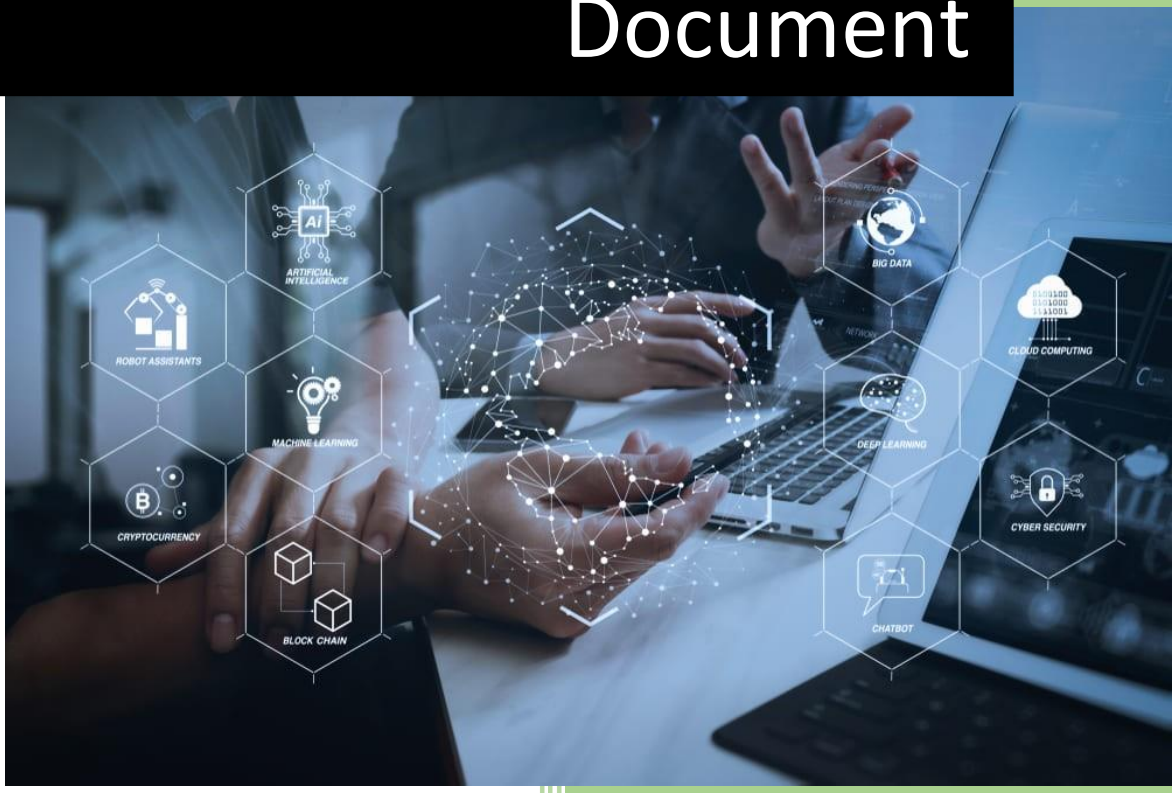
2024

Software Architecture Design Document



Vũ Phạm Thế

8/12/2024

RECORD OF CHANGE

No	*Effective Date*	*Version*	*Change Description*	*Reason*	*Updated By*
1	Augus-12-2024	1.0	Initial document	Create	VuPT

Contents

1. EXECUTIVE SUMMARY

This Software Architecture Document (SAD) provide a comprehensive architectural overview of the **Real-Time Vocabulary Quiz Application (RTVQA)**

This document describes the various aspects of the **RTVQA** design that are considered to be architecturally significant to support the key features and functions of the system that follow requirement. The architectural elements and principles in this document are fundamental for guiding the construction of the **RTVQA**.

2. REQUIREMENTS

Welcome to the Real-Time Quiz coding challenge! Your task is to create a technical solution for a real-time quiz feature for an English learning application. This feature will allow users to answer questions in real-time, compete with others, and see their scores updated live on a leaderboard.

