Technical Design

TLC Quantum

Table of Contents

Introduction

In the technical design, we will go over the more technical parts of the quantum chess tournament system, such as the C4 architecture diagram, sequence diagrams, and code analysis.

C4 Architecture Diagram

C1: System Context

Stretch: System, roles, external applications.
Audience: Every role in the system, tournament participants to admins.

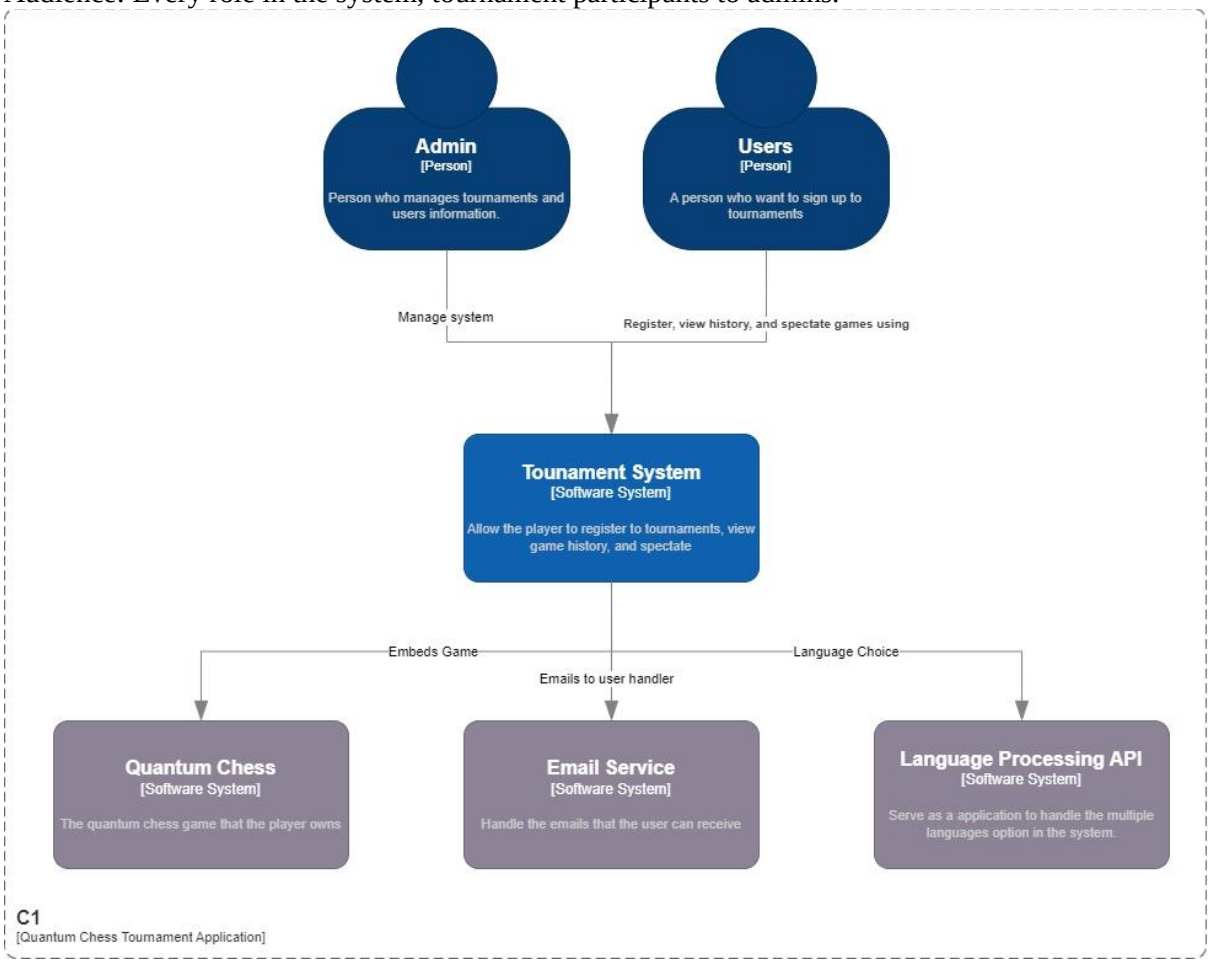

Figure 1: C1 Diagram

Description

Admin: The persons who moderate the application. They can create the tournaments, manage users, and handle complains inside of the system.

Users: The people who participate inside of the tournaments.

Tournament System: This will provide the tournament application.

