**DESIGN DOCUMENT**

**Ticketpass**

**Date:** 01/12/2023

**Version:** 4.0

**Made by:** Hristo Ganchev

**Table of Contents**

[1. Introduction 3](#_Toc153829285)

[2. Architecture Constraints and Design Decisions 3](#_Toc153829286)

[3. C4 Model Diagrams 4](#_Toc153829287)

[**Level 1: System Context Diagram** 4](#_Toc153829288)

[**Level 2: Container Diagram** 5](#_Toc153829289)

[**Level 3: Component Diagram (Backend)** 6](#_Toc153829290)

[**Level 3: Component Diagram (Frontend)** 7](#_Toc153829291)

[**Level 3: Component Diagram (Database)** 8](#_Toc153829292)

[**CI Pipeline Diagram** 9](#_Toc153829293)

[**Level 4: Class Diagram** 10](#_Toc153829294)

[4. Explanations and Rationale 12](#_Toc153829295)

[5. Security Report 13](#_Toc153829296)

[**1: Broken access controls** 13](#_Toc153829297)

[**2: Cryptographic failures** 13](#_Toc153829298)

[**3. Injection** 14](#_Toc153829299)

[**4: Insecure design** 14](#_Toc153829300)

[**5: Security Misconfiguration** 14](#_Toc153829301)

[**6: Vulnerable and outdated components** 14](#_Toc153829302)

[**7: Identification and authentication failures** 15](#_Toc153829303)

[**8: Software and data integrity failures** 15](#_Toc153829304)

[**9: Security logging and monitoring failures** 15](#_Toc153829305)

[**10: Server-side request forgery** 15](#_Toc153829306)

[6. Sprints Progress 16](#_Toc153829307)

[**Sprint 1:** 16](#_Toc153829308)

[**Sprint 2:** 16](#_Toc153829309)

[**Sprint 3:** 16](#_Toc153829310)

[**Sprint 4:** 17](#_Toc153829311)

[**Sprint 5:** 17](#_Toc153829312)

[**Sprint 6:** 18](#_Toc153829313)

# **Introduction**

The purpose of this document is to describe the constraints and decisions that the platform is going to be developed under and the flow of the implemented functions by presenting a C4 Model Diagram.

Ticketpass is a platform that provides the user with a wide range of concerts. They can select a concert they would like to attend, purchase tickets for the selected event and gain access to it at the entrance of the venue.

# **Architecture Constraints and Design Decisions**

Based on customer requirements, Spring Boot will be used for the Backend, React for the frontend, and MySQL for Data Management.