Acceptance Criteria

1. **User Participation**:
    - Users should be able to join a quiz session using a unique quiz ID.
    - The system should support multiple users joining the same quiz session simultaneously.
2. **Real-Time Score Updates**:
    - As users submit answers, their scores should be updated in real-time.
    - The scoring system must be accurate and consistent.
3. **Real-Time Leaderboard**:
    - A leaderboard should display the current standings of all participants.
    - The leaderboard should update promptly as scores change.

Challenge Requirements

Part 1: System Design

1. **System Design Document**:
    - **Architecture Diagram**: Create an architecture diagram illustrating how different components of the system interact. This should include all components required for the feature, including the server, client applications, database, and any external services.
    - **Component Description**: Describe each component's role in the system.
    - **Data Flow**: Explain how data flows through the system from when a user joins a quiz to when the leaderboard is updated.
    - **Technologies and Tools**: List and justify the technologies and tools chosen for each component.

Part 2: Implementation

1. **Pick a Component**:

- Implement one of the core components below using the technologies that you are comfortable with. The rest of the system can be mocked using mock services or data.

2. **Requirements for the Implemented Component**:
   - **Real-time Quiz Participation**: Users should be able to join a quiz session using a unique quiz ID.
   - **Real-time Score Updates**: Users' scores should be updated in real-time as they submit answers.
   - **Real-time Leaderboard**: A leaderboard should display the current standings of all participants in real-time.
3. **Build For the Future**:
   - **Scalability**: Design and implement your component with scalability in mind. Consider how the system would handle a large number of users or quiz sessions. Discuss any trade-offs you made in your design and implementation.
   - **Performance**: Your component should perform well even under heavy load. Consider how you can optimize your code and your use of resources to ensure high performance.
   - **Reliability**: Your component should be reliable and handle errors gracefully. Consider how you can make your component resilient to failures.
   - **Maintainability**: Your code should be clean, well-organized, and easy to maintain. Consider how you can make it easy for other developers to understand and modify your code.
   - **Monitoring and Observability**: Discuss how you would monitor the performance of your component and diagnose issues. Consider how you can make your component observable.

3. ARCHITECTURAL PRINCIPLES

3.1.Principles

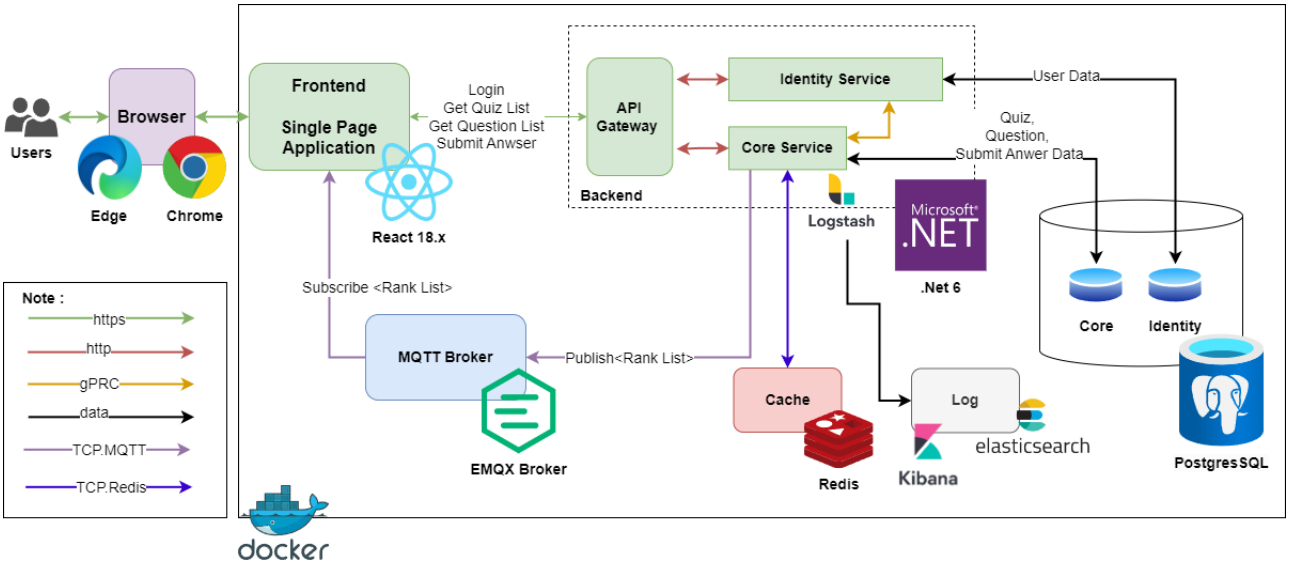
- **Performance Optimization: RTVQA** is built with modern microservice architecture with high scalability and resource optimization in mind to bring the best performance per price.
- **Flexibility: RTVQA** microservice ecosystem are easy to add new service and modify the current one with minimal impact to the other. The services can be built with various tech-stacks to adapt to the requirement.
- **Abstraction: RTVQA** microservices don't call or read each other's method or data directly, but these services are communicating via API calls and MQTT broker a clearly defined communication interface.
- **Scalability: RTVQA** services can be hosted using Azure Cloud Services which can be scaled up and out on the traffic demand.
- **Loose Coupling: RTVQA** microservice are developed and hosted independently and communicate via queue and API calls which follows the loosing coupling principle.

