Implementation Guide

System Design Questions

1. How would you design the overall architecture?

Overall Architecture Design for the Restaurant Management System

The architecture of this project can be designed using a **Model-View-Template (MVT)** pattern, which is the standard architecture for Django-based applications. Below is a high-level overview of the design:

1. Frontend (Client-side)

- **HTML/CSS/JS**: The frontend of the application will be responsible for rendering the UI for the user. It will interact with the Django backend through HTTP requests (using forms, AJAX, or RESTful APIs). The frontend will be structured into components like:
    - Dashboard (for displaying performance metrics)
    - Restaurant details page
    - Interaction and order management
    - Profile and settings page
    - Performance overview (classified by excellent, good, etc.)
- **Bootstrap**: Used for styling the application, ensuring it is responsive and looks clean across devices.
- **JavaScript**: Handle user interactions like form submissions, data display without reloading the page (AJAX), and dynamic updates on the page.

2. Backend (Server-side)

- **Django**: The server-side framework used for processing requests, interacting with the database, and rendering templates. Django follows the MVT architecture.
    - **Model**: Represents the core database structure (e.g., `Restaurant`, `Order`, `Interaction`, `CallFrequency`). Each model defines the fields and the structure of the data in the database.
    - **View**: Handles the request/response cycle, fetching data from the models, processing it, and rendering it to the templates (HTML) or returning JSON

responses for API routes.

    - **Template**: Renders HTML templates with dynamic data passed from the views. The templates use Django's template language to populate data.

3. Database

- **Relational Database (MySQL)**: The primary database will store all the data, such as restaurant information, orders, interactions, and call frequencies. Django ORM will be used to manage database queries.

    - **Restaurant Model**: Stores information about each restaurant (name, address, status, etc.).
    - **KAM:** store the data of KAM like name,email,timezone etc
    - **Contact:**stores data of all the staff members in the restaurant
    - **Order Model**: Stores orders related to restaurants.
    - **Interaction Model**: Records interactions with restaurants (calls, emails, etc.).
    - **CallFrequency Model**: Keeps track of the call frequency for each restaurant.

4. Authentication & Authorization

- **Django Authentication**: Handles user registration, login, and session management.

- **Permissions**: Define which users can access certain views. For example, a `Key Account Manager (KAM)` might have permission to view and manage orders and interactions for their assigned restaurants.

- **Role-based Access Control**: Based on the user's role (admin, KAM, etc.), we will manage permissions to perform CRUD operations.

5. Task Scheduling and Background Jobs

- **Celery**: Used for background tasks like calculating performance scores, sending emails/notifications, or managing scheduled calls for restaurants. Celery can run periodic tasks using **Django-Celery-Beat** to schedule calls or interactions.

6. Performance Metrics Calculation

- **Performance View**: A dedicated view will aggregate the performance of each restaurant based on criteria like:

    - Number of successful orders
    - Frequency of interactions

- This data is then presented in a structured way, such as:
    - Excellent (Score > 60)
    - Good (Score > 40)
    - Average (Score > 20)
    - Underperforming (Score < 20)

7. URLs and Routing

- **Django URL Dispatcher**: Handles the routing of requests to the appropriate view functions. Each page of the app (e.g., restaurant details, performance, interaction management) has a unique route defined in `urls.py`.

8. Caching

- **Django Cache Framework**: For improved performance and to prevent frequent querying of the same data (e.g., restaurant performance), caching can be implemented to store frequently accessed data in memory.

