“Ari on tour” concert ticketing website

– Design document –

Irina-Maria David – 3371581

Student at Fontys University of Applied Sciences – S3CB04

Document version 1.0

Table of Contents

Description of the solution

Below I will explain about the solution that I thought of for this project.

I want to create a website through which fans can keep up with the latest concerts of Ariana and find more information about the artist and her performances.

The **visitor** will be able to create an account, check out the main page, the page about Ariana, contact page and detailed information about the events such as tickets availability and types, date, time, location, pricing. They will not have the ability to buy tickets if they do not have an account on the website. The visitors will also have the ability to see the bought tickets count to all the events.

After registering an account, a **user** will be able to successfully log in, then view the main page, the page about Ariana, contact page, and tickets and event information. A user will be able to purchase tickets for the events that they prefer and pay for them. The users will see the bought tickets count as well. A dashboard will be provided showing information about the user and the bought tickets. A user will have the ability to "Change the password" from the dashboard and will also feature of the" Forgot my password option".

An **administrator** account will be able to successfully log in, view statistics of the sales – tickets, earnings, see the total created account and detailed information about the customers of the website. The administrator account features actions such as adding, editing, deleting, marking events as "to be announced" or "cancelled". When adding an event, the following information should be provided by the administrator: the date, time, location, pricing, ticket types, stocks for tickets, information about opening acts. The administrator will also be able to see the count of all bought tickets and more detailed information such as how many tickets were sold for a certain event, most bought ticket type for a certain event, event with most tickets bought.

How the solution will be implemented

In order to complete this project of building a full stack web application, I will be using ReactJS JavaScript library for the frontend part and for the backend Spring Boot, Java, Gradle, MySQL and Axios. In this case, these choices are influenced by the requirements of the project themselves. But all of the above-mentioned aim to guide me to developing a website and code in a structured way by applying the best practices of the field.

React

React is considered easier to learn compared to other frameworks and requires knowledge of basic JavaScript. React code is considered easy to maintain and flexible due to the modular structure that it has by using components.

React provides the ability of reusing components. For example, in my current version of the project the component which has the Axios get call (gets all the customers of the website) will later be reused when I will implement the refined search by last name for the customers by importing it in the component handling the search through all the registered customers. In React I will use JSX which basically combines React and JavaScript with HTML, by converting the tags into React elements. Also, in React I made use of Axios to make HTTP requests to my backend fake date repository and receive for example all the customer information.

```
function CustomerShowAllPage(){

    const[customersList, setCustomersList] = useState([]);

    useEffect(() =>{
        axios.get("http://localhost:8080/customers")
        .then(response => {
            setCustomersList(response.data.customers);
        })
        .catch(err => console.error(err));
    });

    return (
        <CustomerList customersList={customersList}/>
    )
}
```

Spring Boot

Spring Boot provides a flexible way of configuring database transactions. For example, for when my Spring application will be connected to a MySQL database. Spring Boot can manage REST endpoints, allowing me to make API requests and receive responses for example the GET request to /customers returning a list of all the registered customers of the website.

```
@RequestMapping("/customers")
//@CrossOrigin(origins = {"http://localhost/3000"})
@CrossOrigin(origins= {"*"}, maxAge = 4800, allowCredentials = "false" )
@AllArgsConstructor
public class CustomersController {
    1 usage
    private final GetCustomersUseCase getCustomersUseCase;
    1 usage
    private final GetCustomerUseCase getCustomerUseCase;
    1 usage
    private final CreateCustomerUseCase createCustomerUseCase;
    1 usage
    private final DeleteCustomerUseCase deleteCustomerUseCase;

    Irina
    @GetMapping
    public ResponseEntity<GetCustomersResponse> getCustomers(){
        return ResponseEntity.ok(getCustomersUseCase.getCustomers());
    }
```

Layering of the application

The backend application has a 4 layers Spring Boot Architecture. The Controller layer is the top layer, and it takes care of handling the HTTP requests, while within it the URL mapping of the website is done as well. The Business layer holds the business logic of the application and has the service classes. The logic in the Business layer is used in the Controller layer. The Persistence layer contains the logic of the database storage and is responsible for converting business objects to the database row and the other way around. The database layer contains the database, in our case the MySQL database and performs the CRUD operations.

SOLID

SOLID refers to five class-design principles. It is a structured design approach which will be used in this project as well and it ensures that the software is modular and easy to maintain, understand, debug, and refactor – improving the design of the existing code (restructuring) without changing its original functionality.

Each of the principles focuses on a different concern and below I will explain each of them and add an example of how they are currently applied or will be applied while developing my website.

Single Responsibility Principle (SRP)

The idea behind this design pattern is to have classes, modules or functions that should serve only one purpose, meaning that a class will do one job and it should have only one reason to change.