C2: Container, System Visualization

Stretch: Container relations
Audience: Technical People

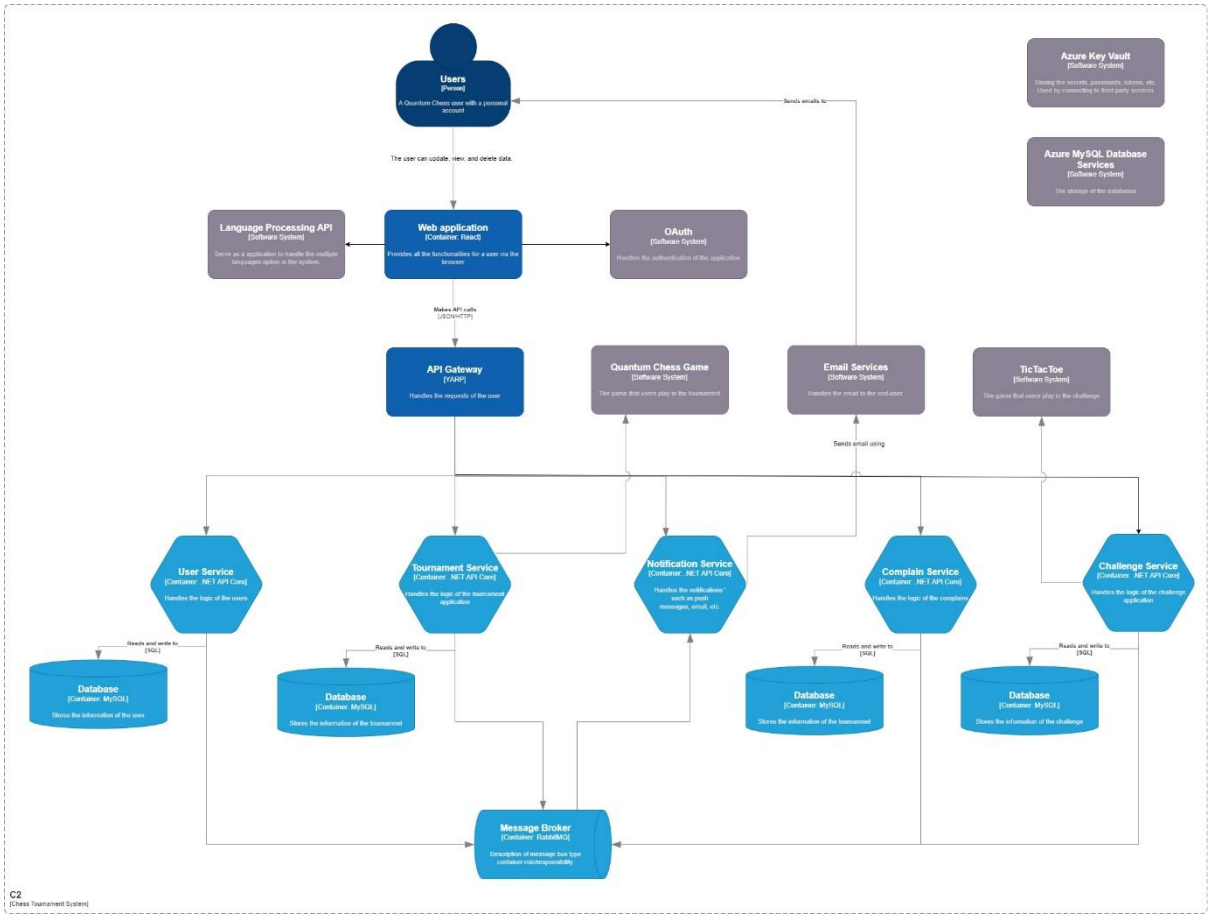

Figure 2: C2 Diagram

Description

Users: All the roles.
Web application: Front end of the application. React is being used as framework and will communicate through API requests.
API Gateway: Serves as a middle layer that receives requests from the front end (entry points), and based on the request will redirect to the correct microservice to handle the request securely.
Quantum Chess Game: The game that will be played in the tournament.
Email Service: Process the emails that need to be sent to the user from the service.
Message Broker: This will be a communication point where services can communicate with each other through the use of publish and subscribe to events.
Database: Stores all the relevant data of the specific service. Each service has its database, which improves loose coupling, independent services, and privacy.
Services: Each service serves a business case inside of the application.

