Go-lang interview

Technical Assessment

Contents

Exercise 1: Chat System (Go-lang)

This assessment is designed to evaluate your proficiency in Golang, distributed systems, cloud infrastructure, and related technologies as per the job description. You will be tasked with creating a microservice that simulates a simplified chat platform. This platform will handle user messages, store them in a distributed database, and ensure efficient retrieval and caching mechanisms. Additionally, you will be required to write tests and deploy the service using infrastructure as code.

Task Breakdown

1. **Microservice Development:**
   - Develop a microservice in Go that handles user authentication, message sending, and message retrieval.
   - Implement RESTful APIs for user registration, user login, sending messages, and retrieving message history.
   - Ensure the APIs follow best practices for design and documentation.

2. **Database Integration:**
   - Use Cassandra as the database to store user and message data.
   - Ensure efficient data storage and retrieval, handling distributed database concerns.

3. **Caching Mechanism:**
   - Integrate Redis as an in-memory cache to optimize data access for frequently requested data.
   - Implement appropriate cache invalidation strategies.

4. **Infrastructure as Code:**
   - Use Docker to containerize the application and the database.
   - Use nginx as an entry point for the API request.

5. **Monitoring and Performance (Plus):**
   - Implement monitoring using Grafana.
   - Ensure the system can log events, monitor performance, and handle troubleshooting efficiently.

6. **Testing (Plus):**
   - Write comprehensive unit and end-to-end tests to ensure reliability and stability.
   - Demonstrate tests for API endpoints and core functionality.

Detailed Instructions

1. **Microservice Implementation:**
   - **User Authentication:**
     - Implement endpoints for user registration (`/register`) and user login (`/login`).
     - Store user credentials securely.
   - **Messaging:**
     - Implement endpoints to send a message (`/send`) and retrieve message history (`/messages`).
     - Messages should include sender, recipient, timestamp, and content.

2. **Database Integration:**
   - Set up a Cassandra cluster (can be a local setup for the assessment).
   - Design a schema to store user information and messages.
   - Ensure the system can handle distributed data correctly and efficiently.

3. **Caching with Redis:**
   - Cache recent messages or frequently accessed user data.
   - Implement cache invalidation when necessary data changes occur.

4. **Containerization and Deployment:**
   - Create a Dockerfile to containerize your microservice.
   - Ensure all components (Cassandra, Redis) are included in the deployment scripts.

5. **Monitoring Setup (Plus):**
   - Configure Grafana for visualization.
   - Set up Prometheus for metrics collection **(Plus)**.
   - Use Loki for log aggregation **(Plus)**.
   - Ensure your service logs necessary events and performance metrics.

6. **Testing (Plus):**
   - Write unit tests for individual components and logic.
   - Write end-to-end tests to verify API functionality.
   - Ensure tests cover user registration, login, message sending, and message retrieval.

Submission Guidelines

- Provide a GitHub repository with the complete source code.
- Include a `README.md` file with instructions on how to build, run, and test the application.
- Ensure the repository includes all relevant Docker and configuration files.
- Include a brief explanation of your architectural decisions and any assumptions made during the implementation.

Exercise 2: Payment Processor (C++)

Develop a simplified payment processor that handles user accounts and transactions.

1. Implement an **Account** class with attributes **accountId**, **ownerName**, and **balance**.

2. Implement a **Transaction** class with attributes **transactionId**, **fromAccountId**, **toAccountId**, **amount**, and **timestamp**.

3. Implement a **PaymentProcessor** class with the following functionalities:

- void **createAccount**(const std::string& ownerName, double initialBalance): Create a new account.

- bool **processTransaction**(const std::string& fromAccountId, const std::string& toAccountId, double amount): Process a transaction between two accounts.

- double **getAccountBalance**(const std::string& accountId): Retrieve the balance of a specific account.

4. Demonstrate your work in a **main** function:

- Create two new accounts.

- Process at least one transaction.

- Display the current balance of each of the two accounts.

- Print the result of each function call to the screen.

5. Use **in-memory data structures** to store accounts and transactions.

6. Ensure **thread safety** for processing transactions (assume multi-threaded execution).

Make sure your code is readable and includes explanatory comments.

Evaluation Criteria

- **Code Quality:** Clarity, organization, and adherence to Go best practices.
- **Functionality:** Correctness and completeness of implemented features.
- **Performance:** Efficiency in handling database operations and caching.
- **Reliability:** Comprehensive testing and error handling.
- **Documentation:** Clarity and completeness of API documentation and setup instructions.
- **Deployment:** Correctness and completeness of infrastructure setup and deployment.

Good luck! We look forward to reviewing your submission.