Design Document
for
FootballTix

By Stoyan Kostadinov(4092384)

Table of contents

1. Introduction
2. Application overview
3. Tools & Technologies
4. High level architecture

Revision #	Date	Reason for change
#1	6 Oct 2021	Initial version

Introduction

The purpose of this document is to outline the design for FootballTix. This document includes:

- High level description of what the application does
- Tools & Technologies used and the reasoning for choosing them
    - Spring Boot
    - Spring Security
    - JWT
    - ReactJS
    - JPA
    - MySQL
    - H2 for testing
- High level architecture and an explanation how SOLID is guaranteed
- C1, C2 and C3 diagrams

Application overview

FootballTix is an online ticket ordering system. Visitors can view upcoming matches and register when they want to order a ticket. Once they register and confirm their email, they can order the ticket for the match they want. Payments will be processed using iDeal. Once the payment is successful, they will receive their ticket via email. The ticket will contain a QR code which they will have to show at the entrance of the stadiums. The app will be targeted mainly at mobile users. Therefore, responsiveness and performance are the main targets.

Tools & Technologies

Spring Boot

Apart from the fact that Spring Boot should be used because it is a requirement for the current semester, a few pros can be pointed out:

- Easy start - Spring Boot comes with out-of-the-box functionality
- Auto-configuration - Spring Boot saves the hassle of coding and unnecessary configuration
- Less source code - annotations are easy to understand and save time from writing boilerplate code - i.e getters and setters
- Ease of dependency management - installing a dependency is as simple as including a one line in the pom.xml file or build.gradle

Spring Security

Spring Security is the authentication & authorization framework for spring boot. It is proven to be reliable, since it is extensively tested. Using it is a no-brainer since implementing such a system from scratch is a tremendous amount of effort.

JWT

The app is mainly targeted at mobile users and JSON Web Tokens do not use cookies which is very mobile friendly. Their performance is superior, because the network round trip times are shorter. And the most important advantage is that they are decentralized, which means they can be

generated anywhere and authenticated anywhere when, for example, using microservices.

ReactJS
Out of the three most-popular front-end frameworks(Angular, React, Vue), React stood out the most for this application due to the following reasons:

- Easy to learn and use - there is a good supply of tutorials and documentation with clear examples
- Readable code - React makes use of the JSX extension, which lets HTML tag syntax to render particular components. This is much easier to read than plain JavaScript
- Reusable components - reusing components saves time and makes code more accurate

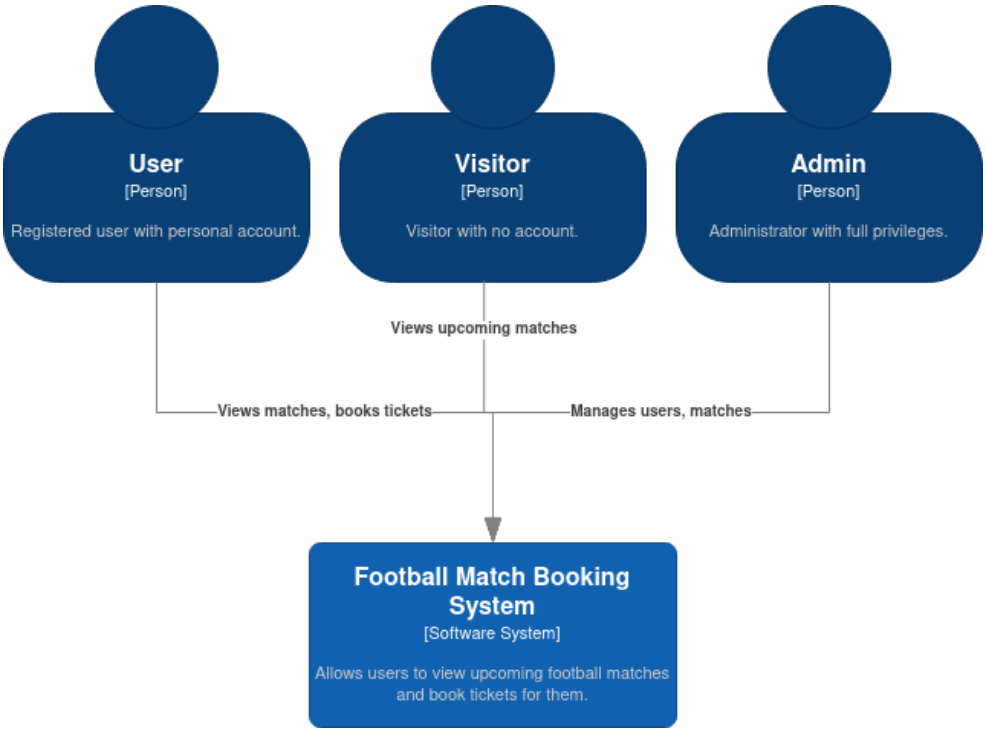
JPA
The Java Persistence Api is a collection of classes and methods for storing, updating and retrieving objects in a database. It does all the heavy work: dividing objects into fields and creating the queries, for example. This will save a considerable amount of time during development. Moreover, it is already tested and works flawlessly.

