Software Architecture Document

(SAD)

**Table of Contents**

1. Introduction
2. System Context (C1)
3. Containers and tech choices (C2)
4. Components (C3)
5. UML Class diagram
6. Sequence diagram
7. CI/CD pipeline diagram

**Introduction**

In this document, I will describe the software architecture, explaining how it functions and the reasoning behind the design decisions. I will provide a detailed walkthrough of the C4 model, offering insights into each segment. The document will conclude with a sequence diagram and a CI/CD pipeline diagram.

**System Context (C1)**

**A diagram of a diagram

Description automatically generated**

**Containers and tech choices (C2)**

**A diagram of a diagram

Description automatically generated**

The FitSphere platform is built upon three primary tiers – Front-end (FE), Back-end (BE), and Database (DB). Here are a few reasons for that:

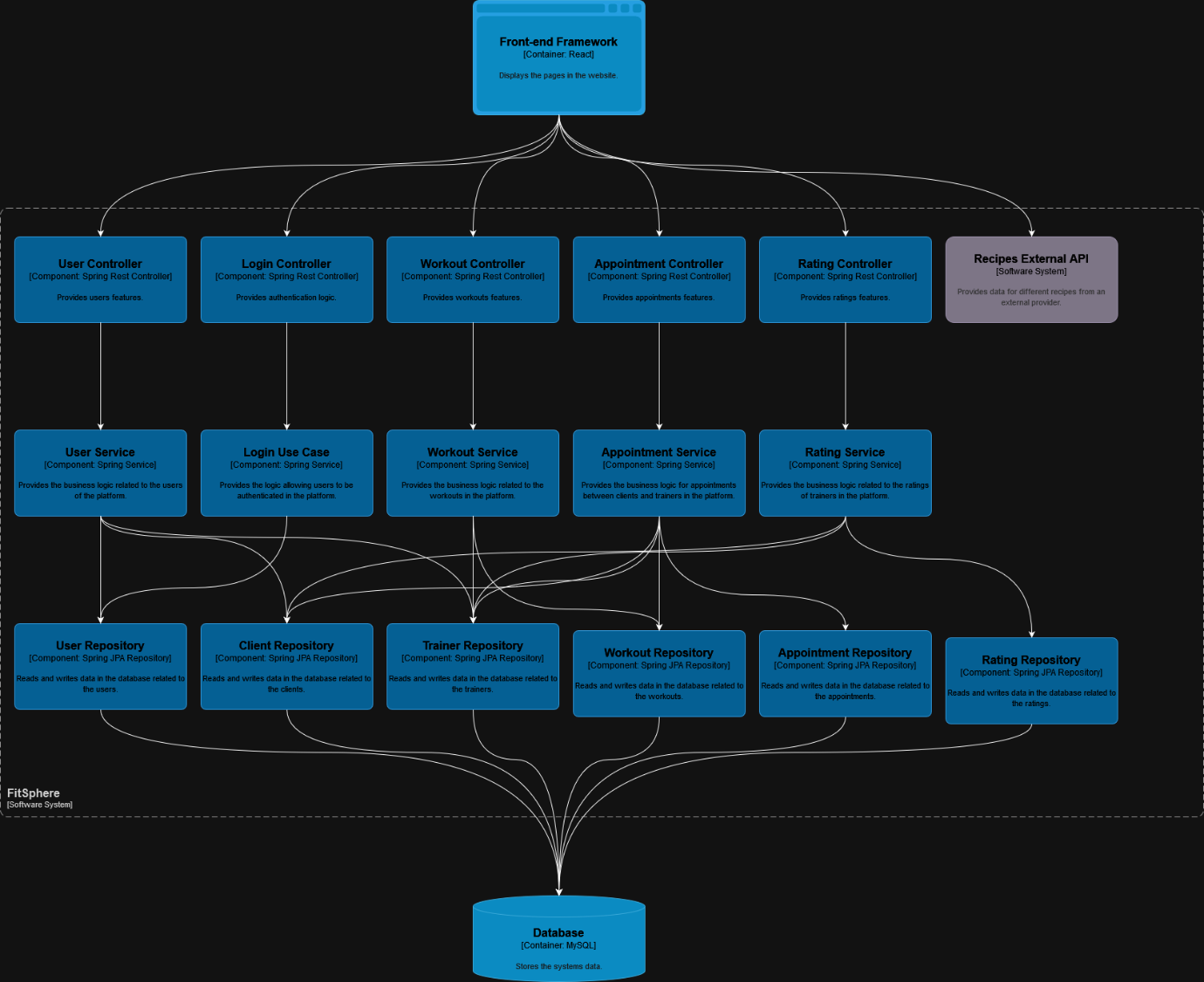
1. You can scale the application up and out. A separate backend tier, for example, allows you to deploy to a variety of databases instead of being locked into one particular technology. It also enables horizontal scaling by adding multiple web servers.
2. It simplifies the maintenance of the codebase by managing presentation code and business logic separately, so changes to the business logic do not impact the presentation layer.
3. It allows you to update the technology stack of one tier without impacting other areas of the application. For example, if I decide to switch from React to Angular, the backend and the database will still work seamlessly with the new frontend.
4. It enhances reliability and independence of the underlying servers or services.

