders from initiation to completion. It includes functionalities for digital menu display, pre-ordering capabilities, real-time order tracking, and integration with payment systems. It serves both the users by providing a seamless ordering experience and the canteen and mess operators by optimizing order handling and preparation processes.

**Payment Integration Subsystem:**

This subsystem is responsible for handling all aspects of payment processing within the application. It supports multiple secure payment methods, ensuring that transactions are executed safely and efficiently. This subsystem is crucial for facilitating seamless financial transactions between the users and the service providers, thus enhancing the overall usability of the application.

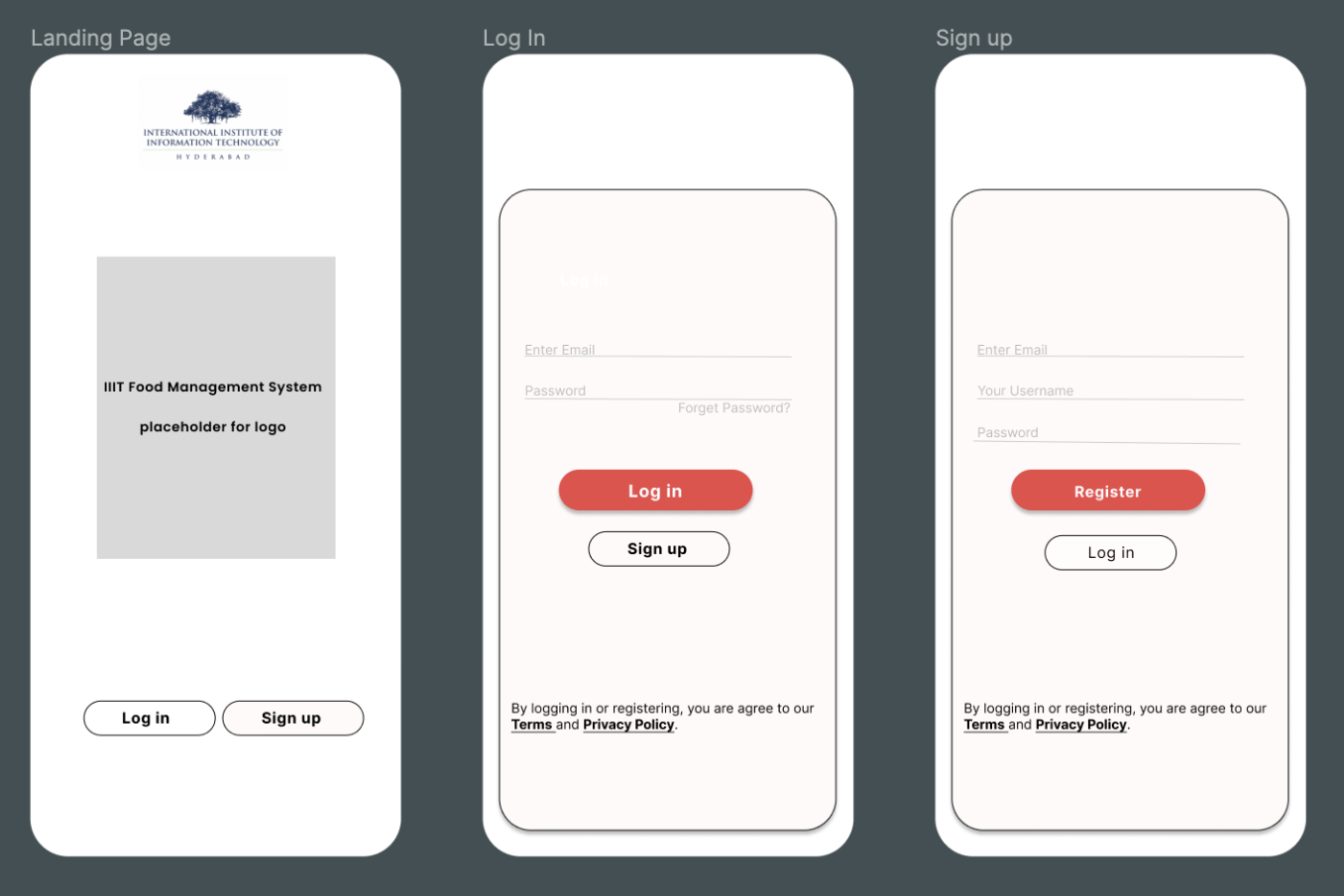
**Feedback Mechanism Subsystem:**

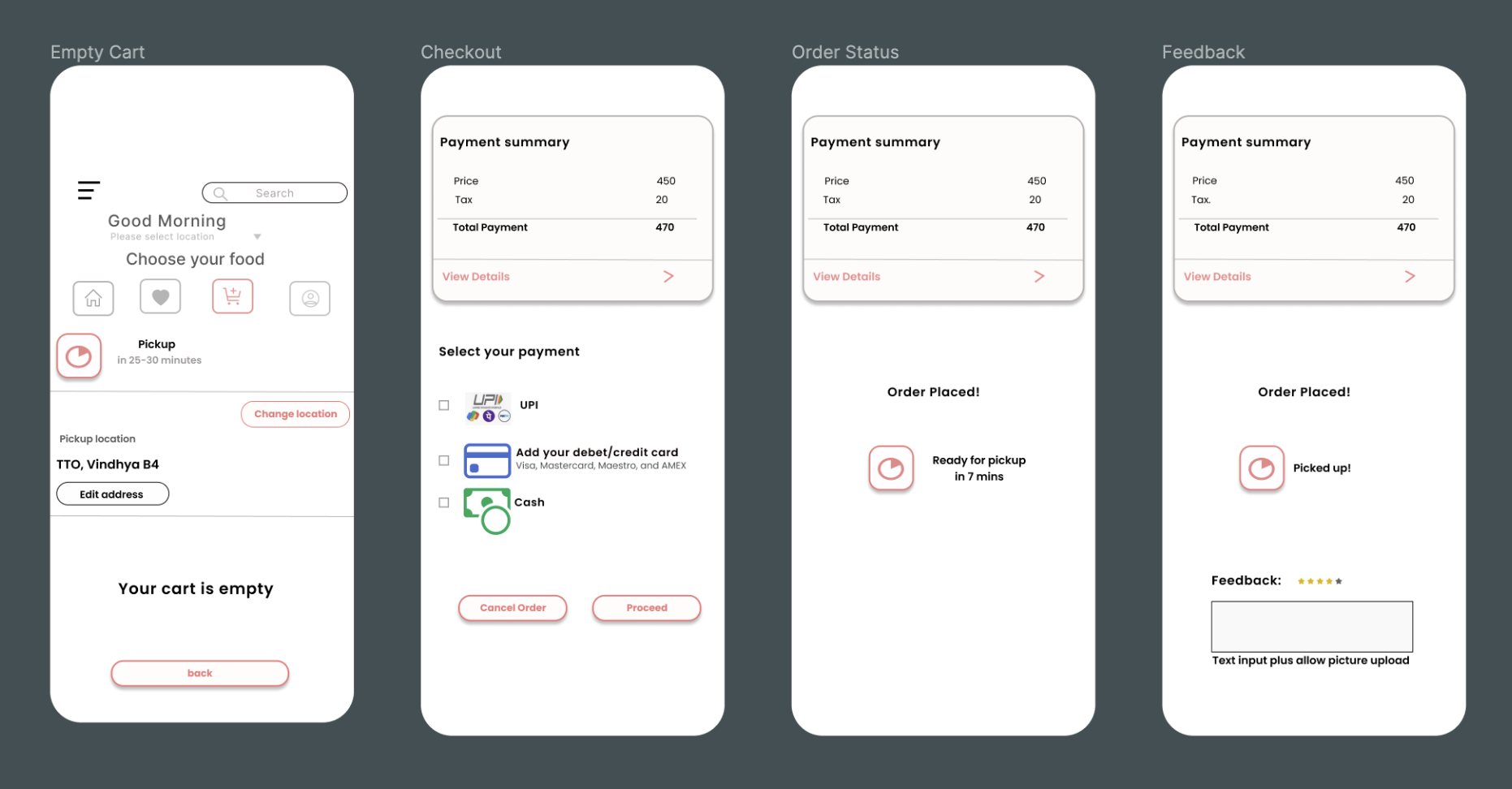