3.2.Architecture design approach

- **Backend**
  - **RTVQA** is developed using Microservices Architecture as proposed.
- **Frontend Architecture:**
  - Using Micro Frontend Framework (like single spa) for **RTVQA** front-end.
- **Database**
  - Database for RTVQA, which can be scaled out or up if needed.
- **Tool Support for Realtime Data**
  - EMQX MQTT Broker for **RTVQA**, which can be used to handle real-time data communication between Backend and frontend components. It enables efficient and scalable message distribution, allowing for real-time updates and interaction within the RTVQA system.
- **Deployment:**
  - Local with Docker for frontend and backend microservices.

4. OVERVIEW OF ARCHITECTURE

4.1.High Level Solution Architecture



Obsolete Full Architecture

4.2.Web Server Architecture

– **Web server will be built by .NET core and consisted of these layer:**

+ **API controller** : Expose the API for Website to call for getting UI/UX JSON content also handle the authentication & authorization via Identity Provider

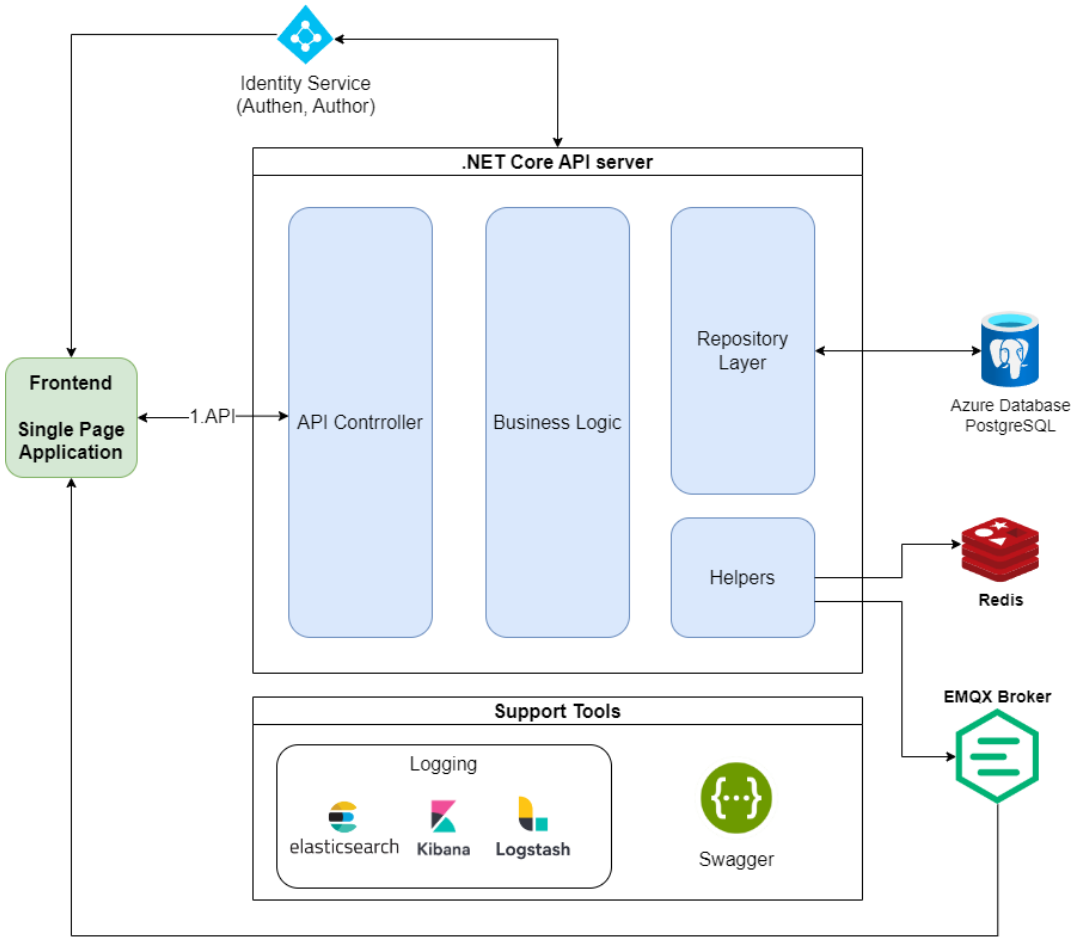
+ **Business Logic**: This layer will handle the business logic of the system for displaying the dashboard data or issuing the user operator based on input condition from the API controller