C3: Communication between Components

Stretch: Components Relation
Audience: Technical Users

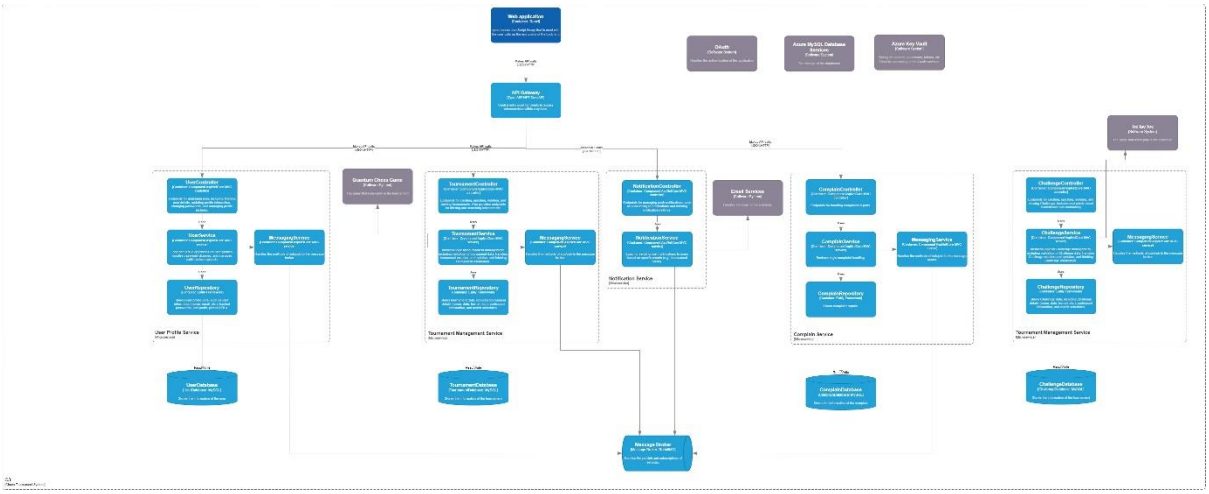

Figure 3: C3 Diagram

Separation of Components

The difference between C3 and C2 is that we now have a separate overview of the services. Since SOLID is applied, we want to ensure the single responsibility principle is applied, so that only services and components in the layer have only one reason to change. So in each service, we have a controller, service, and repository.

Inside Service flow

- The controller will receive an API call with a JSON/HTTP. This handles different CRUD operations. To handle the call, it will then go to the service.
- The service layer will check the validation of the request.
- When approved the logic will call the repository layer, where which then will write to the database.

Broad Overview

- User makes a request by interacting with the application.
- Request will be handled by the gateway, and check authentication and authorization, which will then be correctly redirected to a service.
- The service will handle the request and use the database for needed data.

Why SOLID

With the use of SOLID, we will help extend the code base to be sustainable, which is also easier to manage. This will allow us to use a codebase as a team, by applying the principle will improve coordination. (SOLID-ify Your Code With Code With SOLID Design Principles, n.d.)

C4: Code Structure

Stretch: Single-Classes
Audience: Technical Users (software developers and designers)

This is the main structure inside of the service, regarding the code. This structure is applied in each in microservice.