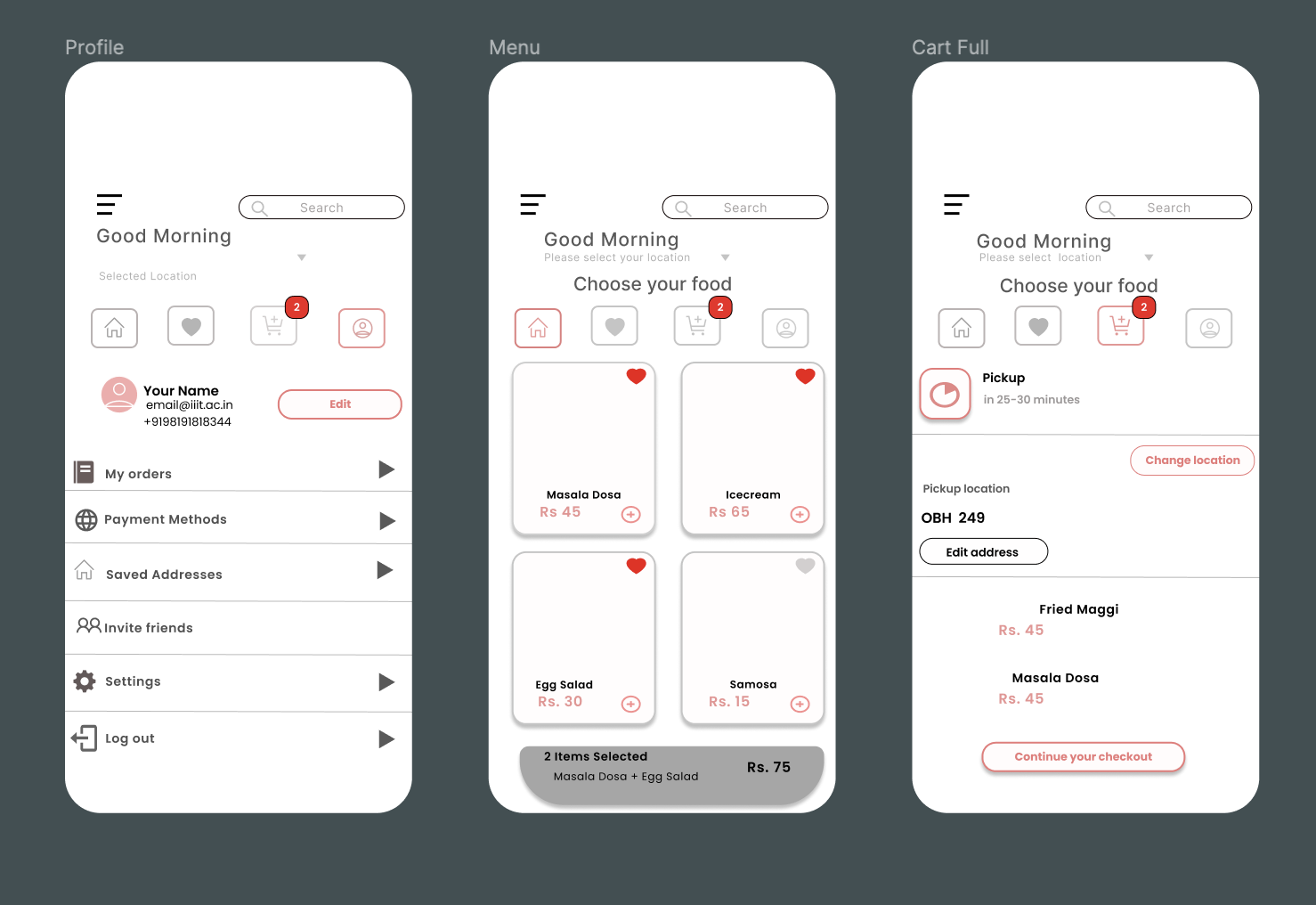
Designed to gather and manage user feedback, this subsystem allows users to submit feedback on their dining experiences directly through the application. This feedback is then available to the canteen and mess operators for service improvement purposes. By fostering engagement between users and service providers, this subsystem plays a vital role in the continuous enhancement of dining services on campus.