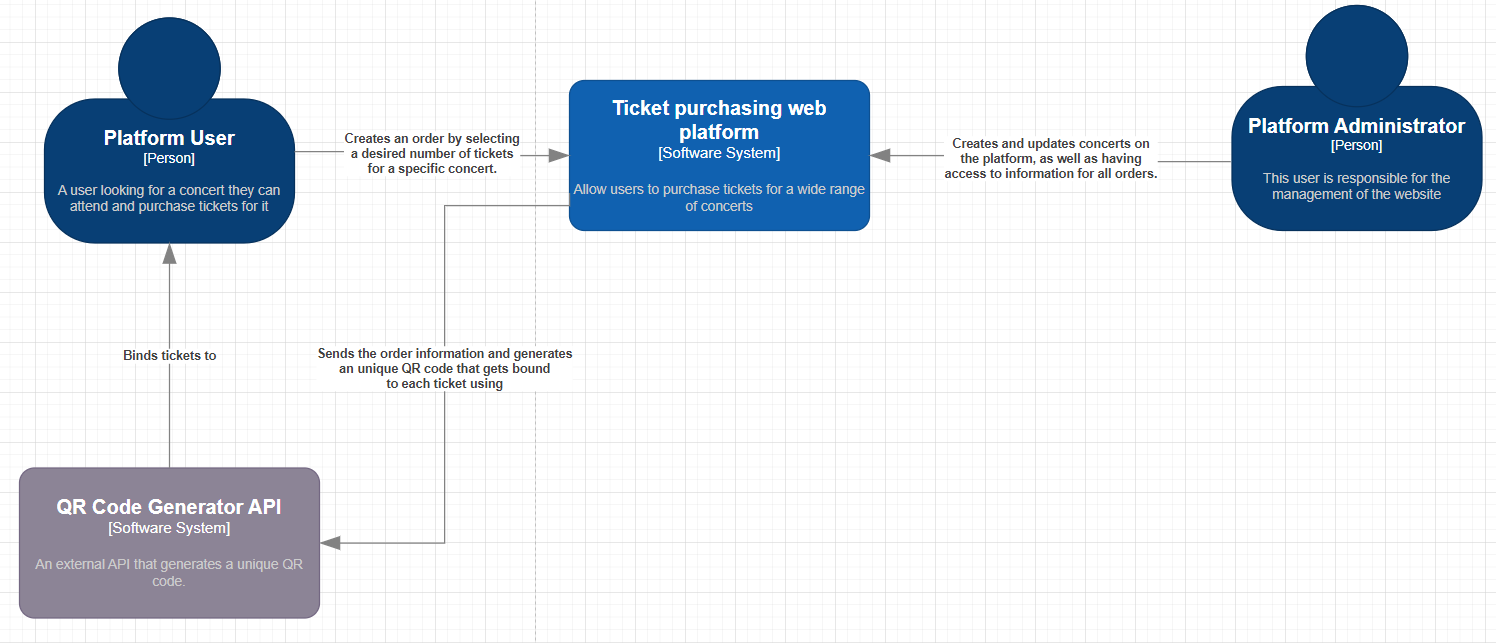
Spring Boot is a framework that provides the developer with a quick compilation of their product and thus the development of the application is conducted more efficiently. What is more, Spring Boot has a large and active community, as well as a wide ecosystem, which will significantly reduce development time. By using this framework, the Dependency Inversion Principle will be followed, promoting loose coupling and avoids the need of continuously and repetitively re-declaring services/repositories throughout the application. The Single-Responsibility Principle and the YAGNI (You Aren’t Gonna Need It) Principle will also be followed.

React uses a virtual DOM and updates only the necessary elements to the UI (User Interface). This clearly follows the KISS (Keep It Simple, Stupid) Principle, making the website application easier to compile and maintain, and promotes a better user experience. Moreover, the framework is perfect for the DRY (Don’t Repeat Yourself) principle, since a component is declared only once and then can be reused in the entire application, without the need of copy-pasting code. The Single-Responsibility Principle will also be followed.

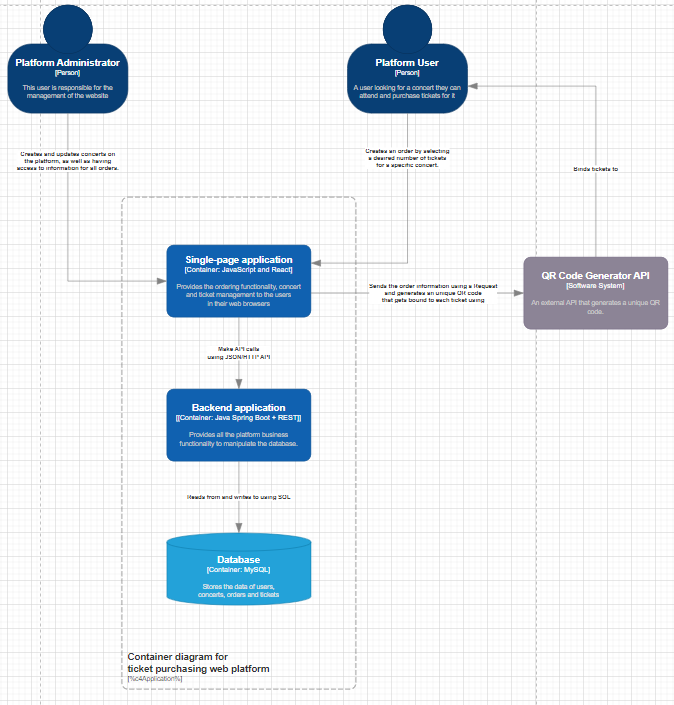
MySQL will be used due to the wide ecosystem of tools and libraries. This will make the database much more adaptable to the development environment. MySQL also has a track record for reliability and performance.