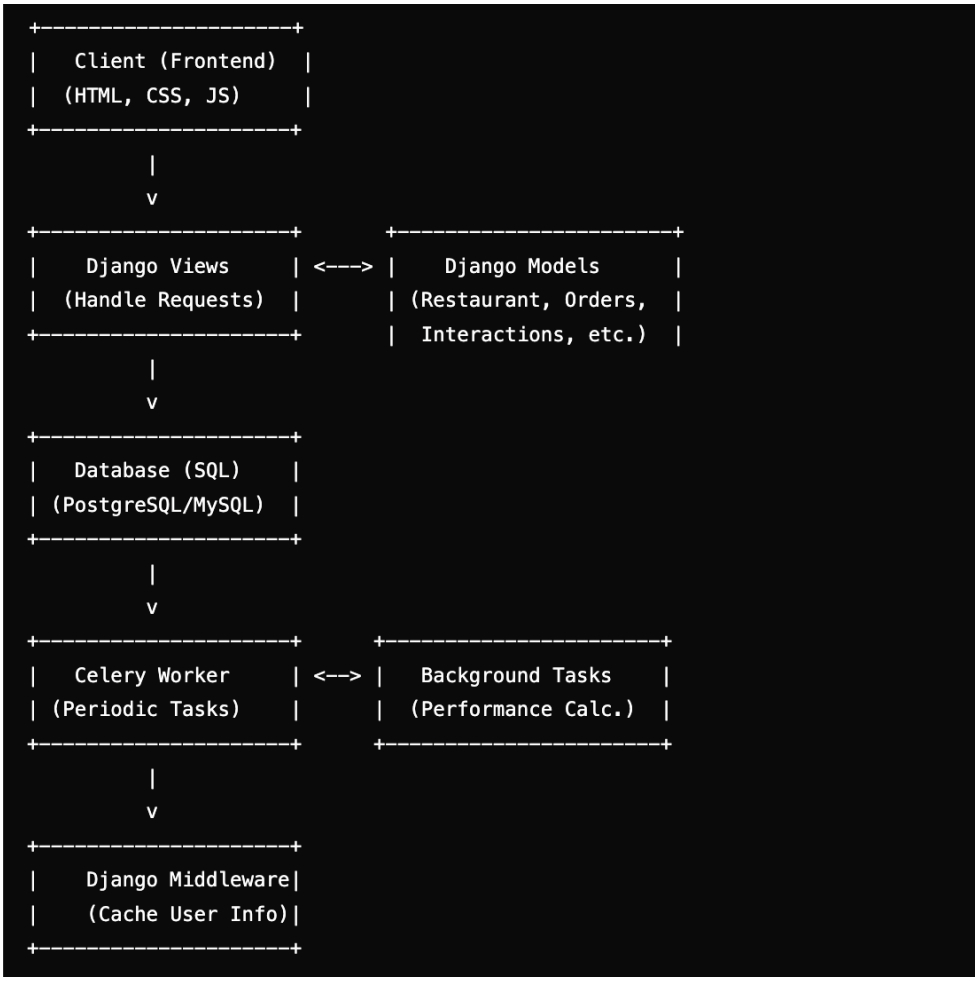
- **Caching User Info using Middleware**

    - ❖ **Django Middleware**: To enhance performance and avoid frequent database queries, custom middleware will be used to cache user-related information. This can help store session details or frequently accessed user data (like performance scores, restaurant preferences, or last login times) in the cache.
        - ➢ The middleware will intercept requests and check if the user's data is already cached. If the data exists, it will be used directly from the cache.
        - ➢ If not, the middleware will fetch the data from the database, store it in the cache, and then proceed with the request.
    - ❖ **Benefits**: This reduces database load, increases the speed of page rendering, and ensures that common user data is quickly accessible.
    - ❖ **Cache Store**: Redis or Django's built-in cache system can be used to store user-related data temporarily.

9. Admin Panel

- **Django Admin**: To make management easier for admins, Django comes with an out-of-the-box admin interface. This can be used for managing restaurant data, orders, interactions, and call frequencies.

Architecture Diagram Overview



Key Technologies Used

- **Django**: Framework for server-side application
- **Django ORM**: To interact with the relational database
- **Celery**: For background tasks
- **PostgreSQL/MySQL**: Database for storing restaurant, order, and interaction data
- **Bootstrap**: For frontend UI design
- **JavaScript (AJAX)**: For dynamic interactions on the frontend
- **Django REST Framework (optional)**: For building APIs if needed

This design ensures a modular, maintainable, and scalable architecture for the Restaurant Management System, focusing on ease of management and performance monitoring of restaurants.

2. What database(s) would you choose and why?

ANS:

For the Restaurant Management System, the choice of database is crucial as it should handle various types of data, such as restaurant details, orders, interactions, performance metrics, and user information, while ensuring high performance and scalability.

I would recommend using the following databases:

1. Relational Database (Primary Choice)

- **MySQL**

Why Choose a Relational Database (RDBMS)?