**Operational Insights Subsystem:**

A subsystem dedicated to providing actionable insights to canteen and mess operators. This could involve data analysis tools for understanding user preferences, order patterns, and feedback trends to optimise operations and improve services.

**UX Design – Akanksha**

****

****

**Stakeholder Identification:Akanksha**

1. Users (Students, Faculty, Staff)

Concerns:

* Ease of Use: Users demand an intuitive and easy-to-navigate application.
* Service Availability: Concerned with the app's reliability and uptime.
* Menu Variety and Nutritional Information: Looking for diverse food options and nutritional details.
* Payment Security: Ensuring that their financial transactions are safe.
* Order Accuracy and Feedback Responsiveness: Expecting orders to be correctly processed and feedback to be acted upon

Viewpoints and Views:

* User Interface Design View: Focuses on creating an intuitive layout that is accessible and easy to navigate.
* Security View: Emphasizes protecting user data and securing payment transactions.
* Feedback System View: Outlines how users can provide feedback and how it is addressed by the service providers.

2. Canteen and Mess Operators

Concerns:

* Order Management Efficiency: Streamlining the process from order receipt to fulfilment.
* Payment Processing and Inventory Management: Efficient handling of transactions and stock.
* Feedback Reception and Operational Insights: Using feedback for service improvement and gaining insights from data analysis.

Viewpoints and Views:

* Operational Management View: A comprehensive dashboard for tracking orders, managing inventory, and monitoring feedback.
* Payment Integration View: Ensures smooth processing of payments.
* Data Analytics View: Provides insights into customer behavior, menu preferences, and feedback trends.

3. Administrative Staff of IIIT Hyderabad

Concerns:

* System Integrity and Data Privacy: Ensuring the application is secure and user data is protected.
* User Satisfaction and Financial Transparency: Maintaining high service quality and clear financial transactions.
* Regulatory Compliance: Adhering to legal and policy requirements.

Viewpoints and Views:

* Compliance and Privacy View: Focuses on meeting legal standards and protecting user data.
* Financial Management View: Overseeing and reporting on financial transactions.
* System Performance View: Monitoring application reliability and user satisfaction.

4. Developers and Maintainers

Concerns:

* System Scalability: Ability to accommodate growth in user numbers and data volume.
* Maintainability and Extensibility: Ease of updating and adding new features.
* Interoperability: Integration with existing and future systems or services.

Viewpoints and Views:

* Development and Maintenance View: Ensures the technical architecture facilitates easy maintenance and future development.
* Scalability View: Focuses on the system's ability to scale resources as needed.
* Integration View: Addresses the need for seamless integration with other services.

5. External Partners (Payment Gateway Providers, Nutritional Information Sources)

Concerns:

* Integration Ease: Simplifying the integration process with the main system.
* Data Exchange Security: Ensuring secure transmission of data.
* Service Reliability: Dependability of external services for continuous operation.

Viewpoints and Views:

* External Integration View: Outlines the process for integrating external services smoothly.
* Security View: Ensures secure data exchange between systems.
* Reliability View: Focuses on the dependability of external services to maintain uninterrupted operation.

