TECHNICAL TEST – Back-End Engineer

Waktu pengerjaan: 3 hari

Instructions:

- **Language**: Write your code in Go (Golang) & SQL (preferred using PostgreSQL / MySQL dialect)
- **Clarifications**: If you need clarification on any part of the question, feel free to ask. Otherwise assume the requirements are complete.
- **Sending the Results:** Please send the answer to the test to hr@nexmedis.com and cc: tech@nexmedis.com

1. You are tasked with designing an API for an e-commerce platform. The system must support the following features:
    - User registration and authentication
    - Viewing and searching products
    - Adding items to a shopping cart
    - Completing a purchase

    Design the RESTful endpoints for the above features. Describe your choice of HTTP methods (GET, POST, PUT, DELETE), URL structure, and the expected response formats. Assume that users need to authenticate before performing certain actions (e.g., adding items to the cart).

    answer:

    ## Endpoints & HTTP Methods:

    1. **User Registration & Authentication**
        - `POST /api/v1/register` → Register a new user
        - `POST /api/v1/auth/login` → Authenticate user and return JWT token
        - `GET /api/v1/profile` → Get authenticated user details *(Requires authentication)*

2. **Category**
   - `GET /api/v1/categories` → Get a list of products with search
   - `POST /api/v1/categories` → store products

   - `GET /api/v1/categories/{id}` → Get product details by ID
3. **Product Management**

   - `GET /api/v1/products` → Get a list of products with search
   - `POST /api/v1/products` → store products

   - `GET /api/v1/products/{id}` → Get product details by ID

4. **Shopping Cart**

   - `POST /api/v1/cart` → Add an item to the cart *(Requires authentication)*

   - `GET /api/v1/cart` → View cart *(Requires authentication)*

   - `DELETE /api/v1/cart` → Remove an item from the cart *(Requires authentication)*

5. **Checkout & Orders**

   - `POST /api/v1/checkout` → Complete a purchase *(Requires authentication)*
   - `POST /api/v1/payments/pay` → Complete a Payment *(Requires authentication)*
   - *`GET /api/v1/orders` → View order history (Requires authentication)*

2. Consider a database table Users with the following columns:
   - id (Primary Key)
   - username
   - email
   - created_at

   Your task is to design an indexing strategy to optimize the following queries:
   1) Fetch a user by username.
   2) Fetch users who signed up after a certain date (created_at > "2023-01-01").
   3) Fetch a user by email.

   Explain which columns you would index, and whether composite indexes or individual indexes would be appropriate for each query. Discuss trade-offs in terms of read and write performance.

   ➔ Create a unique index for column username.

   Pros:

   Username is usually unique to each user, a unique index ensure no duplicate username and improves query performance (where username = ?)

   Cons:

   Increasing write overhead do to uniqueness check during insert and update

   ➔ Create a regular index for column created_at.

   Pros:

   Query filtering by (created_at > "2023-01-01") benefits from index scan to avoid full table scan and Index created_at speed up query (>, <, between) by allowing the database to find efficient relevan rows.

   Cons:

   Index maintenance adds a small overhead when inserting new records.

   ➔ Create a unique index for column email.

   Pros:

Emails are often unique and used for user authentication and lookups.
A unique index ensures faster lookups and prevents duplicate emails.
Queries using `WHERE email = ?` are highly optimized with an index.

Cons:

Slightly increases `INSERT/UPDATE` time due to uniqueness constraint checks.

3. You need to implement a function that simulates a bank account system. Multiple users can simultaneously access and update their account balance. Your system must ensure that concurrent access does not result in race conditions.
   Implement the function that:
   - Deposits money into an account.
   - Withdraws money from an account (ensuring there's enough balance).
   - Ensures thread-safety while handling concurrent deposits and withdrawals.

   answer:

4. Given a database table orders with the following schema:

```
CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT,
  product_id INT,
  order_date TIMESTAMP,
  amount DECIMAL(10, 2)
);

*) Assume that customer_id is indexed, but amount and order_date are not indexed.
```

   - Write an optimized SQL query to find the top 5 customers who spent the most money in the past month.
     answer:

     Select customer_id, SUM(amount) as total_spent
      from orders

```
Where order_date >= NOW()
Group by customer_Id
Order by total_spent DESC
Limit 5
```

- How would you improve the performance of this query in a production environment?

  answer
  Create indexing on column customer_id, order_date, and amount

5. You are tasked with refactoring a monolithic service that handles multiple responsibilities such as authentication, file uploads, and user data processing. The system has become slow and hard to maintain. How would you approach refactoring the service?
   - What steps would you take to decompose the service into smaller, more manageable services?
   - How would you ensure that the new system is backward compatible with the old one during the transition?

Step migrate from monolith to microservice

1. Identifying Domain Boundaries

   Separate services into Auth Service, User Service, Product Service, Order Service, etc.

2. Use an API Gateway as a Compatibility Layer

   The API Gateway will gradually route requests from the monolith to microservices.

3. Use a Database Per Service (If Necessary)

   For example, User Service can have its own database, separate from Order Service.

4. Use Event-Driven Architecture for Synchronization

   Example: When a user places an order, the Order Service can send an event to the Inventory Service.

5. Use Feature Flags for Gradual Deployment

   The monolith can still be used while some features run on microservices.