Software Architecture Document (SAD)

For ChessNow Application

# General information:

|  |  |
| --- | --- |
| Date | 13-10-2023 |
| Version | 3.0 |
| State | In development |
| Author | Balan Mihai-Stefan |

# Distribution

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Author(s) | Changes | State |
| 1.0 | 13-10-2023 | Balan Mihai-Stefan | Initial document | Finished |
| 2.0 | 3-11-2023 | Balan Mihai-Stefan | Added C4 Diagram | Finished |
| 3.0 | 29-11-2023 | Balan Mihai-Stefan | Updated C4, added CI/CD diagram | Finished |

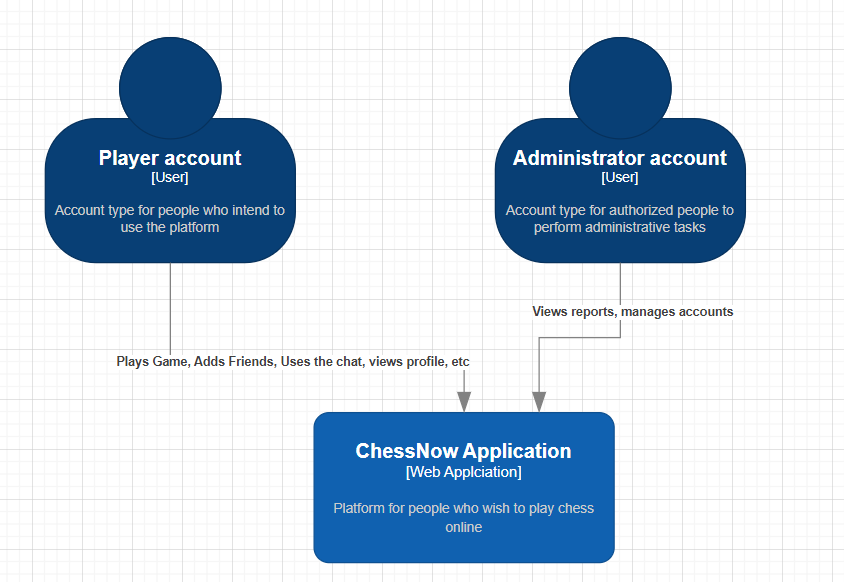
## Introduction

This document primarily focuses on delineating the software system's architectural aspects, a vital component of the broader agile software development process. The architecture serves as a guiding blueprint for the development team, elucidating the system's structural organization and the interplay of its diverse components. By encapsulating the architectural details within a Software Architecture Document (SAD), my objective is to offer a lucid and succinct representation of the system's design, alongside the rationale underpinning significant architectural choices. This approach facilitates ease of future maintenance.

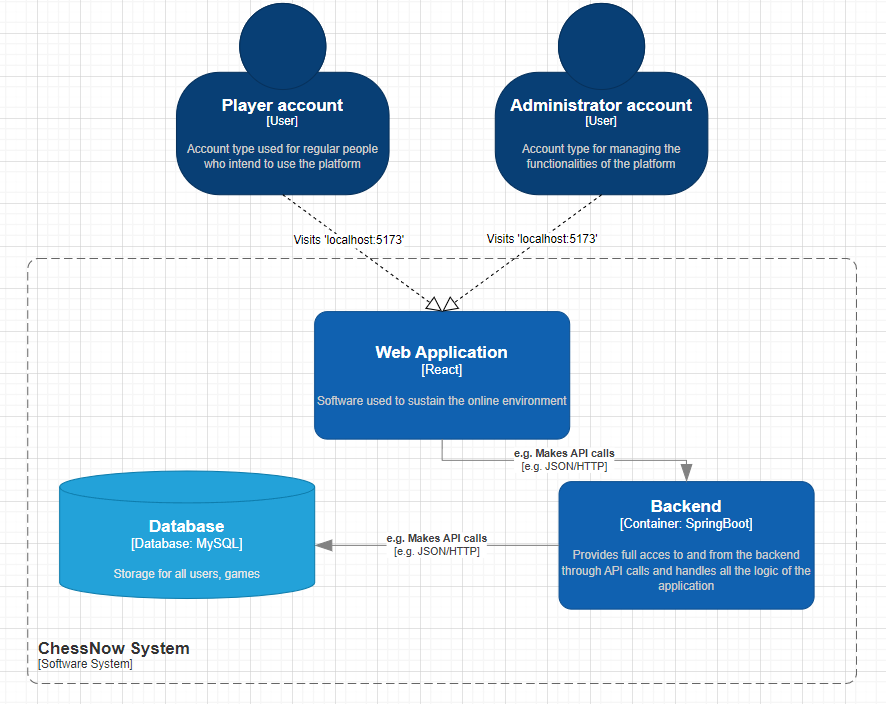
It's noteworthy that the SAD is a living document, subject to continuous evolution throughout the project's lifespan as the system's design undergoes modifications and refinements.