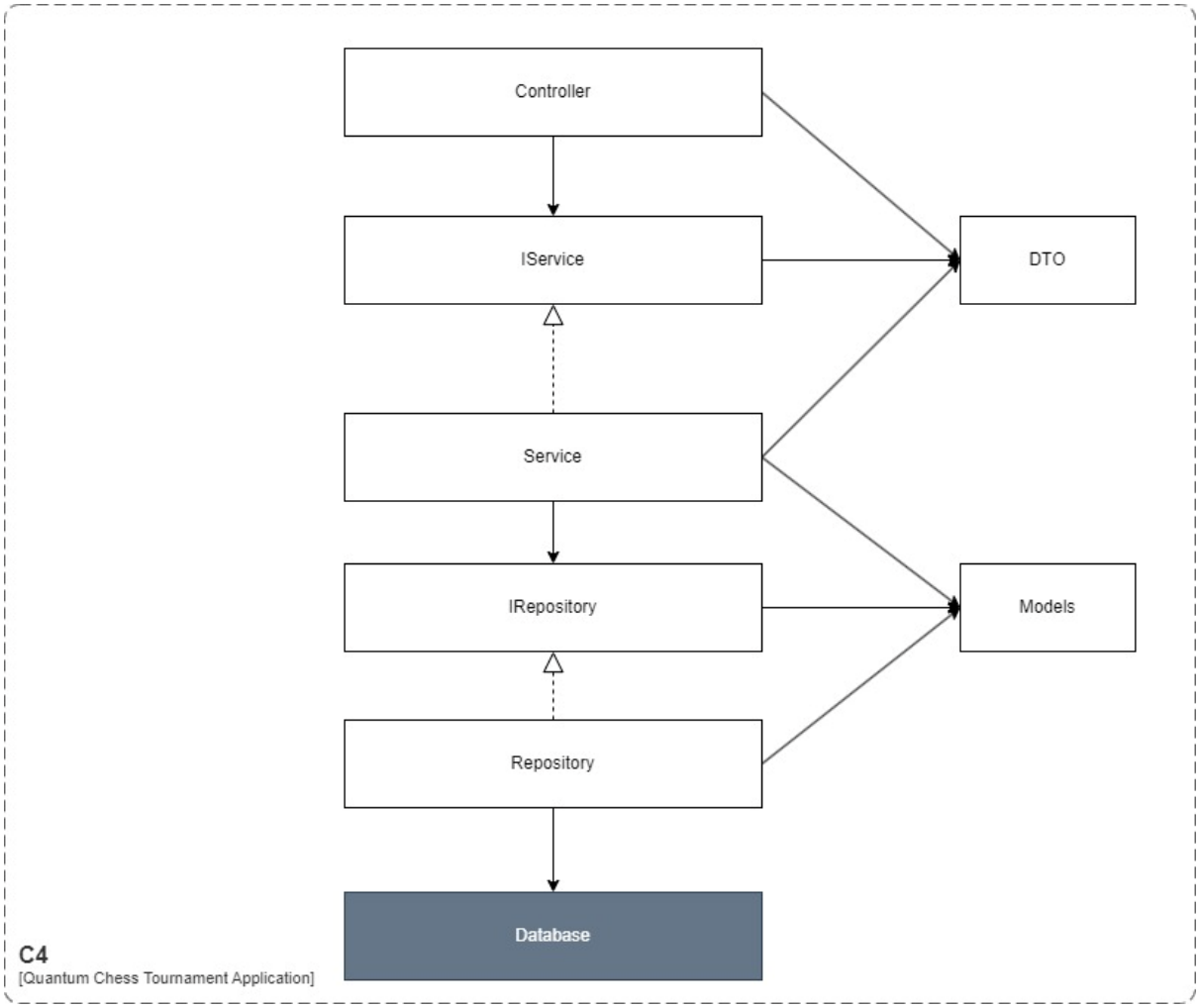

Figure 4: C4 Diagram

Message Broker

The message broker that we are going to use is RabbitMQ. We chose this because it is user-friendly, is open source, supports various standardized protocols, scalable and flexible, and is high in performance. (VINKA, 2020)
It will be mainly used as an interface between two services, to make it possible so that they can communicate with each other. A service can either send or receive a message to the RabbitMQ message broker, where services that need to receive the message can get it.
See the following sequence diagram to have an idea of the flow of how a message broker works between two services. Note that one service subscribes (Consumes) and one publishes (Produce) a message.