+ **Repository Layer** : this layer is for interacting with the database for creating / getting / updating / deleting data

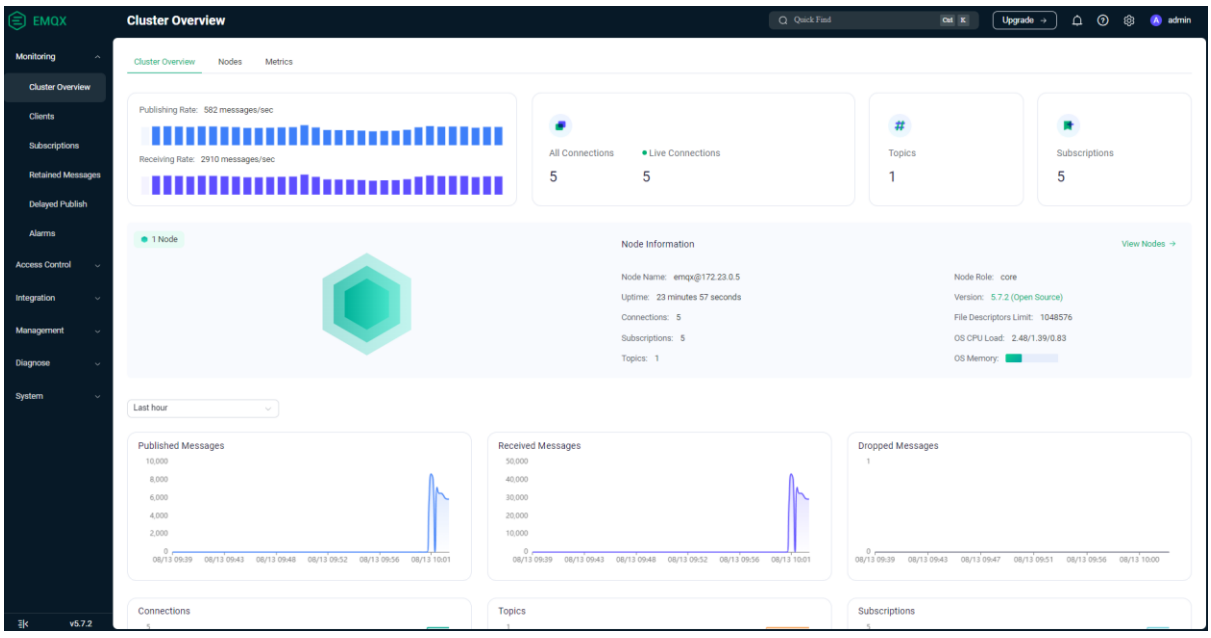
+ We also have the support tools layer (including logging & monitoring with **EKL**, publish <Rank List> data with **EMQX MQTT** Broker, Cache with **Redis**, OpenAPI like **Swagger**) and CICD layer (**GitHub Action**)

