**Design Document**

***Knockout Ticket***

|  |
| --- |
| **Date : 19/06/2024** |
| **Version : 4.1** |
| **State : Final Phase** |
| **Author : Valentin-Gabriel Morisca** |

Contents

[1. Introduction 3](#_Toc167286257)

[2. Architecture Constraints and Design Decisions 4](#_Toc167286258)

[3. C4 Model 5](#_Toc167286259)

[4. Principles Applied 9](#_Toc167286260)

[5. Gitlab Pipeline, SonarQube + JaCoCo 10](#_Toc167286261)

[6. Access Token Storage Trade-offs 11](#_Toc167286262)

[7. Conclusion 12](#_Toc167286263)

# Introduction

This document describes the software architecture for our project, detailing the design decisions made, the constraints we operated under, and the principles we have followed.

# Architecture Constraints and Design Decisions

**Why Spring Boot over other Java frameworks like Java EE or Micronaut?**

* **Rapid Development:** Spring Boot's defaults enable quick project starts without the need for extensive configuration.
* **Rich Ecosystem:** It integrates seamlessly with various Spring projects like Spring Security, Spring Data, etc. Micronaut does offer integrations but the ecosystem around Spring Boot is more mature, and more developed.
* **Maintainability & Scalability:** Spring Boot provides a solid base for creating scalable and maintainable solutions for software applications.

**Why React over Vue.js or Angular?**

* **Component-Based:** React's architecture focuses on building reusable UI components. While Vue.js and Angular also support component-based development, React's ecosystem and hooks make component reuse and management more intuitive, and easy to use. Also, it encourages a rapid development.
* **Rich Ecosystem:** React has a massive community, resulting in extensive libraries and tools to extend its functionality. While Angular and Vue.js also have strong ecosystems, React's is larger due to its longer tenure and backing by Facebook.

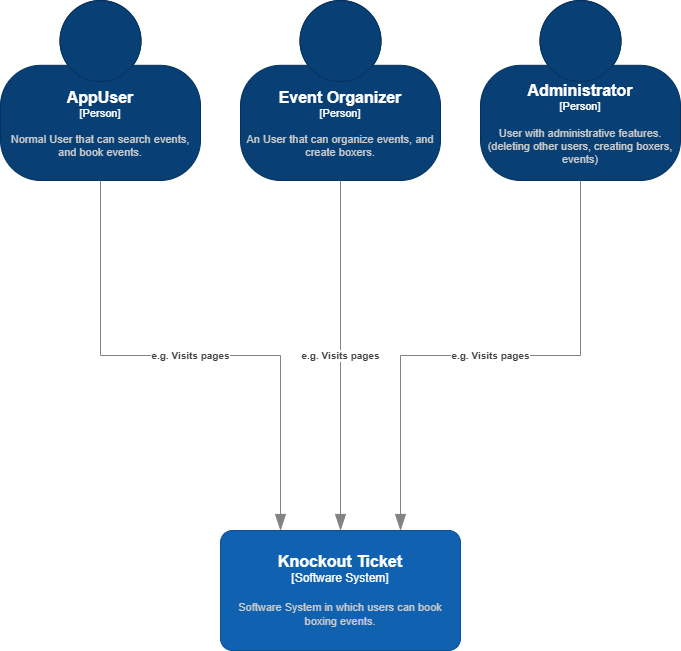
**Why MySQL over PostgreSQL or MongoDB?**

* **Popularity & Support:** MySQL is one of the most widely-used relational databases with extensive documentation and community support. While PostgreSQL is also popular and might offer some advanced features, MySQL's simplicity and support have made it a top choice for many.
* **Scalable:** MySQL can manage a significant amount of data without compromising on performance. PostgreSQL can also scale, but might require more configuration for high-performance scenarios. MongoDB, being NoSQL, scales horizontally but presents different data modeling and consistency trade-offs.

# C4 Model

**Level 1: System context**

This diagram represents how our system interacts with its users and external systems.



**Explanation:**

* Users interact with our system through the front end, built using React.
* The backend, developed using Spring Boot, manages the business logic and data storage.