- **Structured Data**: The system involves structured data such as restaurants, orders, interactions, and performance scores. Relational databases are ideal for this type of data, where entities have well-defined relationships (e.g., each order belongs to a restaurant, each interaction belongs to a restaurant, etc.).
- **ACID Compliance**: Both PostgreSQL and MySQL are ACID-compliant, ensuring the system handles transactions (like order placements and status updates) reliably without data loss or corruption.
- **Complex Queries**: Relational databases support powerful querying and aggregation features like JOINs, GROUP BY, and window functions, which are essential for generating performance metrics, handling orders, and calculating interaction frequencies.
- **Django ORM Support**: Django's built-in ORM is designed to work seamlessly with relational databases, which simplifies development and reduces the need for manual SQL queries.
- **Scalability**: MySQL is scalable, capable of handling high traffic and large datasets, making them suitable for growing applications.
- **Performance**: MySQL can be faster for simple read-heavy operations. If the application requires high read throughput, MySQL might provide better performance in some cases.
- **Maturity and Stability**: MySQL is a well-established database with a large ecosystem, providing reliable performance and support.

3. How would you ensure scalability?

ANS:

Caching for Performance Optimization

- **Django Cache**:

    - **Session Caching**: Use **Django** to cache user session data, restaurant performance scores, and frequently accessed information, reducing the need to query the database repeatedly. This ensures faster response times and reduces the load on the database.
    - **Data Expiry**: Implement cache expiration strategies to ensure that stale data is not served. For example, restaurant performance scores can be cached for a set duration (e.g., 10 minutes) to avoid recalculating them on every request.

Asynchronous Task Processing

- **Celery** : Use **Celery** for handling asynchronous tasks such as sending email notifications, processing long-running computations (e.g., calculating restaurant performance), or updating orders. By offloading these tasks to a background worker, the main application can remain responsive to user requests.
- **Queue Management**: Redis acts as a message broker for Celery, managing the task queue and ensuring efficient task execution. As the application grows, the number of workers can be scaled to ensure tasks are processed quickly and without delay.

Data Modeling Questions

The provided models describe a system that involves managing key account managers (KAM), restaurants, contacts, orders, interactions, and call frequencies. Here's a high-level design for the database schema or class structure based on the given models:

1. User Model (`KAM`)

- **Table Name**: `kam_user`
- **Fields**:
    - `email`: Unique email for the user (Primary Key for authentication).
    - `name`: Full name of the user.
    - `dob`: Date of birth.
    - `phone`: Contact number.
    - `region`: Geographical region associated with the user.
    - `timezone`: Timezone preference.
    - `is_active`: Boolean indicating if the user is active.
    - `is_staff`: Boolean indicating if the user has admin privileges.
- **Indexes**:
    - Unique index on `email`.
- **Relationships**:
    - Has many restaurants through `KAMUserManager`.

2. Restaurant Model (`Restaurant`)

- **Table Name**: `restaurant`
- **Fields**:
  - `name`: Restaurant name.
  - `address`: Address of the restaurant.
  - `region`: Geographical region of the restaurant.
  - `status`: Status of the restaurant (New, Active, Inactive).
  - `kam`: Foreign Key linking to `KAM` (Key Account Manager).
  - `timezone`: Timezone for the restaurant.
- **Indexes**:
  - `kam_id` (Foreign Key to `KAM`).
  - `status`.
- **Relationships**:
  - Has many contacts (through `Contact`).
  - Has many orders (through `Order`).
  - Has many interactions (through `Interaction`).

3. Contact Model (`Contact`)

- **Table Name**: `contact`
- **Fields**:
  - `name`: Name of the contact person.
  - `phone`: Phone number.
  - `email`: Contact email (optional).
  - `role`: Role of the contact (Owner, Manager, Staff).
  - `is_lead`: Boolean to indicate if the contact is the lead for the restaurant.
  - `restaurant`: Foreign Key linking to `Restaurant`.
- **Indexes**:
  - `restaurant_id` (Foreign Key to `Restaurant`).
- **Relationships**:
  - Has many interactions (through `Interaction`).

4. Order Model (`Order`)

- **Table Name**: `order`
- **Fields**:
  - `order_id`: Unique identifier for the order.
  - `restaurant`: Foreign Key linking to `Restaurant`.
  - `date_time`: Date and time of the order.