For the FE, I opted for React. This decision was influenced by React's component-based code organization, which enhances readability and simplifies the process when dealing with numerous files. Additionally, the vast selection of libraries available and the supportive community further enrich its development capabilities. React's beginner-friendly attributes also played a role in this choice.

For the backend, I chose Java for the API due to its longstanding reputation of over twenty years, demonstrating both stability and maturity. Java-based applications, including APIs, can be used across multiple platforms without any changes. Moreover, Java provides a comprehensive collection of frameworks and libraries, such as Spring Boot, which facilitate API development.

For the database, I selected MySQL. One of its strengths is its ease of integration with Java-based applications. MySQL is renowned for its swift performance, particularly with read-heavy tasks. It is tailored for web applications and can manage a significant volume of concurrent connections. Furthermore, its compatibility with multiple platforms, such as Windows, Linux, and macOS, provides versatility in deployment.

**Components (C3)**

****

For the backend software architecture of FitSphere, I have opted to emphasize the use of service classes over use cases. This decision stems from my familiarity and comfort with this approach, as I find the use cases method to be somewhat overcomplicated for my project's needs.

The codebase is managed using the Spring Boot framework. The rationale behind choosing Spring Boot includes its efficient dependency management, fast build process, automation capabilities, reduced code requirements, and built-in modules. In the business layer, REST Controllers handle the sending and receiving of requests and responses. These requests are then passed to the service classes that manage the business logic, which subsequently interact with the database. This design adheres to SOLID principles, particularly promoting code separation and dependency inversion.

**UML Class Diagram**

This is the full image of the source code of the BE without the relations due to enormous picture (**below key features are explained**)

****

code of the BE without the relations

**Classes for the application features**

1. **Authentication**

This section details a core feature of the application: JWT token-based authentication and authorization, with future considerations for integrating OAuth. The JWT payload holds critical user details—Id, role, and expiration date—to streamline front-end performance and enhance security while keeping sensitive information secure.

The authentication process involves "LoginRequest" and "LoginResponse" classes. The "LoginRequest" class collects user credentials, which are validated by the "AuthenticationRequestRequestedFilter" on the server side. This includes password hashing and user verification, leading to the creation of an AccessToken delivered in a "LoginResponse."

"AccessTokenEncoderDecoder" manages token security, consolidating related functions into one class to maintain code integrity. The authentication architecture is further supported by "AuthService" and "AuthController," which are kept separate from user management to ensure clear separation of responsibilities and maintainable code. The "WebSecurityConfig" class handles CORS and HTTP security settings to regulate frontend-backend interactions and block unauthorized connections.

1. **Exception handler**

The "SystemExceptionHandler" class is dedicated to managing custom exceptions, which all derive from broader exceptions like "RuntimeException" and "ResponseStatusException." This implementation serves two main purposes: backend debugging and user-friendly messaging.

In the backend, " SystemExceptionHandler " aids in debugging by providing precise reasons for failures instead of generic exceptions that often lack detailed explanations. This helps in identifying and resolving issues more efficiently. "RuntimeException" and "ResponseStatusException" were chosen as parent exceptions because they effectively handle status failures and runtime exceptions.

Additionally, the exception messages are crafted to be informative for both the frontend and backend. In the frontend, these messages help users understand why a particular functionality isn't working or if incorrect data has been provided, enhancing the overall user experience. This dual functionality ensures that both developers and users receive clear, actionable information, improving both the debugging process and user satisfaction.

**CI/CD pipeline diagram**

A black grid with white text and icons

Description automatically generated with medium confidence

**Source Code**: The source code comprises all the programmatic instructions and dependencies required for my application, managed via version control systems to track and accommodate changes over time.

**GitLab**: GitLab serves as a centralized platform offering Git repository management, integrated CI/CD pipelines via `.gitlab-ci.yml`, and issue tracking, enhancing automation and collaboration.

**GitLab Runner**: This is a standalone service that works with GitLab CI/CD to execute jobs defined in your project's pipeline script, allowing for scalable and isolated builds on separate machines.

**Gradle**: Gradle is a build automation tool that uses Groovy or Kotlin-based DSLs for scripting, optimizing the build process for Java applications by managing dependencies and integrating development tools.

**Jacoco**: Jacoco is a plugin used within build automation tools like Gradle to generate detailed test coverage reports, helping developers identify untested parts of their Java code.

**Docker**: Docker facilitates the creation, deployment, and running of applications by using containers to ensure consistent environments across development, testing, and production stages.

**Docker Hub**: A cloud-based registry service where you can share and access Docker images, simplifying collaborations and deployments across different environments.

**SonarQube**: SonarQube analyzes and reviews code to identify bugs, vulnerabilities, and code smells, providing detailed reports on code quality.