Distributed systems assignment report

**Introduction**

This report outlines the implementation of a Joke Microservice, Submit microservice and moderate microservice. Joke microservice consists of three components: an application (joke), a message consumer (ETL), and a database (MySQL). Each component runs in its own Docker container, providing modularity, scalability, and portability. I implemented a basic web UI to request a single joke of a specific type selected from a list of available types. The UI includes a dropdown menu to select joke types, which is refreshed from the database on page reload or dropdown interaction. I developed the joke component using Node.js and Express.js. The Express server serves static content (HTML, CSS, and JavaScript files) for the web UI. I implemented two endpoints: ‘/type’ to return the current list of types from the database and ‘/joke’ to retrieve jokes based on specified query parameters for example, ‘/joke?type=programming’ the user will be able to see whatever query they have put in the URL. I deployed the Node.js server and application into a Docker container for portability and isolation. I also configured MySQL database to run in a separate Docker container and created a persistent volume to ensure MySQL database data survives container restarts or re-creations. I implemented an Extract Transform Load (ETL) application as a RabbitMQ queue consumer. The ETL application detects available jokes, reads them from the queue, and writes them into the MySQL database independently of the joke application. The ETL application runs on its own Node.js server deployed into a Docker container.

Submit Microservice, which consists of two components: an application (submit) and a RabbitMQ message broker configured as a basic queue (SUBMITTED\_JOKES). Both components are deployed in their own Docker containers, providing modularity and isolation. A basic web user interface has been developed to allow users to submit a new joke of a specific type selected from an up-to-date list of available types. The user interface includes a dropdown menu to select joke types, which is refreshed on page reload, dropdown click, or other interactions to reflect any new types added to the joke microservice database. The submit component runs on a Node.js Express server, providing endpoints for submitting jokes, retrieving types, and serving static content. The server serves HTML, CSS, and JavaScript client files as static content from the submit component's Node server, ensuring a seamless user experience. The submit component exposes three endpoints: ‘/types’: returns the current list of types to refresh the “types” list in the UI. ‘/sub’: receives the new joke posted from the UI and writes it to the message queue. ‘/docs’: returns openAPI compliant documentation for the endpoints of this microservice, facilitating easy integration and understanding of the API. In the event of the joke service being unavailable, the submit microservice implements a backup mechanism to ensure resilience. It maintains a backup copy of the types by storing them in a file on a Docker volume. This ensures that the microservice can continue to function seamlessly even if the joke microservice is down. To ensure that the types are always up-to-date, the submit microservice fetches the types from the joke microservice's ‘/types’ endpoint via an HTTP request. This integration ensures that the submit microservice always has the latest list of types available for selection.

Moderate Microservice, consisting an application (moderate) and a RabbitMQ message broker configured as a basic queue (MODERATED\_JOKES). Both components are deployed in separate Docker containers to ensure modularity and isolation. A basic web user interface has been developed to display a submitted joke read from the SUBMITTED\_JOKES queue. The UI presents the setup and punchline of the joke in editable web UI elements. Additionally, the joke type read from the queue is used to display its type in the types list, serving as the default selected type. The moderator has the flexibility to choose a different type from the dropdown, add a new type, submit or delete the joke as per their discretion. On page load, the moderator UI displays a new joke if one is available, with the option to delete the current joke and request a new one if desired. The moderate component runs on a Node.js Express server and serves static content (HTML, CSS, and JavaScript client files) from its node server. It exposes two endpoints: ‘/types’: returns the current list of types to refresh the types list in the UI. ‘/mod’: reads the SUBMITTED\_JOKES queue and returns a joke to the UI or displays a response indicating there are no jokes available. In case the joke service is unavailable, the moderator component maintains a backup copy of the types by storing them in a file on a Docker volume. This ensures the microservice's resilience to joke microservice failures. Authentication is required for moderators to gain access to the moderator microservice, ensuring secure access and preventing unauthorized usage. The types are fetched from the joke microservice's ‘/types’ endpoint via an HTTP request, ensuring that the moderator component always has an up-to-date list of types available for selection. All microservices are available from single IP address. Kong serves as the API gateway, providing three endpoints: ‘/joke/’: retrieves the UI and accesses the joke API. ‘/submit/’: retrieves the UI and accesses the submit API. ‘/mod/’ retrieves the UI and accesses the moderator API.

I believe I should receive a mark of 60%+ as I have completed all the requirements in the joke microservice, submit microservice and moderate microservice and all components work how they should. It provides jokes from the test producer, and I have deployed it to the cloud. I have also completed all the submission microservice requirements and I have deployed it to the cloud. I have completed all the moderate microservice requirements but not used the analytics microservice. All microservices work with each other.

**Design**

The current design of the microservices ecosystem exhibits several shortcomings in its design patterns, which could be improved for better scalability, resilience, and maintainability. Firstly, the architecture follows a monolithic approach where each microservice consists of multiple tightly coupled components. This can hinder scalability and agility, as changes to one component may require modifications across the entire service. Transitioning to a microservices architecture would offer a more modular and flexible approach, allowing each service to be developed, deployed, and scaled independently, thus enhancing agility and scalability. Another aspect of the design is the direct communication between microservices using HTTP requests. While this approach is common, it can introduce tight coupling and dependency between services. An alternative approach would be to adopt an event-driven architecture using message brokers like Kafka or RabbitMQ. This decouples services and enables asynchronous communication, improving scalability and fault tolerance by reducing dependencies between services. Furthermore, some microservices in the ecosystem handle multiple responsibilities, violating the Single Responsibility Principle (SRP). For instance, certain services serve static content, manage endpoints, and handle business logic within the same codebase. Applying SRP by separating concerns and ensuring each microservice has a single responsibility would improve maintainability and scalability by reducing complexity and dependencies. The design also relies on storing backup copies of critical data, such as types, in files on Docker volumes. While this provides a backup mechanism for resilience, it introduces complexity and potential inconsistency. An alternative approach would involve utilizing distributed databases or cache solutions like Redis to store critical data redundantly, ensuring data consistency, durability, and scalability without relying on file-based backups. Moreover, the current design mandates authentication for access to certain microservices, which may introduce overhead and complexity. A potential alternative would be to use a centralized authentication service or identity provider for authentication and authorization. This centralizes access control and simplifies management across microservices, enhancing security and scalability. Additionally, the design lacks explicit consideration for service discovery and load balancing, essential for managing distributed systems. Adopting container orchestration platforms like Kubernetes would automate deployment, scaling, and management of microservices, providing features such as service discovery, load balancing, and automatic rollouts, thus improving reliability and scalability. Finally, the design lacks explicit mention of logging and monitoring practices, crucial for troubleshooting and performance monitoring in distributed systems. Implementing centralized logging and monitoring solutions would enable proactive monitoring, debugging, and optimization of the system, enhancing its reliability and performance.

**Testing**

I conducted tests for individual components to ensure functionality and reliability after each requirement. I also performed integration tests to validate communication between frontend, backend, and database components. I utilized Thunder client for API testing to verify the correctness of endpoints and data retrieval.

|  |  |  |
| --- | --- | --- |
| **Action** | **Test** | **Working or not** |
| types | Is types called on UI? | yes |
| /type | When localhost:3000/type is requested, does the UI show with types? | yes |
| /joke | When localhost:3000/joke is requested, does the UI show with joke/s? | yes |
| /sub | Does /sub receive the new joke from the UI? | yes |
| /docs | Does /docs return swagger for the endpoints? | yes |
| /types | Does /types show that joke service is operational? | yes |
| /mod | Does /mod retrieve the UI and access the moderator API. | yes |