# **C4 Model Diagrams**

## **Level 1: System Context Diagram**

****

## **Level 2: Container Diagram**



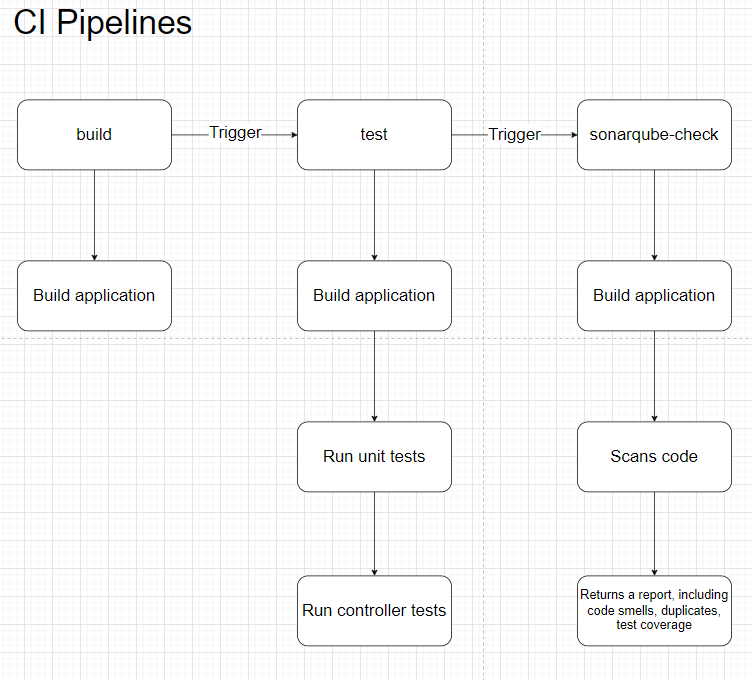
## **Level 3: Component Diagram (Backend)**

## **Картина, която съдържа текст, екранна снимка, диаграма, Паралелен Описанието е генерирано автоматичноLevel 3: Component Diagram (Frontend)**

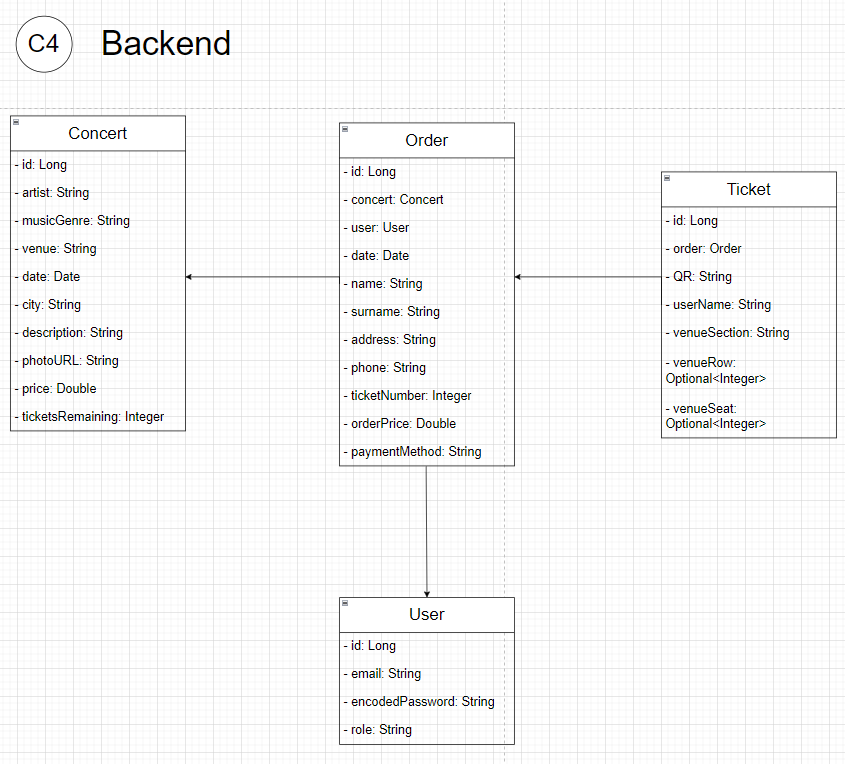
## **Картина, която съдържа текст, екранна снимка, диаграма, линия Описанието е генерирано автоматичноLevel 3: Component Diagram (Database)**