    - `order_details`: Description of the order.
    - `status`: Status of the order (Pending, Success, Cancelled).
- **Indexes**:
    - `restaurant_id` (Foreign Key to `Restaurant`).
- **Relationships**:
    - Has many interactions (through `Interaction`).

5. Interaction Model (`Interaction`)

- **Table Name**: `interaction`
- **Fields**:
    - `restaurant`: Foreign Key linking to `Restaurant`.
    - `contact`: Foreign Key linking to `Contact`.
    - `kam`: Foreign Key linking to `KAM`.
    - `interaction_type`: Type of interaction (Call, Email, Meeting).
    - `date_time`: Date and time of the interaction.
    - `notes`: Additional notes for the interaction.
    - `is_order_related`: Boolean to indicate if the interaction is linked to an order.
    - `order`: Optional Foreign Key linking to `Order`.
- **Indexes**:
    - `restaurant_id` (Foreign Key to `Restaurant`).
    - `contact_id` (Foreign Key to `Contact`).
    - `order_id` (Foreign Key to `Order`).
- **Relationships**:
    - Belongs to `KAM`, `Restaurant`, and `Contact`.

6. CallFrequency Model (`CallFrequency`)

- **Table Name**: `call_frequency`
- **Fields**:
    - `restaurant`: Foreign Key linking to `Restaurant`.
    - `days_between_calls`: Integer indicating the number of days between calls.
    - `last_call`: Date and time when the last call was made.
    - `next_call`: Date and time when the next call is scheduled.
    - `next_call_list`: list of pair storing {next_call,order_id} where next_call is the next_call needed to be made to that particular restaurant id.
- **Indexes**:
    - `restaurant_id` (Foreign Key to `Restaurant`).
- **Relationships**:
    - Belongs to `Restaurant`.

Relationships Overview:

- **KAM** has many **Restaurants**.
- **Restaurant** has many **Contacts**, **Orders**, and **Interactions**.
- **Contact** has many **Interactions**.
- **Order** has many **Interactions**.
- **Interaction** links **Restaurant**, **Contact**, and **KAM** and may also be related to **Order**.
- **CallFrequency** is associated with each **Restaurant** and stores the call scheduling information.

Data Integrity:

- Foreign keys ensure referential integrity (e.g., a `Restaurant` must have a valid `KAM`).
- Relationships between models ensure consistent data access, like fetching all `Contacts` for a `Restaurant` or all `Orders` for a `Restaurant`.
- Optional fields like `email` in `Contact` are handled using `null=True` to allow flexibility.

This structure efficiently handles restaurant management and interactions, ensuring smooth data management and relationships between key entities (KAM, Restaurants, Contacts, Orders, Interactions).

→How would you handle change in KAM?

Ans:

Business Logic for Changing KAM

- **KAM Reassignment**: When a restaurant's KAM is changed, the `kam_id` field in the `Restaurant` model should be updated to reference the new `KAM`. This reassignment implies that the restaurant is now managed by a different Key Account Manager.
- **Impact on Existing Data**:
    - The restaurant's `KAM` will now be associated with a new user, and future actions (such as interactions, order management, and reporting) will be attributed to the new `KAM`.
    - Past data such as interactions or orders related to the previous `KAM` will still be retained, ensuring historical information remains intact.

Handling timezone differences for **call scheduling** is crucial for ensuring that calls are scheduled at appropriate times for both the Key Account Manager (KAM) and the restaurant, especially when they may be located in different time zones. Here's how to handle timezone differences effectively:

→How would you handle time zone differences for call scheduling?

1. Store Timezone Information for Each Entity

- **KAM's Timezone**: Store the KAM's timezone in the user profile, as KAMs may be in different regions and could be managing restaurants across the globe.
- **Restaurant's Timezone**: As shown in the `Restaurant` model, store the timezone for each restaurant. This ensures that the system is aware of the time zone in which the restaurant operates.

2. Timezone-Aware Datetime

- **Use Timezone-Aware Dates**: Ensure all date and time fields (such as `next_call` in the `CallFrequency` model) are timezone-aware. This ensures that the datetime values are correctly adjusted based on the timezone.
- **Django's Timezone Support**: Django has built-in support for timezone-aware datetimes through the `timezone` module. You can use `timezone.now()` to get the current datetime in the system's timezone or use `pytz` to convert between timezones.