## Architecture diagrams

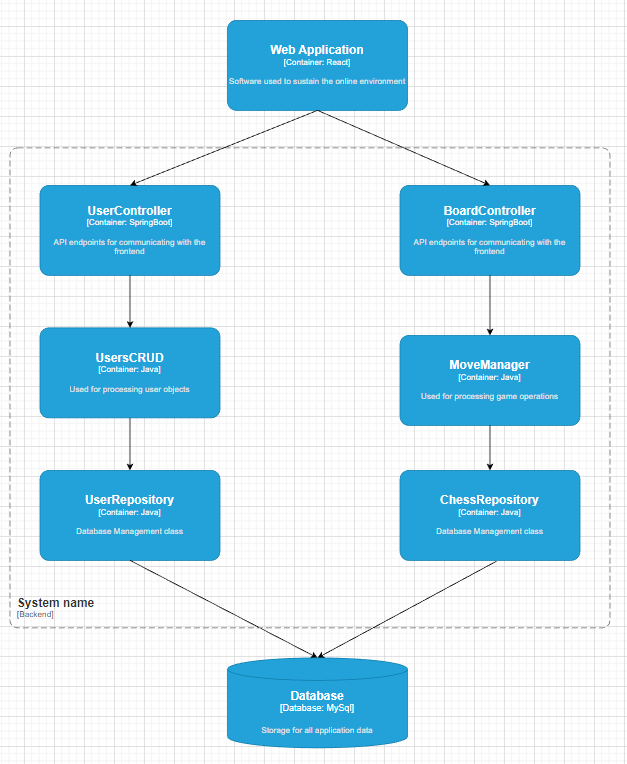
1. C1 Diagram – High level overview of the system’s design



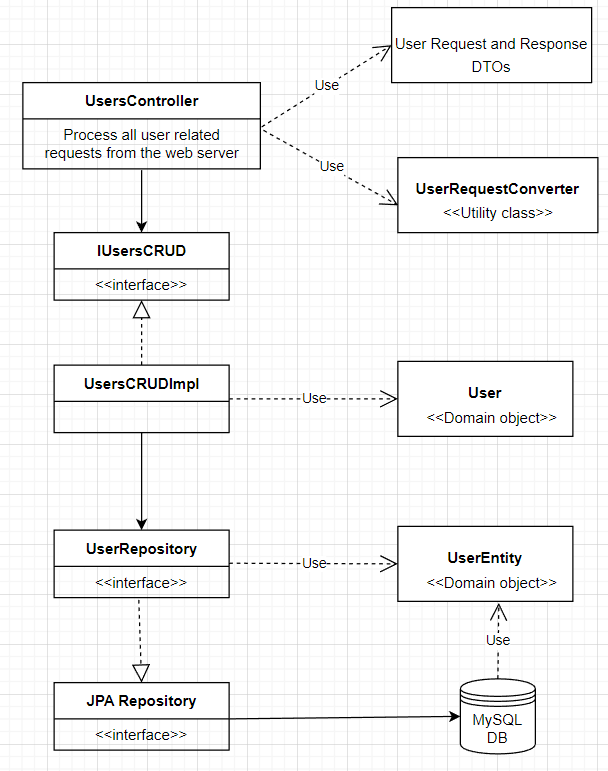
1. C2 Diagram – Zoom in on the software component of the C1 diagram



1. C3 Diagram – Greater detail on the architecture of the back-end of the application



1. C4 Diagram – In depth look at dependencies and relationships in the application



## SOLID Explanation

1. S – Single Responsibility Principle

The system I'm creating follows the Single Responsibility Principle, which means it's well-organized with distinct roles for different parts. In this design, we have separate layers and classes.

The Controller layer handles incoming HTTP requests, coordinates with business rules, and sends back the appropriate HTTP response, including status codes and data.

The Business layer is where the core operations and data validation happen. It includes manager classes and the interfaces these classes follow. This layer also provides access to the data storage layer.

The Persistence layer is in charge of storing data in an external database. Its primary responsibility is data storage and retrieval.

1. O – Open-Closed Principle

The system I'm designing adheres to the Open Closed Principle, mainly due to the structure of the Persistence layer. This layer is built upon an interface that encompasses CRUD (Create, Read, Update, Delete) operations. The classes responsible for interacting with the storage service implement this interface.

The interface lies at the heart of this design, making the layer open for extension. This means we can introduce new file storage systems without changing the existing code. Simultaneously, it's closed for modification because the interface ensures that every implementing class offers the required CRUD operations.

1. I – Interface Segregation Principle

The system abides by the Interface Segregation Principle because no client is forced to depend on methods it does not use. Each client of any interface implements all the methods of the interface it implements.

1. D – Dependency Inversion Principle

The system adheres to the Dependency Inversion Principle by utilizing interfaces to separate the architectural layers of the application. This design ensures that high-level modules are not reliant on low-level modules; instead, both rely on abstractions.

Additionally, this principle is instrumental in enabling effective unit testing, as it safeguards against inadvertently storing testing data in the database, promoting a clean and isolated testing environment.

## Technologies used

* Java and SpringBoot:
  + Platform independent language: The application can be deployed and used on any operating system
  + Security: Java has numerous libraries which can implemented and used for protecting the application from attacks like SQL injection, cross-site scripting, DDOS, etc.
  + Rapid development: SpringBoot provides a variety of tools that make the development process easier, meaning a RESTful API service can be built really quickly.
  + Compatibility: Other technologies like Hibernate ORM have integration have built in integration and support for SpringBoot.
  + Testing: SpringBoot features packages like Mockito, which can be used to create unit test for the APIs, ensuring the application functions properly.
* React.js:
  + Component based structure: Front-end code is structured in reusable components that do not depend on each other. This allows for the creation of high quality UI/UX
  + Compatibility with other chosen technologies: React features compatibility with SpringBoot built APIs, via the use of tools like Axios and HTTP requests.
  + Widely used language: The main language of this software is JavaScript, which a very popular language, who’s issues are well documented. There exist many blog posts about interesting things JavaScript can do for the front-end of the application, like animation, logic, data processing or data collection.

## Continuous Integration and Continuous Delivery

The diagram below explains how my CI/CD pipeline is set up and what stages my code goes through before being merged into the main branch and being deemed a ‘stable release’. The Quality Gate checks for coding style, naming conventions, bugs and test coverage.