**Major Design Decisions: Aditya**

ADR 1: Adoption of the MVC Architecture

Context: The "Food in IIIT Hyderabad" application requires a robust structure that facilitates clear separation of concerns, simplifies development, and enhances maintainability.

Decision: Implement the application using the Model-View-Controller (MVC) architecture.

Rationale: MVC architecture allows for the separation of data access (Model), user interface (View), and business logic (Controller), promoting organized and modular development. This separation enhances developer productivity, facilitates easier testing, and supports scalable application growth.

Consequences: While MVC provides a structured approach, it requires diligent adherence to the separation of concerns to prevent any one component from becoming overly complex. The team must ensure clear communication and documentation to maintain the architecture's integrity as the application evolves.

**Architecture** – **Aditya**

Users will interact with the system through a mobile application developed using Flutter, known for its excellent UI capabilities and cross-platform efficiency.

**Design Patterns and Architecture**

• MVC Pattern: For separating the application logic, UI, and data.

• Repository Pattern: To abstract the data layer, making the system more modular and testable. • Singleton Pattern: For database connections and API clients to ensure efficient resource use.

• Observer Pattern: For real-time updates on order status.

**Technical Stack** –

• Frontend: Developed in Flutter for compatibility with both Android and iOS devices.

• Backend: Python with frameworks such as Flask to manage APIs, database interactions, and business logic.

• Database: Use of PostgreSQL or MongoDB, depending on schema requirements.

• Payment Integration: Implementation of secure APIs for payment processing.

**Task 3: Architectural Tactics and Patterns**

**<<Nileema>>**

**Architecture Diagram**

A diagram of a software company

Description automatically generated with medium confidence

**DB model** –

A computer screen shot of a computer

Description automatically generated

**Task 4: Prototype Implementation and Analysis**

**Prototype Development:**

**Work Allocation for Prototype -**

1. **API** 
   1. User Registration – Akanksha
   2. FoodMenu Cart, Order, Menu– Chandana
   3. Feedback, Notification, Address, Payment, Canteen - Nileema
2. **DB Creation – Nileema**
3. **UI – Aditya and Bharti**
4. **Pub-Sub for notification – Aditya**

**Architecture Analysis: <<Aditya>>**

**DB model script** –

@startuml

' hide the spot

' hide circle

' avoid problems with angled crows feet

skinparam linetype ortho

!define primary\_key(x) <b><color:#b8861b><&key></color> x</b>

!define foreign\_key(x) <color:#aaaaaa><&key></color> x

!define column(x) <color:#efefef><&media-record></color> x

!define table(x) entity x << (T, white) >>

table(address) {

primary\_key(addressid) : UUID

column(typeofaddress): character varying(120)

column(address\_desc): character varying(120)

column(createddate): datetime

column(updateddate): datetime

foreign\_key(userid): integer <<FK>>

}

table(canteen) {

primary\_key(canteenid): UUID

column(location): character varying(120)

column(canteenname): character varying(120)

column(canteenowner): character varying(120)

column(canteenstatus): character varying(120)

column(createddate) : datetime

column(updateddate) :datetime

}

table(cart){

primary\_key(cartid): UUID

column(orderid): integer

column(cartstatus): character varying(50)

column(cartprice): integer

column(cartusername): character varying(120)

column(createddate): datetime

column(updateddate): datetime

column(cartuserid): integer

}

table(cartmenuitem) {

primary\_key(cartmenuitemid) : UUID

foreign\_key(cartid) : integer <<FK>>

foreign\_key(menuitemid) : integer <<FK>>

foreign\_key(canteenid): integer <<FK>>

column(menuitemquantity) : integer

column(permenuitemprice) : integer

column(totalmenuitemprice): integer

column(createddate): datetime

column(updateddate): datetime

}

table(feedback) {

primary\_key(feedbackid): UUID

foreign\_key(feedbackuserid) : integer <<FK>>

column(feedbackdesc) : character varying(120)

foreign\_key(orderid) : integer <<FK>>

foreign\_key(menuitemid) : integer <<FK>>

column(feedbackdate) : datetime

column(feedbackstatus) : character varying(50)

column(feedbackaction) : character varying(120)

column(feedbackclosureremarks) :character varying(120)

column(feedbackactionuserid) : character varying(120)

column(createddate) : datetime

column(updateddate) : datetime

column(feedbackusername) : character varying(120)