**Level 2: Container**

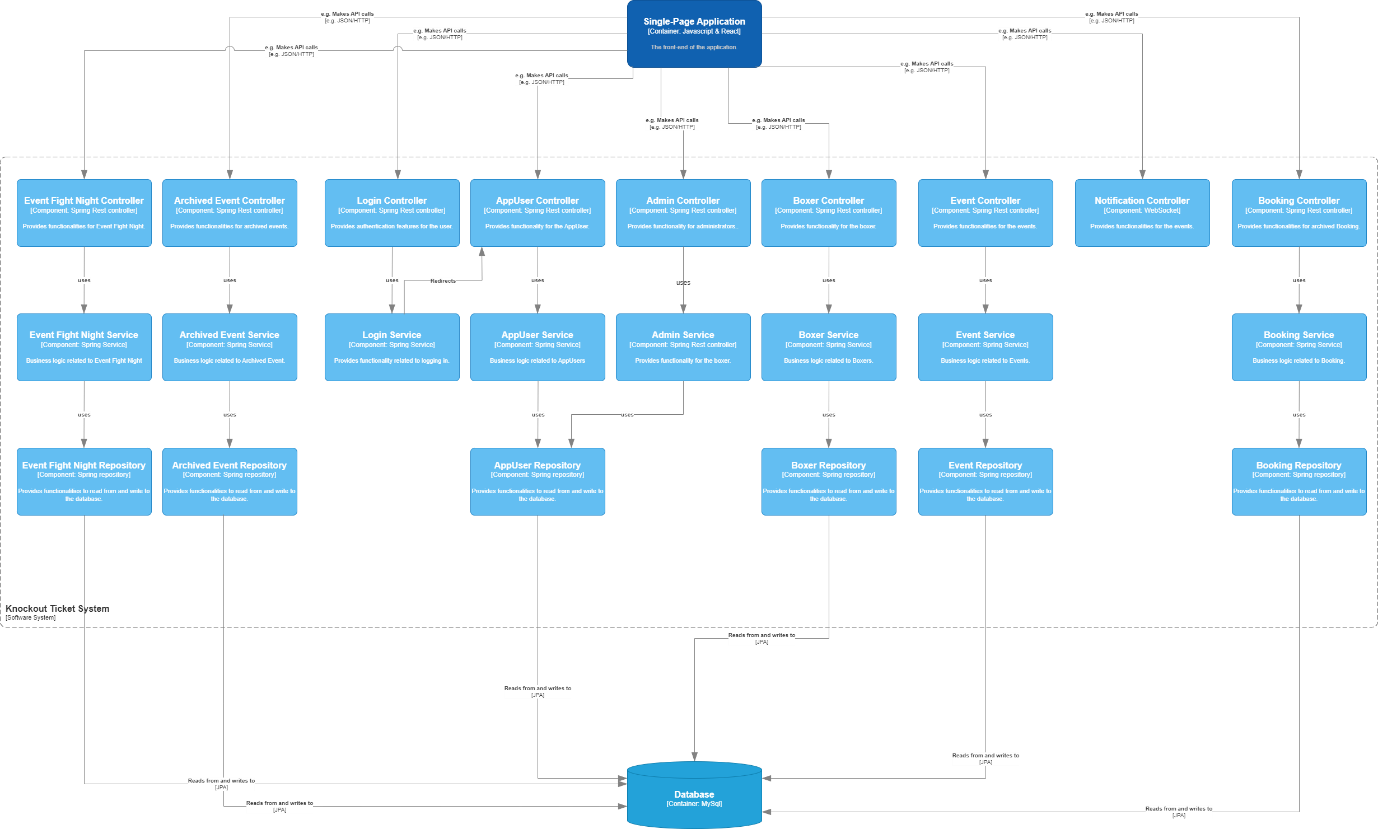
This level shows the high-level technical architecture of the system

A screenshot of a computer

Description automatically generated

**Explanation:**

* Web Application: This is our React application where users interact with the system.
* API Application: Our Spring Boot application which the front end communicates with to fetch, update, and delete data.
* Database: MySQL database for data persistence.
* **Level 3: Component**

****

**Explanation:**

* Controller Layer: The entry point for incoming requests. It delegates the work to the service layer.
* Service Layer: Where our business logic resides. It interacts with the repository layer to fetch and store data.
* Repository Layer: Directly interacts with the database, fetching and storing data.