MySQL
A relational database would be the wise choice for this application since there are lots of relations between users, bookings and tickets. I already have previous experience

with MySQL which will cut development times significantly.

H2 for testing
Since we want tests to be quick, H2 is the perfect fit for this. It is extremely fast, supports inner, outer joins, subqueries and is in-memory, which is suitable for integration tests where we have temporary data.

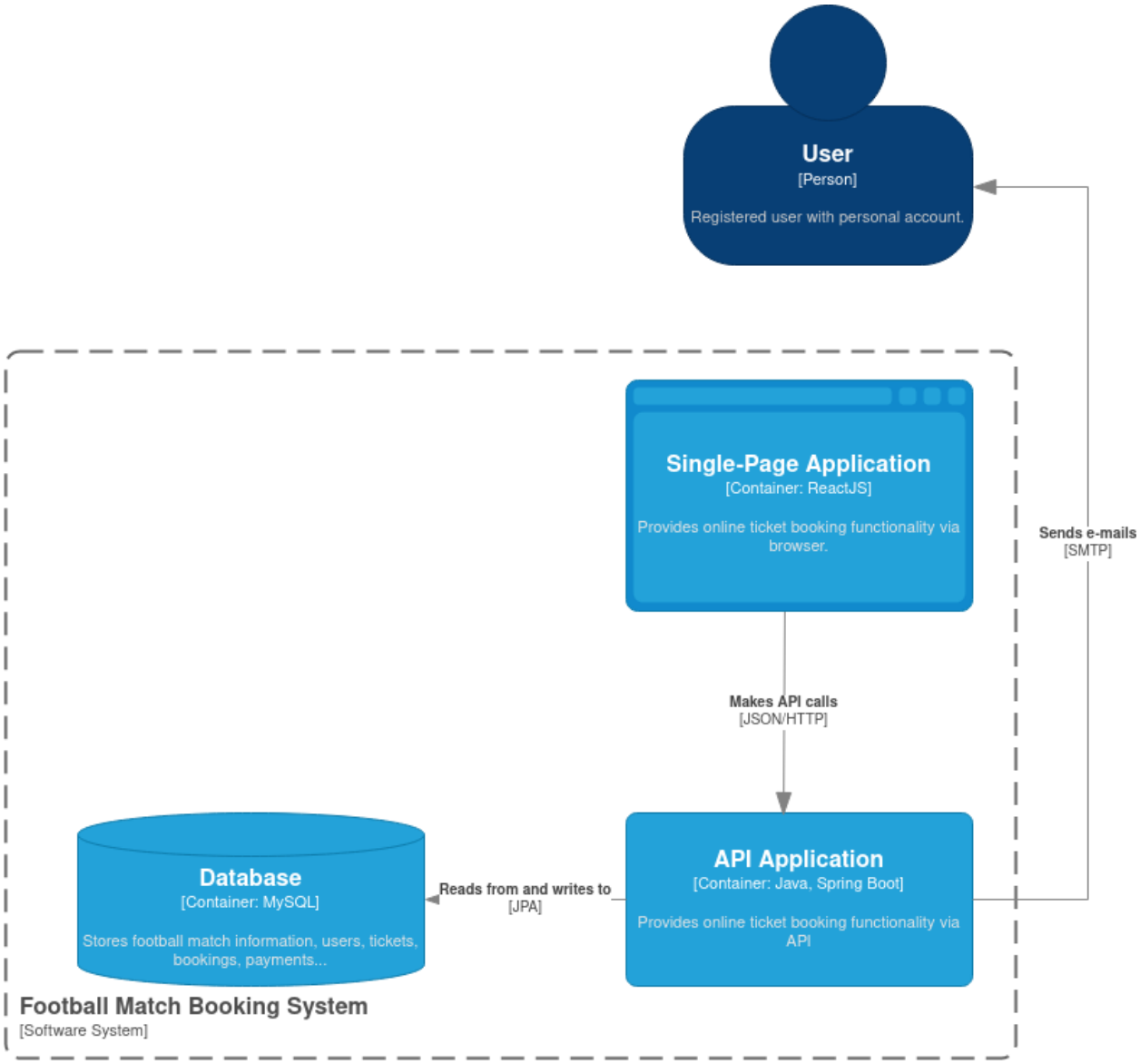
High level architecture
At the *System Context* level, the application has 3 types of users - Registered users, Visitor and Admin. Each user will interact with the software system. The software system will not interact with external systems.