}

table(menuitem) {

primary\_key(menuitemid) : UUID

foreign\_key(canteenid) : integer <<FK>>

column(menuitemdesc) : character varying(120)

column(Permenuitemprice) : integer

column(menuitemstatus) : character varying(50)

column(menuitemtype) : character varying(50)

column(createddate) : datetime

column(updateddate) : datetime

}

table(notification) {

primary\_key(notificationid): UUID

foreign\_key(userid) : integer <<FK>>

column(notificationdesc) : character varying(120)

column(notificationcreateddate) : datetime

column(notificationtype) : character varying(120)

column(notificationstatus) : character varying(50)

column(createddate) : datetime

column(updateddate) : datetime

}

table(order){

primary\_key(orderid): UUID

foreign\_key(userid) : integer <<FK>>

column(orderdate) : datetime

column(cartid) : integer

column(orderstatus) : character varying(120)

column(paymentid) : integer

column(orderprice) : integer

coumn(createdon) : datetime

coumn(updatedon) : datetime

}

table(users) {

primary\_key(userid) : UUID

column(username) : character varying(80)

column(firstname) : character varying(120)

column(lastname) : character varying(120)

column(mobileno) : bigint

column(email) : character varying(120)

column(aadharid) : character varying(120)

column(userrole) : character varying(120)

column(userstatus) : character varying(120)

column(password) : character varying(100)

column(registered\_on) : timestamp

column(updateddate) : timestamp

column(remarks) : character varying(120)

}

table(payment) {

primary\_key(paymentid) : UUID

foreign\_key(userid) : integer <<FK>>

column(mobileno) : bigint

column(amount) : integer

column(bankaccountnumber) : character varying(120)

column(bankname) : character varying(120)

column(FSCIcode) : character varying(120)

column(pincode) : character varying(100)

column(paymentdate) : timestamp

column(updateddate) : timestamp

column(paymentstatus) : character varying(120)

}

order||..|| users

order ||..|| cart

order ||..|| payment

notification }|..|| users

menuitem||..|| canteen

feedback||..|| menuitem

feedback||..|| order

feedback||..|| users

cart||..||order

cart}|..||users

cartmenuitem }|..||cart

cartmenuitem ||..||menuitem

cartmenuitem }|..||canteen

canteen }|..||users

address}|..||users

@enduml

**API Generation - FoodMenu, Cart, Order, Menu**

**Controllers**

**1. cart\_controller.py**

This file contains a Flask blueprint for handling cart-related operations. It imports necessary functions from the `cart\_service` module and defines routes for adding items to the cart, removing items from the cart, viewing the cart, and placing an order.

- **add\_to\_cart\_route()**: This route handles POST requests to `/add`. It requires a JSON Web Token (JWT) for authentication. It extracts the user ID from the JWT and the menu item ID and quantity from the request data. It calls the `add\_to\_cart` function from the `cart\_service` module to add the item to the user's cart and returns the added cart item with a 201 status code.

- **remove\_from\_cart\_route(cart\_item\_id)**: This route handles DELETE requests to `/remove/<int:cart\_item\_id>`. It requires a JWT for authentication. It extracts the user ID from the JWT and the cart item ID from the URL parameter. It calls the `remove\_from\_cart` function from the `cart\_service` module to remove the item from the user's cart. If the removal is successful, it returns a success message with a 200 status code; otherwise, it returns an error message with a 400 status code.

- **view\_cart()**: This route handles GET requests to `/`. It requires a JWT for authentication. It extracts the user ID from the JWT and calls the `get\_or\_create\_cart` function from the `cart\_service` module to retrieve or create the user's cart. It returns the cart data serialized using the `CartSchema` with a 200 status code.