Each layer has a distinct responsibility, ensuring we follow the Single Responsibility Principle from SOLID.

**Level 4: UML**

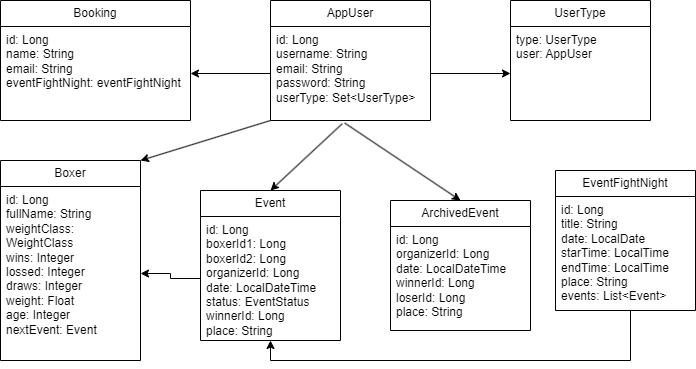
A diagram of a company

Description automatically generated

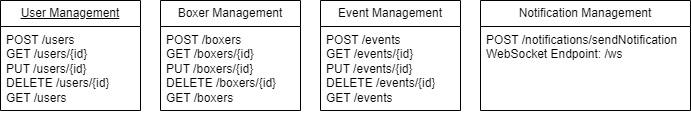
**Explanation:**

* Represents the static structure of a software application
* Classes define attributes and operations of objects.
* Relationships include associations between classes, inheritances, and dependencies.

**Domain Model Diagram**

****

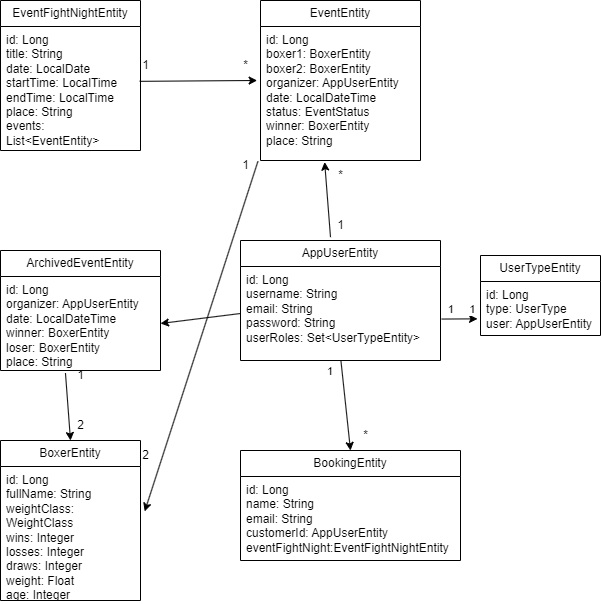
**Endpoints Diagram**

****

**A screenshot of a computer screen

Description automatically generated**

**Entity Relationship Diagram**

****

# **Principles Applied**

**SOLID Principles:**

* **Single Responsibility Principle (SRP):** Each class/component has a single responsibility. This is evident in our layered architecture where each layer has a defined role.
* **Open/Closed Principle (OCP):** Our system is open for extension but closed for modification. For instance, new features can be added as new services or components without altering existing ones.
* **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types. This ensures that any class can be replaced with a derived class without affecting behavior.
* **Interface Segregation Principle (ISP):** We've ensured that classes are not forced to implement interfaces they do not use. For instance, controllers only depend on the services they use, not on all available services.
* **Dependency Inversion Principle (DIP):** High-level modules (like services) do not depend on low-level modules (like repositories) but both depend on abstractions.

**KISS (Keep It Simple, Stupid) Principle:**

* **Description:** The KISS principle advocates for simplicity in design and code. Avoid unnecessary complexity and try to keep solutions straightforward and understandable.
* **Application:** In our system, we've avoided over-engineering solutions. For example, if a simple loop can solve the problem, we don't opt for more complex data structures or algorithms. Our focus is on clear, readable, and maintainable code.