C1 - System Context

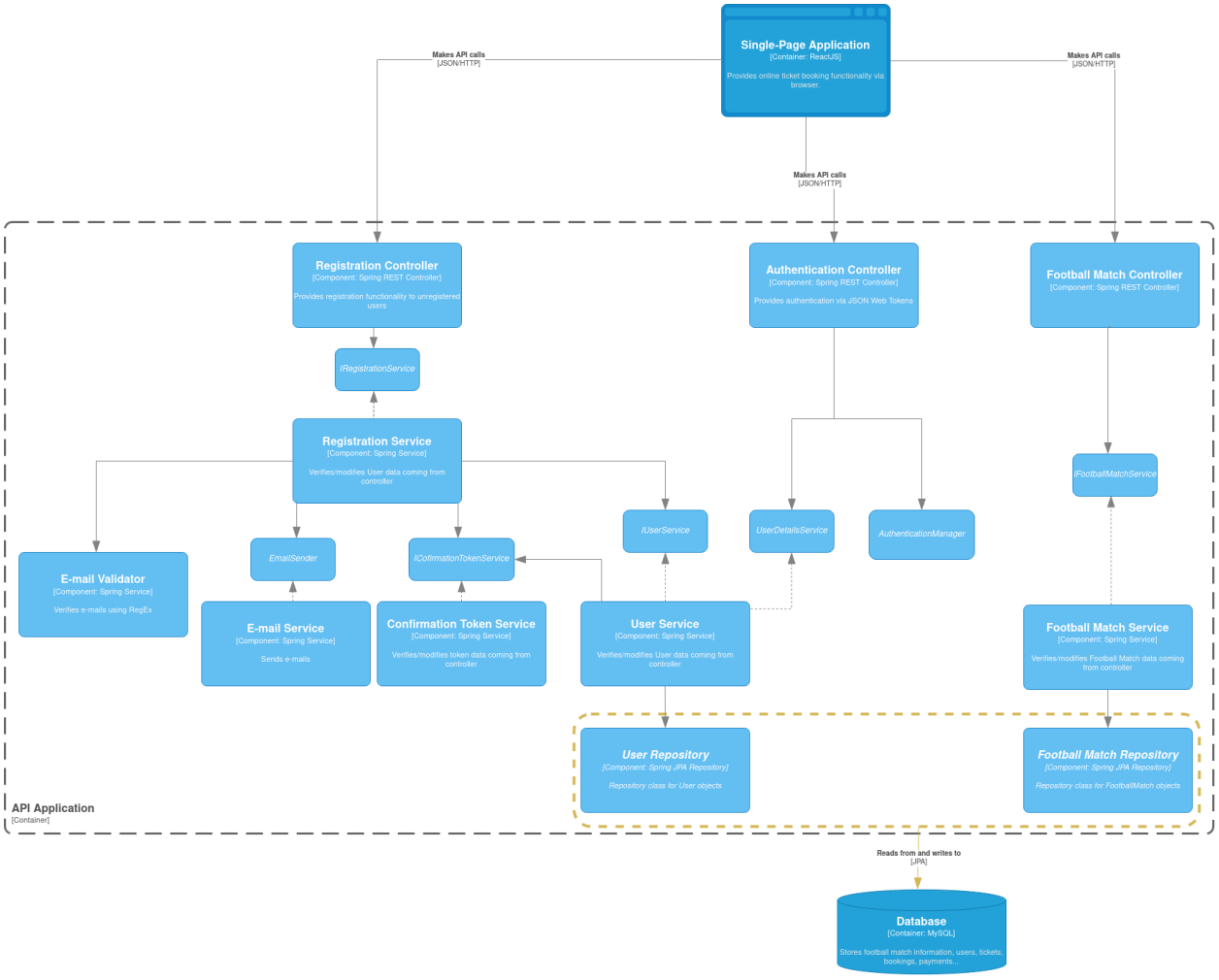
At the *Container* level, the *Software System* is divided into 3 containers:

- Single-Page Application written in ReactJS, which will make API calls to the API Application
- API Application written in Java using Spring Boot, which will read from and write to a database
- MySQL database to store all the data



C2 - Containers

At the *Component* level, the *API Application Container* is divided into Controllers, Services, Repositories and utility classes.



C3 - Components

Single responsibility principle is guaranteed by creating utility classes. For example, the registration service has to ensure the email passed to it is valid, so instead of validating the email into the service, we create a separate class for that task.

Open/Closed principle is not used yet.

Liskov substitution principle is guaranteed by matching the parameter types and return types of the subclass with the parent class. For instance, *WebSecurityConfig* extends *WebSecurityConfigurerAdapter*, but matches the parameter types and return types of *configure()* methods.

Interface segregation principle is not used yet.

Dependency inversion principle is guaranteed by making classes depend on abstraction. In this project, the high-level interface *EmailSender* describes the sending operation and the *UserService* class uses that interface instead of the *EmailService* class directly. This way the low-level *EmailService* class depends on the high-level abstraction.