- **place\_order\_route()**: This route handles POST requests to `/place\_order`. It requires a JWT for authentication. It extracts the user ID from the JWT and calls the `place\_order` function from the `cart\_service` module to place an order for the items in the user's cart. It returns a success message with the order ID and a 201 status code.

**2. menu\_controller.py**

This file contains a Flask blueprint for handling menu-related operations. It imports necessary functions from the `menu\_service` module and defines routes for creating a menu, getting menu details, getting menu items, and creating a menu item.

- **create\_menu()**: This route handles POST requests to `/menus`. It requires a JWT for authentication. It extracts the menu name from the request data and calls the `create\_menu` function from the `menu\_service` module to create a new menu. It returns the created menu data serialized using the `MenuSchema` with a 201 status code.

- **get\_menu\_details(menu\_id)**: This route handles GET requests to `/menus/<int:menu\_id>`. It calls the `get\_menu` function from the `menu\_service` module to retrieve the menu with the provided ID. If the menu is found, it returns the menu data serialized using the `MenuSchema` with a 200 status code; otherwise, it returns a "Menu not found" message with a 404 status code.

- **get\_menu\_items\_list(menu\_id)**: This route handles GET requests to `/menus/<int:menu\_id>/items`. It calls the `get\_menu\_items` function from the `menu\_service` module to retrieve the menu items for the provided menu ID. If menu items are found, it returns the menu items data serialized using the `MenuItemSchema` with a 200 status code; otherwise, it returns a "No menu items found" message with a 404 status code.

- **create\_menu\_item(menu\_id)**: This route handles POST requests to `/menus/<int:menu\_id>/items`. It requires a JWT for authentication. It extracts the menu item name, description, and price from the request data and calls the `create\_menu\_item` function from the `menu\_service` module to create a new menu item for the provided menu ID. If the menu item is created successfully, it returns the created menu item data serialized using the `MenuItemSchema` with a 201 status code; otherwise, it returns a "Failed to create menu item" message with a 400 status code.

**Models**

**1. cart\_model.py**

This file contains database models for the cart, cart items, orders, and order items. It defines the following models:

- **Cart**: This model represents a user's cart. It has columns for the cart ID, user ID (foreign key referencing the `users` table), and the creation timestamp. It has a one-to-many relationship with `CartItem`.

- **CartItem**: This model represents an item in a cart. It has columns for the item ID, cart ID (foreign key referencing the `carts` table), menu item ID (foreign key referencing the `menu\_items` table), quantity, and the creation timestamp.

- **Order**: This model represents an order placed by a user. It has columns for the order ID, user ID (foreign key referencing the `users` table), a one-to-many relationship with `OrderItem`, the total cost of the order, and the creation timestamp.

- **OrderItem**: This model represents an item in an order. It has columns for the item ID, order ID (foreign key referencing the `orders` table), menu item ID (foreign key referencing the `menu\_items` table), quantity, price, and the creation timestamp.

**2. menu\_model.py**

This file contains database models for menus and menu items. It defines the following models:

- **Menu**: This model represents a menu. It has columns for the menu ID, name, creation timestamp, and a one-to-many relationship with `MenuItem`.

- **MenuItem**: This model represents a menu item. It has columns for the item ID, name, description, price, menu ID (foreign key referencing the `menus` table), and the creation timestamp.

The controller and model files work together to provide functionality for managing menus, menu items, user carts, and orders in a restaurant or food delivery application. The controllers define the API endpoints for interacting with the application, while the models define the database structure for storing and retrieving data related to menus, carts, and orders.

**Schemas**

**1. cart\_schema.py**

This file defines Marshmallow schemas for serializing and deserializing cart and cart item data. It imports the `MenuItemSchema` from the `menu\_schema` module for nesting menu item data within the cart item schema.