+ **EMQX** MQTT broker is a high-performance, scalable, and open-source MQTT message broker that facilitates the real-time exchange of messages between devices and applications. It supports the MQTT protocol, which is widely used in IoT (Internet of Things) environments for lightweight and efficient communication. In **your system**, **EMQX** is used to publish and manage <Rank List> data, enabling real-time updates and communication across connected client.

+ **EMQX** support Monitoring Dashboard. It's a valuable tool for overseeing the performance and health of the MQTT system, offering key features.



Web Server Architecture



EMQX Monitoring Dashboard

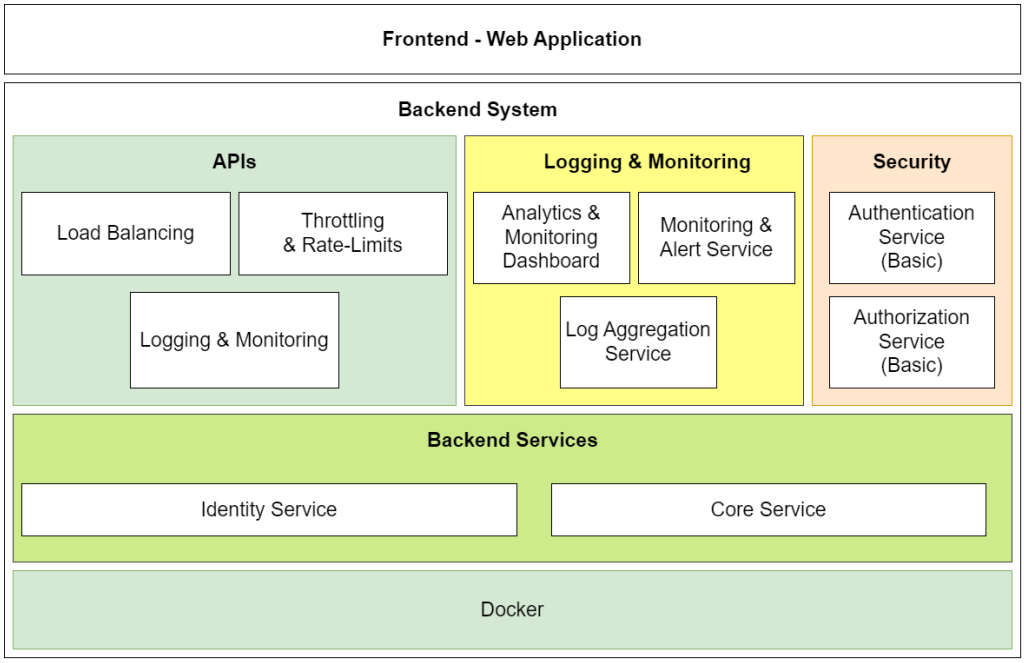
4.3.Conceptual Architecture

The **RTVQA** will be build based on **Client-Server Architecture** with Frontend (Client) and Backend (Server) layers separated to make the system decoupled and easy to extend and scale.

