**✅ Backend Engineer Interview Prep – General Software Focus**

**🧠 Core Knowledge Areas to Focus On:**

| **Area** | **Key Focus** |
| --- | --- |
| **Microservices** | Clean architecture, REST APIs, service communication (HTTP & Kafka), isolation |
| **SQL Server** | Schema design, indexing, joins, stored procedures, performance |
| **Kafka** | Event-driven communication, topic modeling, producers/consumers, offset handling |
| **.NET Core** | Web API development, dependency injection, async/await, EF Core |
| **Architecture** | Scalability, availability, idempotency, eventual consistency, observability |
| **Security** | JWT, API Gateway, validation, rate limiting |

**🔧 System Design Challenges (Non-Gaming, Platform-Oriented)**

**1. Design a User Profile Management Microservice**

📌 **Scenario**: A service responsible for CRUD operations on user profiles across the system (e.g., for a SaaS platform).

**Discuss:**

* SQL Server schema design: Users, Preferences, ActivityLogs
* REST endpoints: GET /users/{id}, POST /users, etc.
* Async event publishing via Kafka when user data changes (user.updated event)
* Use DTOs, validation, layered architecture in .NET Core

**2. Design a Notification System**

📌 **Scenario**: A backend service that sends notifications (email, SMS, push) based on system events.

**Components:**

* Kafka: other services publish events like user.signup, invoice.generated
* Notification Service consumes relevant topics
* Stores notification status in SQL Server
* Exposes GET /notifications for user views

**Tech Tips:**

* Retry + DLQ (dead-letter queue) patterns for failed sends
* Use background services in .NET for async processing

**3. Design an Audit Logging System**

📌 **Scenario**: Collect and store events across the platform for compliance and debugging.

**Flow:**

* Microservices publish audit events to Kafka (e.g., user\_logged\_in, file\_uploaded)
* AuditService consumes events → stores in SQL Server
* Include metadata: userId, timestamp, request IP

**SQL Schema Tip:**

CREATE TABLE AuditLogs (

Id UNIQUEIDENTIFIER PRIMARY KEY,

EventType NVARCHAR(100),

UserId UNIQUEIDENTIFIER,

Timestamp DATETIME,

Payload NVARCHAR(MAX)

);

**4. Design an Order Management System (e.g., for e-commerce or internal tooling)**

📌 **Scenario**: Handles order creation, updates, and notifications.

**Microservices**:

* OrderService: Manages SQL Server persistence
* InventoryService: Reacts to order events via Kafka
* BillingService: Listens to order.created, processes payments

**Patterns**:

* Outbox Pattern for event publishing after DB insert
* Saga Pattern for distributed transaction coordination (order → payment → confirmation)

**5. Design a File Upload + Processing System**

📌 **Scenario**: A web app allows users to upload files which then need background processing (e.g., PDFs, reports).

**Flow**:

* FileUploadService saves metadata to SQL Server, file to blob storage
* Publishes file.uploaded event to Kafka
* FileProcessingService picks up, processes file, updates SQL

**Discuss**:

* Chunked uploads
* File validation
* Kafka offsets and retries

**💡 Example Kafka Event Payloads**

**User Created Event:**

{

"eventId": "guid",

"type": "user.created",

"userId": "123",

"email": "user@example.com",

"createdAt": "2025-07-02T13:00:00Z"

}

**Order Created Event:**

{

"orderId": "789",

"userId": "123",

"total": 199.99,

"items": ["sku1", "sku2"]

}

**🧠 Backend Interview Questions & Answers**

**1. How do you ensure reliable communication between services using Kafka?**

✅ **Answer**:

* Use **acknowledgements** from Kafka producers (acks=all)
* Track **offsets** in consumers to avoid duplicate processing
* Use **idempotent consumers** (e.g., check eventId in DB)
* Apply **Outbox pattern** to avoid message loss on DB failure

**2. How do you structure microservices in .NET Core?**

✅ **Answer**:

* Follow **Clean Architecture**: split Domain, Application, Infrastructure, API
* Use **Dependency Injection**, **Configuration**, and IHostedService for Kafka consumers
* Isolate service boundaries (e.g., UserService, OrderService, etc.)
* Use shared contracts (DTOs) or Protobuf when needed

**3. How do you handle schema changes in SQL Server safely?**

✅ **Answer**:

* Apply **backward-compatible changes** first (e.g., new columns, avoid breaking changes)
* Use **EF Core Migrations** + version control
* Use **feature toggles** and **blue-green deployments**
* Write safe rollbacks for critical changes

**4. How do you model and track failures in a Kafka-based system?**

✅ **Answer**:

* Retry with exponential backoff on consumer failures
* Send failed messages to **Dead-Letter Topics**
* Use correlation IDs for tracing
* Store failures in a SQL table for inspection

**5. What does a high-performance SQL Server table design look like?**

✅ **Answer**:

* Use **proper indexing** (covering indexes for key queries)
* Normalize only when needed; consider denormalization for reads
* Use **DATETIME2** over DATETIME for precision
* Keep narrow tables for high-write workloads

**📦 Key Patterns to Know**

| **Pattern** | **Description** |
| --- | --- |
| Outbox | Store events in DB before publishing to Kafka for reliability |
| Saga | Orchestrate distributed workflows across services |
| CQRS | Separate read/write models for scalability |
| Retry + DLQ | Make message processing fault-tolerant |
| Clean Architecture | Separate business logic from tech concerns |