- **CartItemSchema**: This schema defines the structure for serializing and deserializing a cart item. It includes fields for the item ID, nested `MenuItemSchema` for the menu item details, quantity, and the creation timestamp. The `menu\_item` field is nested using the `MenuItemSchema`.

- **CartSchema**: This schema defines the structure for serializing and deserializing a cart. It includes fields for the cart ID, user ID, a list of nested `CartItemSchema` instances for the cart items, and the cart's creation timestamp.

The schema instances `cart\_schema` and `cart\_item\_schema` are created for use in the controllers or other parts of the application.

**2. menu\_schema.py**

This file defines Marshmallow schemas for serializing and deserializing menu and menu item data.

- **MenuSchema**: This schema defines the structure for serializing and deserializing a menu. It includes fields for the menu ID, name, creation timestamp, and a list of menu items (not serialized by default).

- **MenuItemSchema**: This schema defines the structure for serializing and deserializing a menu item. It includes fields for the item ID, name, description, price, menu ID (the ID of the menu the item belongs to), and the creation timestamp.

The schema instances `menu\_schema`, `menus\_schema` (for serializing multiple menus), `menu\_item\_schema`, and `menu\_items\_schema` (for serializing multiple menu items) are created for use in the controllers or other parts of the application.

**Services**

**1. cart\_service.py**

This module contains functions for managing user carts, adding and removing items from carts, and placing orders.

- **get\_or\_create\_cart(user\_id)**: This function retrieves the cart for the given user ID from the database. If the cart doesn't exist, it creates a new one and saves it to the database. It returns the cart object.

- **add\_to\_cart(user\_id, menu\_item\_id, quantity=1)**: This function adds a menu item to the user's cart. It first retrieves or creates the user's cart using `get\_or\_create\_cart`. It then checks if the menu item is already in the cart and updates the quantity if it is. If the item is not in the cart, it creates a new `CartItem` instance and adds it to the cart. The changes are committed to the database, and the new or updated `CartItem` instance is returned.

- **remove\_from\_cart(user\_id, cart\_item\_id)**: This function removes a cart item from the user's cart. It retrieves the user's cart using `get\_or\_create\_cart` and the `CartItem` instance with the given `cart\_item\_id`. If the `CartItem` belongs to the user's cart, it is deleted from the database, and `True` is returned. Otherwise, `False` is returned.

- **place\_order(user\_id)**: This function places an order for the items in the user's cart. It retrieves the user's cart using `get\_or\_create\_cart` and creates a new `Order` instance. It iterates through the cart items, retrieves the corresponding menu items, and creates `OrderItem` instances for each cart item. The total cost of the order is calculated by summing the prices of the menu items multiplied by their quantities. The `Order` instance is saved to the database, and the cart is cleared. The created `Order` instance is returned.

**2. menu\_service.py**

This module contains functions for managing menus and menu items.

- **create\_menu(name)**: This function creates a new menu with the given name. It instantiates a new `Menu` object with the provided name, adds it to the database session, commits the changes, and returns the created `Menu` instance.

- **get\_menu(menu\_id)**: This function retrieves a menu from the database by its ID. It returns the `Menu` instance if found, or `None` if not found.

- **get\_menu\_items(menu\_id)**: This function retrieves all menu items belonging to the menu with the given ID. It first retrieves the `Menu` instance using `get\_menu`, and then returns the `items` relationship, which is a query object for the associated `MenuItem` instances.

- **create\_menu\_item(name, description, price, menu\_id)**: This function creates a new menu item with the given details and associates it with the menu specified by `menu\_id`. It instantiates a new `MenuItem` object with the provided name, description, price, and `menu\_id`, adds it to the database session, commits the changes, and returns the created `MenuItem` instance.

The schema and service files work together to provide functionality for managing menus, menu items, user carts, and orders in a restaurant or food delivery application. The services modules handle the business logic and interact with the database models, while the schemas define the structure for serializing and deserializing data for use in the controllers or other parts of the application.