The Backend System will be modularized and grouped by functionalities to make features and functions of the system decoupled, which will help the system to easily adapt, reduce impact of changes and be easy to extend to add new features in future. The backend system of the **RTVQA** is comprised of:

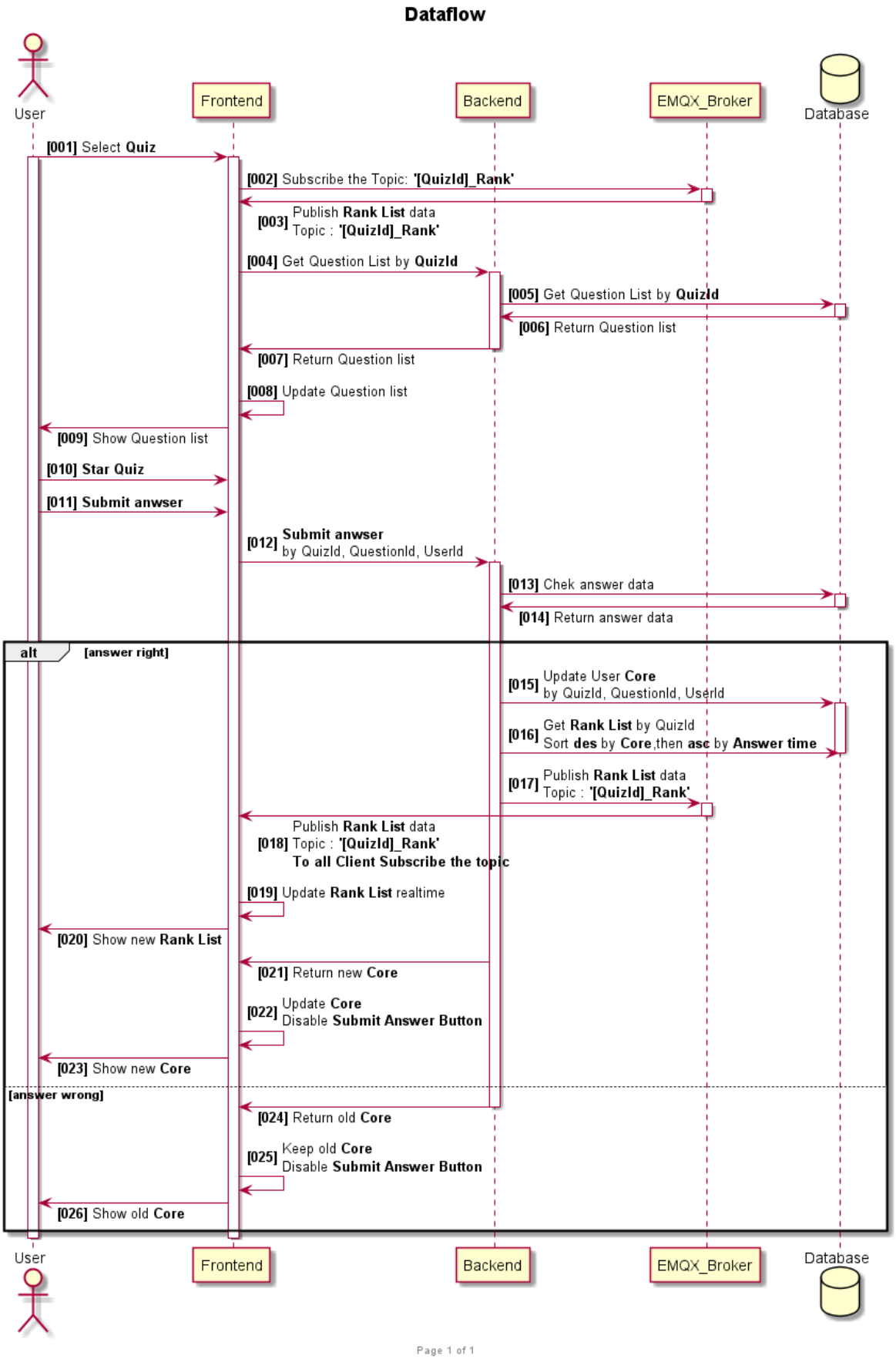
- **Backend Services**: built using Microservices architecture, each microservice will handle a specific function. This allows easy scaling and extension of the system in the future
- **APIs**: integrate/connect to the frontend application
- **Logging /Monitoring**: track/monitor all events that happen in the system to provide information for inspection/investigation when required
- **Security**: authorization/authentication management to control access to the system

The Frontend Web Application will be built as a Single Page Application (SPA). The SPA technology helps the web application interact with the user by dynamically re-rendering the current web page with new data from the web server, instead of the default method of a web browser loading entire new pages. It helps transitions faster that make the website feel more like a native application.