**DRY (Don't Repeat Yourself) Principle:**

* **Description:** The DRY principle emphasizes reducing repetition in code. Duplication in logic or function can lead to maintenance challenges, inconsistencies, and bugs.
* **Application:** Throughout our application, common logic or functionalities have been abstracted into shared methods or components. For instance, instead of writing multiple functions to fetch data from different parts of the database, we've created a single, configurable data-fetching function. This not only reduces code redundancy but also ensures that any future changes can be made in one place.

**YAGNI (You Aren't Gonna Need It) Principle:**

* **Description:** YAGNI cautions against adding functionalities or complexities based on an assumption of future requirements. It's a reminder that one should only implement something when it's genuinely required.
* **Application:** In the development process, we've prioritized current, well-defined requirements over hypothetical future needs. For example, while it might seem tempting to build in advanced filtering and sorting for our data right now, if our current use case only requires basic listing, we stick to that. By doing so, we avoid unnecessary code and complexity, making the system easier to maintain and understand.

# Gitlab Pipeline, SonarQube + JaCoCo

**GitLab's** Continuous Integration/Continuous Deployment (CI/CD) pipeline is a part of GitLab that automates the process of software delivery and deployment. Automating these stages speeds up the development process, reduces the chance of human error, and ensures that changes are tested and deployed in a consistent manner.

**SonarQube** is a static code analysis tool used to identify bugs, vulnerabilities, and code smells in your codebase. SonarQube scans your source code and compares it against a set of coding rules. It then provides a detailed report of issues, which can be integrated into your GitLab pipeline. This way, every push or merge request can be analyzed automatically. It helps maintain code quality, ensures adherence to coding standards, and can prevent potential vulnerabilities from making it into production.

**JaCoCo** is a code coverage library for Java, which is particularly useful for measuring how much of your code is covered by unit tests. JaCoCo integrates with your build tools (like Maven or Gradle) and generates a report on the code coverage of your unit tests. This report can be published in your GitLab pipeline. It encourages developers to write more comprehensive tests, leading to more robust and reliable software.

A black background with white squares

Description automatically generated

# Access Token Storage Trade-offs

**Local Storage**

* **Persistence:** High. Data persists even after the browser is closed and reopened.
* **Security:** Moderate to low. Vulnerable to Cross-Site Scripting (XSS) attacks as the data is accessible via JavaScript.
* **Size:** Good. Allows for approximately 5MB of data, which is more than enough for tokens.
* **Network Overhead:** None. Stored data doesn't get sent with every HTTP request.
* **Scope:** Limited to the domain. Cannot be read or written across different domains.

**Session Storage**

* **Persistence:** Limited. Data is cleared when the page session ends (e.g., when the tab is closed).
* **Security:** Moderate to low. Similar to Local Storage, vulnerable to XSS attacks.
* **Size:** Good. Similar capacity to Local Storage.
* **Network Overhead:** None. Like Local Storage, it doesn't impact network traffic.
* **Scope:** Limited to the tab or window. Each tab/window has its own isolated storage.

**Cookies**

* **Persistence:** Configurable. Can be set to expire after a specific date or upon session end.
* **Security:** Higher with proper configuration. Can be set to HttpOnly to prevent JavaScript access (mitigating XSS risks) and Secure to ensure transmission over HTTPS. Still vulnerable to Cross-Site Request Forgery (CSRF) attacks.
* **Size:** Limited. Only allows for about 4KB of data.
* **Network Overhead:** High. Cookies are sent with every HTTP request to the server, increasing load.
* **Scope:** Configurable. Can be restricted to specific paths and domains.

**In-Memory (JavaScript Variables)**

* **Persistence:** Very Low. Data is lost on page reload or navigation.
* **Security:** High. Not accessible via XSS if implemented correctly (as it's not accessible via the DOM). However, if an attacker can run JavaScript, they can access in-memory data.
* **Size:** Depends on the browser's memory capacity, generally not a limitation for token storage.
* **Network Overhead:** None. Stored in memory, doesn't contribute to network load.
* **Scope:** Limited to the lifespan of the web page. Data is isolated to the page's JavaScript context.

# Conclusion

This architecture provides a robust, maintainable, and scalable solution for our project's requirements. By leveraging the strengths of Spring Boot, React, and MySQL, and adhering to the SOLID principles, we've established a foundation for sustainable growth and future expansion.