For the backend of a **food delivery app** like Uber Eats, you'll want to use a combination of **design patterns** that support scalability, maintainability, and clean architecture. Here are some recommended design patterns for your backend system:

**1. MVC (Model-View-Controller)**

* **Use Case**: Organizing backend logic and API routes.
* **Why**: It separates business logic, database models, and controllers to keep code clean and maintainable.
* **How**:
  + **Model**: Handles data and database queries (e.g., Order, User, Restaurant).
  + **View**: Not applicable in a pure backend API (only useful in full-stack), but the API response serves as the "view."
  + **Controller**: Handles requests, processes data using models, and sends back the response.

**2. Repository Pattern**

* **Use Case**: For interacting with the database in a clean and reusable way.
* **Why**: Keeps your database queries separate from your business logic, making your system easier to test and maintain.
* **How**:
  + Create a UserRepository, OrderRepository, and RestaurantRepository.
  + Use methods like findUserById(), createOrder(), and getRestaurantMenu().
  + Makes it easier to swap out database engines (e.g., from PostgreSQL to MongoDB) without affecting your business logic.

**3. Singleton Pattern**

* **Use Case**: For shared resources like database connections, logging, and caching.
* **Why**: You only want **one instance** of certain services to exist (e.g., database connections, caching clients, or configuration loaders).
* **How**:
  + Create a Database.js class that initializes and returns the same database instance.
  + Use tools like Redis (for caching) or AWS S3 clients that need to be reused across requests.

**4. Factory Pattern**

* **Use Case**: To create objects dynamically, especially if you have different order types or payment methods.
* **Why**: If you have objects like Order, Payment, and Delivery, you can create them dynamically based on user input (e.g., different payment gateways like PayPal, Stripe, etc.).
* **How**:
  + Create a **PaymentFactory** that returns an instance of a payment class (Stripe, PayPal) based on the input.
  + This avoids direct instantiation and hardcoding of classes.

**5. Observer Pattern**

* **Use Case**: For handling **real-time order tracking** and notifications.
* **Why**: Whenever the delivery status changes (e.g., "Order is being prepared" → "Out for delivery"), you want to notify the customer **and** the restaurant.
* **How**:
  + When the order status changes, notify the **user** (via push notification) and **restaurant** (via a dashboard update).
  + Use **WebSockets** or **EventEmitters** to propagate changes to all "subscribed" systems (e.g., user app, restaurant dashboard).

**6. Strategy Pattern**

* **Use Case**: To select the **payment method** dynamically.
* **Why**: If you want to support multiple payment options like **Stripe, PayPal, or Apple Pay**, you can avoid if-else logic.
* **How**:
  + Implement a PaymentStrategy interface with methods like processPayment().
  + Create classes for **StripePayment**, **PayPalPayment**, and **ApplePayPayment** that follow this interface.
  + Dynamically select which strategy to use based on user input.

**7. Command Pattern**

* **Use Case**: To handle **order processing** steps (placing an order, sending notifications, confirming payments).
* **Why**: It allows you to queue and execute tasks in a systematic way (like an "undo" system) and is great for asynchronous task processing.
* **How**:
  + Each action (like PlaceOrder, NotifyCustomer, ChargePayment) is a command.
  + Use a **command queue** to ensure that actions are executed in the right order (e.g., don't send an order confirmation before charging the payment).
  + This is useful if you later want to use a **message queue (like RabbitMQ or AWS SQS)** for background processing.

**8. Dependency Injection (DI)**

* **Use Case**: For loosely coupling services, especially in tests.
* **Why**: It allows you to swap out implementations of dependencies for **mocks** or **stubs** during testing.
* **How**:
  + Inject dependencies (like Database, Logger, NotificationService) into classes (like **OrderController**) at runtime.
  + Use libraries like **Inversify.js** or **typedi** for dependency injection.

**9. Event-Driven Architecture (EDA)**

* **Use Case**: For **order status changes** and **asynchronous workflows**.
* **Why**: Instead of running everything synchronously, you emit events like "Order Created" or "Payment Confirmed" and let **consumers** (like user app, restaurant, and delivery driver) respond accordingly.
* **How**:
  + Use tools like **Redis Pub/Sub**, **RabbitMQ**, or **AWS EventBridge**.
  + For example, when an order is placed, emit an "OrderPlaced" event. The system listens for this event to send notifications, update dashboards, and start delivery tracking.

**10. Builder Pattern**

* **Use Case**: For building complex objects like **order objects**.
* **Why**: When creating an Order object, you may have optional fields like **coupon codes**, **discounts**, **custom delivery instructions**, etc.
* **How**:
  + Use a OrderBuilder class with methods like setCustomer(), setDeliveryAddress(), and addItem().
  + Return a complete Order object when the build is finished.

**11. CQRS (Command Query Responsibility Segregation)**

* **Use Case**: Separate **read operations** (like "get order status") from **write operations** (like "place order").
* **Why**: Improves **read performance** and simplifies **write logic**.
* **How**:
  + Use **read services** for queries (e.g., "Get all orders for this user").
  + Use **write services** to handle updates (e.g., "Update order status to 'delivered'").
  + You can optimize reads (caching) without affecting writes.

**12. Adapter Pattern**

* **Use Case**: To integrate third-party APIs (like Stripe, PayPal, or Google Maps).
* **Why**: If APIs change, you only need to update the adapter, not the whole system.
* **How**:
  + Create **PaymentAdapter** to abstract payment API differences.
  + Create **MapAdapter** to standardize Google Maps or Mapbox differences.