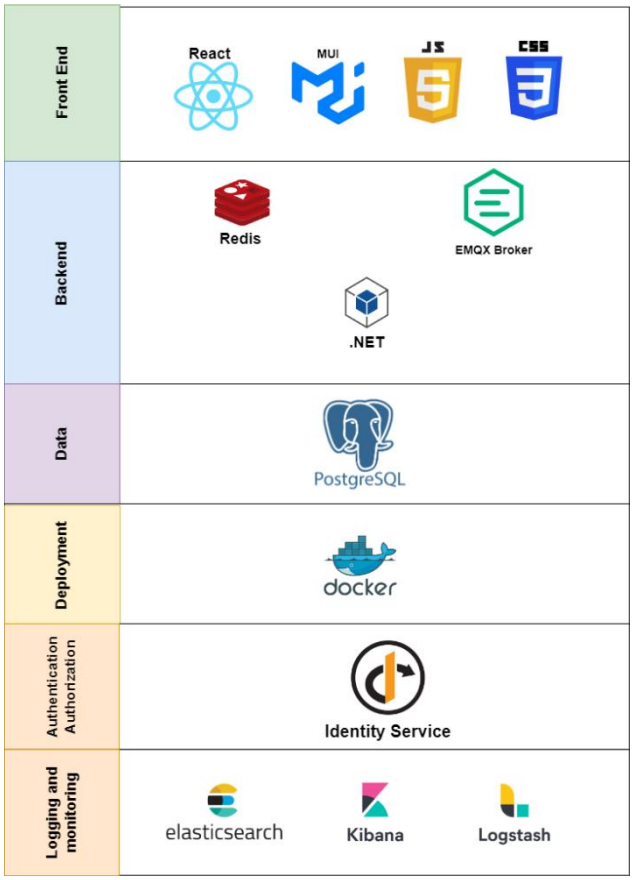
Conceptual Architecture Diagram

4.4.Dataflow



Page 1 of 1

4.5.Technical Stack



Technical Stack

5. APPENDIX

5.1. Technology Stack Information

Technology	Version	License	Description	References	Need license ?
.NET	6.0.7	Micros oft	.NET is a framework used to develop custom backend services	https://dotnet.microsoft.com/en-us/	Free
React	v 18.2.0	MIT License	React is a frontend framework used to develop web frontend application	https://react.dev	Free
Redis	7.0.4	BSD licenses	Redis (Remote Dictionary Server) is an in-memory data structure store, used as a distributed, in-memory key–value database, cache and message	https://redis.io/	Free

			broker, with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indices.		
EMQX	5.7.2	Apache License 2.0	EMQX Broker is the one MQTT Platform	EMQX github repo	Free
Ocelot	18.0.0	MIT	Ocelot is an API Gateway. It is aimed at people using .NET running a micro services / service orientated architecture that need a unified point of entry into their system.	https://ocelot.readthedocs.io/en/latest/index.html	Free
MediatR	7.0.0	Apache License 2.0	Supports request/response, commands, queries, notifications and events, synchronous and async with intelligent dispatching via C# generic variance.	https://www.nuget.org/packages/mediatr	Free

5.2. Implementation of Security

In order to comply with the standards, the platform will have certain features and will apply certain policies as below:

The platform must be secured from source code to funtionalities used by users via web application.

- The platform will comply with best practices of Open Web Application Security Project (OWASP) and will strictly follow OWASP Application Security Verification Standard (ASVS).
- The source code will be scanned by Static Application Security Testing (SAST) tool

- **In The feature the system must implement these security behaviors:**
  - All connection will use HTTPS
  - All sensitive data will be encrypted
  - Data in transit will be protected using Secure Socket Layer/Transport Layer Security (SSL/TLS) or client-side encryption (TLS version 1.2)