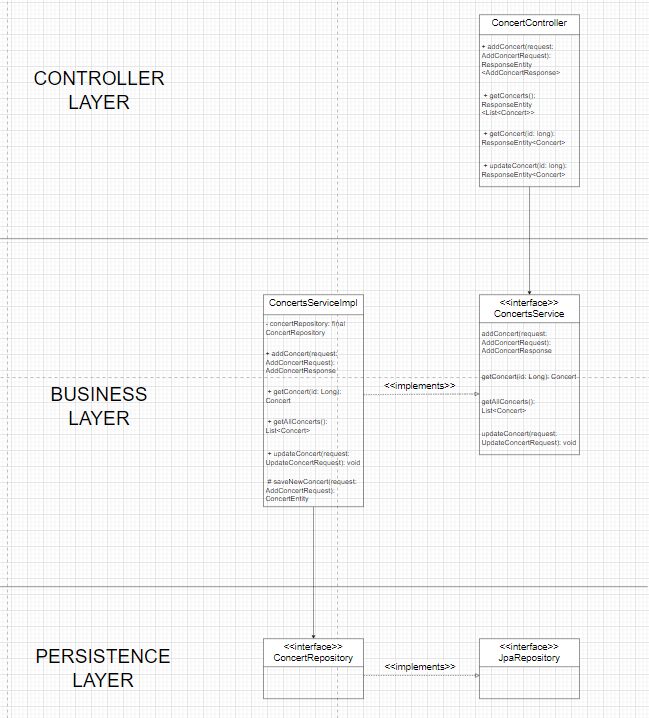


## **CI Pipeline Diagram**

****

## **Level 4: Class Diagram**

****

****

# **Explanations and Rationale**

The administrator is responsible for managing the platform. They can add and update concerts from the MySQL Database, can review orders, and overall manage the flow of the platform.

The user is a ticketholder. They select a concert, purchase the desired number of tickets, which get bound to their account.

The project is divided into three parts: a single-page application, a backend application, and a database.

The single-page application is updated accordingly, and only necessary UI elements are displayed. This makes the platform much easier to compile and provides the user with a fast and easy experience.

The backend application is implemented using Java Spring Boot. It is responsible for all the logic behind the application. It is separated into 4 different categories (User, Ticket, Order, Concert). Each category consists of a REST Controller that is responsible for managing the HTTP requests, a Service layer where all of the logic is managed, and a Repository that assists with the retrieval and updating of information to/from the database.

The database is where all of the information is stored. Measures will be taken in order to keep sensitive information (e.g. passwords) hidden in case of data leaks.

The repository where the backend application is stored is connected to three CI Pipelines: build, test, and sonarqube-check. If the pipeline prior to the current one fails, all of the other pipelines are skipped in order to not run faulty code.

The first pipeline is responsible for the building of the application. If this pipeline fails, all the others will not be executed.

The second pipeline is responsible for running the tests in the application. First, it compiles the application, and then runs all of the tests in the program. If at least one tests fails, the pipeline fails, too.

The last pipeline in the flow is sonarqube-check. It will not run unless the application successfully compiles. It is responsible for giving more insight into the application. By “scanning” through the code, it recommends pieces of code being changed, which code blocks are duplicate, and what percentage of the application do the tests cover. It is only applied in branches, such as main and master.

# **Security Report**

The OWASP Top 10 Security Risks represent the most critical risks to modern web applications. In this security report, it is described how the application deals (or does not deal) with them.

## **1: Broken access controls**

Broken Access Controls involves bypassing access control checks (e.g. executing admin operations while logged in as a user). For this exact reason, the application has two “shields”. One is the @RolesAllowed annotation, which prevents any unauthorized user to run methods eligible for admins only, and another one that checks for the ID of the logged in user, and whether the ID is equal to the one provided in the access token. If it is not, the method is unable to execute.

## **2: Cryptographic failures**

Cryptographic failures may lead to exposing data, including passwords and other personal information. For this purpose, all of the passwords in the database are hashed using a B Crypt Password Encoder, and then decoded when logging into the application.

## **3. Injection**

Injection involves using a form or other text fields for submitting data for executing malicious SQL statements or OS commands. To protect the application, instead of using default SQL statements, the database is working only with parameterized queries, meaning that each field that should be filled from the user is read as a parameter. If a user tries to, for example, fill in an SQL query in the filter for the application, the statement will be recognized as a parameter, and the query will not be executed.