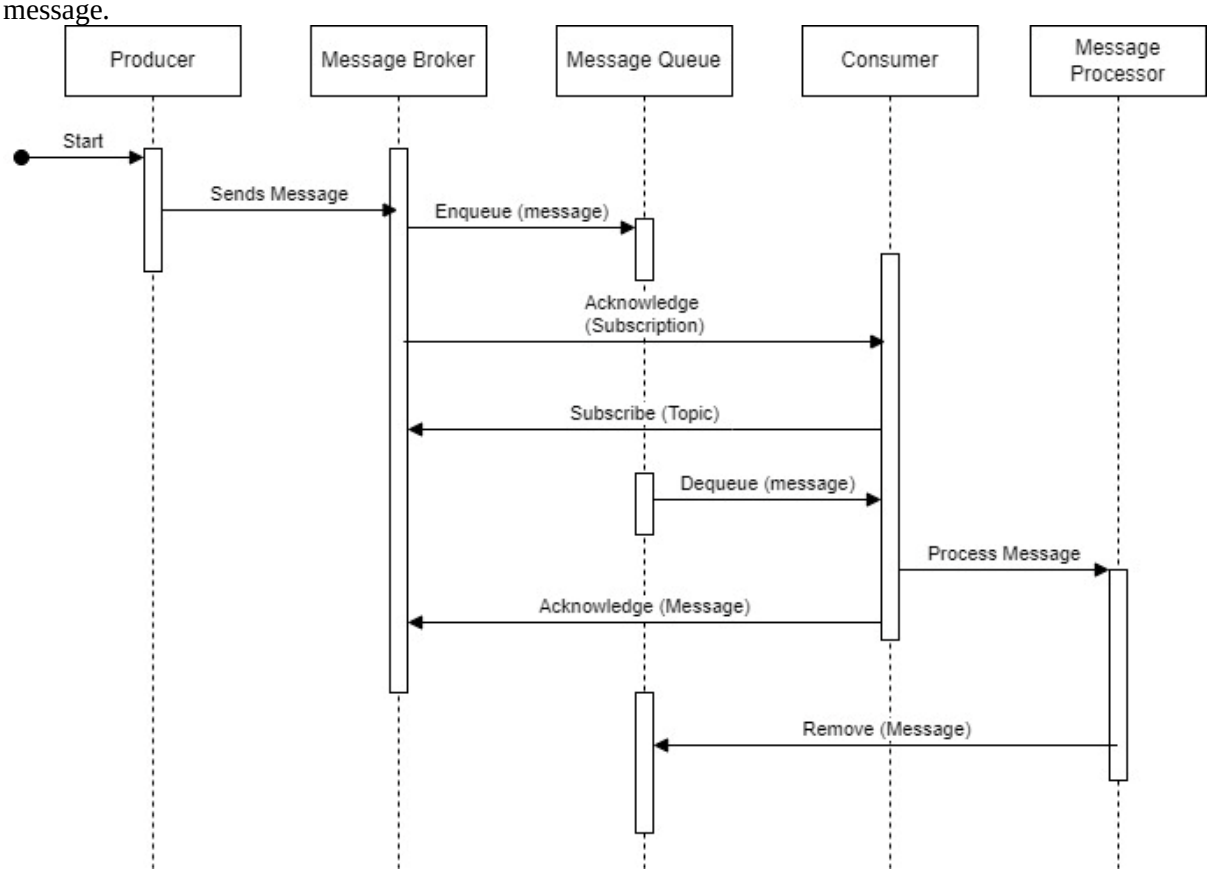

Figure 5: Sequence Diagram Message Broker

Load balancing

This method can be used so that we distribute the network's traffic, over an equal number of resources. Furthermore, FestivalConnect needs to handle a lot of traffic, we will have many servers with duplicate data, where the load balancer will evenly the traffic. This will improve the following:

- **Availability**: When a server is down, the load balancer will increase the fault tolerance, by also providing auto-detecting server problems and redirecting requests to available servers.
- **Scalability**: FestivalConnect can scale horizontally because traffic can be spread among multiple servers.
- **Security**: Load balancing will introduce an extra layer of security. It can deal with DDoS attacks, but also it provides monitoring traffic and automatically redirects attack traffic to specific backend servers which will have minimal impact on the system.
- **Performance**: Distributing the load will improve the response time and reduce the latency.

FestivalConnect can choose different algorithms to effectively use a load balancer, such as static load balancing, where you have limited rules to the load, such as round-robin or

weighted round-robin method. Moreover, you can also choose to have a dynamic load balancing, which will look at the current state of traffic for the servers, which will make it possible to choose from servers that have the least traffic at the moment, etc. (What is Load Balancing?, n.d.)

Rate limiting

With rate limiting we can make it possible to set a maximum of requests for the client which will prevent overload, and ensure there is stability and security. It will improve the performance since it can protect FestivalConnect against threats such as DoS attacks. (Microservices Rate Limiting, 2023)

References

Microservices Rate Limiting. (2023, 09 27). Retrieved from appmaster: https://appmaster.io/glossary/microservices-rate-limiting

SOLID-ify Your Code With Code With SOLID Design Principles. (n.d.). Retrieved from komododigital: https://www.komododigital.co.uk/insights/solid-ify-your-code-with-solid-design-principles/

VINKA, E. (2020, 04 03). *Microservices - why use RabbitMQ?* Retrieved from cloudamqp: https://www.cloudamqp.com/blog/why-use-rabbitmq-in-a-microservice-architecture.html

What is Load Balancing? (n.d.). Retrieved from aws.amazon: https://aws.amazon.com/what-is/load-balancing/#:~:text=Load%20balancers%20improve%20application%20performance,closer%20server%20to%20reduce%20latency