## **4: Insecure design**

Insecure design is a risk more focused on flaws in design. If proper security controls are not created in the application, it is vulnerable to attacks. However, the security of the product is taken into consideration, and measures are taken to protect the website. One example is the generation of access tokens. In order to encode a token, a signature key is used. The key is stored in an .env file outside of the repository, meaning that only the website owner has access to it, and no one else. Other measures are the hashing of passwords, parameterized queries, methods accessible only to authorized users, and ID checks.

## **5: Security Misconfiguration**

This risk can occur if unnecessary frameworks and features are enabled, or latest security features are not configured correctly. This is why imports from unknown origin and/or missing knowledge are avoided and not present.

## **6: Vulnerable and outdated components**

Vulnerable and outdated components might mean that some of the components used in the application are outdated, making them much more vulnerable and easier to exploit. This is why the components used are examined whether they are still supported or not. Moreover, both Spring and React have mechanisms for handling unsupported components.

## **7: Identification and authentication failures**

Identification and authentication failures involve the attacker bypassing security measures, gaining access and assuming the identity of a user. This can involve obtaining a user’s orders without being authenticated. As mentioned above, there are already measures that deal with this risk. Both the @RolesAllowed annotations and the ID checks present in the Service Implementations in the Backend prevent unauthorized users from executing endpoints unless they are logged in as the said user.

## **8: Software and data integrity failures**

These failures can be caused by plugins, libraries or modules from untrusted sources, which use insecure update mechanisms, or insecure deserialization. This is why only popular libraries are used, since they are considered to be more secure against such circumstances.

## **9: Security logging and monitoring failures**

This failure involves not monitoring suspicious behavior (e.g. failed logins or high-value transactions). For now, logs are only stored locally, however, if any suspicious activity is noticed, there is the possibility to ban the user involved. In case of a false ban, the user ban can also be reverted.

## **10: Server-side request forgery**

SSRF allows attackers to make requests to separate resources. They can send a request to a destination that is not expected. At the moment, no firewall preventing such attacks is present in the application.

(Source: <https://www.zscaler.com/blogs/product-insights/what-owasp-top-10?psafe_param=1&_bt=677493654214&_bk=&_bm=&_bn=g&_bg=154576188893&utm_source=google&utm_medium=cpc&utm_campaign=google-ads-na&gad_source=1&gclid=EAIaIQobChMI-9XkhMiZgwMVjtV3Ch1DkAmoEAAYASAAEgKvD_D_BwE>)

(Source: https://owasp.org/Top10/)

# **Sprints Progress**

## **Sprint 1:**

1. As an administrator, I want to add concerts to the database with the necessary information so that they can be displayed to the users of the platform.

* A database that stores concerts should be created.
* Each concert should have a unique ID, Artist Name, Music Genre, Venue, City, Date, Description, Photo URL, Ticket Price and Tickets Remaining.
* A method that adds concerts to the database should be implemented

## **Sprint 2:**

1. As a user, I want to see all concerts in one place, so that I have a wide selection of different concerts of different genres available in a glance.

* A page where concerts are retrieved from a database should be created
* Each concert should be displayed with their name, photo, and description in the “Concerts” page

## **Sprint 3:**

1. As a user, I want to be able to purchase tickets online so that I can reserve my place in the event I would like to attend fast and easy.

* A “Checkout” page should be implemented.
* A method for making an order should be created.
* All orders made should be stored in the database.

## **Sprint 4:**

1. As an administrator, I want to have all of my users passwords hashed, so that in case of a data leak their personal information will not be available to the public.

* Create a method for hashing a password.
* Create a method that compares the inputted value for password (when logging in) and the hashed password stored in the database.
* Passwords must NOT be stored explicitly in the database.
* Create a method for generating an access token
* Administrators should have authorization for specific methods that regular users cannot access.

1. As a user, I want to be able to see all of my tickets for a specific concert in one place, so that it is easier for me and my friends to show them at the entrance of the venue.

* A “Tickets” page should be created
* All tickets should be stored in the database
* Each ticket that is equal to a specific order ID should be displayed on the page with an unique QR code

1. As an administrator, I want to have access to all orders made, so that I can confirm whether a user that has contacted me has made a valid purchase or not.

* An “All orders” page should be created.
* All orders made by all users should be displayed in the “All orders” page

## **Sprint 5:**

-

## **Sprint 6:**

-