For example, in the backend we have separated layers and each one of the components of the layers has only one job. The same applies for the frontend where we have separated pages and also each of the components have their own purpose.

Open-closed principle (OCP)

It says that objects or entities should be open for extension but closed for modification, basically meaning that a class should be extendable without modifying the class itself.

When I wanted to find the user by his username, I did not modify the method existsById and instead I created a new method existsByUsername.

```
public interface CustomerRepository {
    1 usage  1 implementation  Irina
    List<CustomerEntity> findAll();
    6 usages  1 implementation  Irina
    CustomerEntity save(CustomerEntity customer);
    1 usage  1 implementation  Irina
    void deleteById(long customerId);
    1 usage  1 implementation  Irina
    Optional<CustomerEntity> findById(long customerId);
    1 usage  1 implementation  new *
    boolean existsById(long customerId);

    1 usage  1 implementation  Irina
    int count();

    1 usage  1 implementation  Irina
    boolean existsByUsername(String username);
}
```

Liskov Substitution Principle (LSP)

It states that objects of a superclass should be replaceable with objects of its subclass without breaking the application. This is applied in the projects' backend when creating interfaces and the classes implement those interfaces.

For example, in our Business layer, there an interface is created for each of the use cases, having classes which provide a certain use case implementation.

Interface Segregation Principle (ISP)

It states that clients should not be compelled to implement interfaces they don't use. ISP splits very large interfaces into smaller and specific ones, so that the client will only have to know about the methods that represent interest to them.

For example, in my backend project instead of having a big interface containing all the use cases, I split it up into smaller interfaces, where each one focuses on each use case.

Dependency inversion (DIP)

Dependency inversion principle helps in connecting classes together while keeping them independent at the same time. Dependency inversion says that high modules should not depend on low level modules, their communication is established through abstractions. In our case an example is the constructor of the CustomerController which communicates with the CreateUseCaseImpl through the interface CreateCustomerUseCase.

```
public interface CreateCustomerUseCase {
    1 usage  1 implementation  Irina
    CreateCustomerResponse createCustomer(CreateCustomerRequest request);
}
```

```
@Service
@AllArgsConstructor
public class CreateCustomerUseCaseImpl implements CreateCustomerUseCase {
    2 usages
    private final CustomerRepository customerRepository;

    1 usage  Irina
    @Override
    public CreateCustomerResponse createCustomer(CreateCustomerRequest request){
        if(customerRepository.existsByUsername(request.getUsername())){
            throw new UsernameAlreadyExistsException();
        }
        CustomerEntity savedCustomer = saveNewCustomer(request);
        return CreateCustomerResponse.builder()
                .customerId(savedCustomer.getId())
                .build();
    }
```

```
public class CustomersController {
    1 usage
    private final GetCustomersUseCase getCustomersUseCase;
    1 usage
    private final GetCustomerUseCase getCustomerUseCase;
    1 usage
    private final CreateCustomerUseCase createCustomerUseCase;
    1 usage
    private final DeleteCustomerUseCase deleteCustomerUseCase;
```

Dependency injection

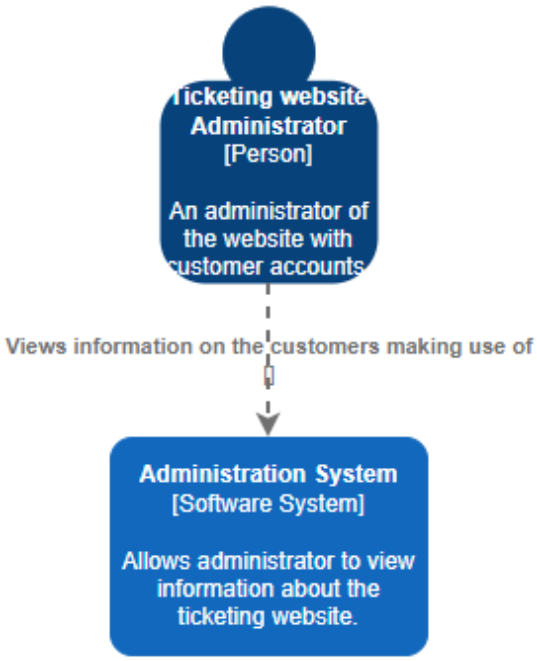
The dependency injection is the concept in which objects get other required objects from outside. The dependencies of a class can be passed through a constructor, a property, or a method. It is not yet implemented in my project, but it will be when for example I will use an outside object in a class. The benefit of dependency injection is decreased coupling between the classes and their dependencies. In my application I will be using dependency injection to increase the possibility of reusing classes and testing them independently while unit testing.

Separation of concerns

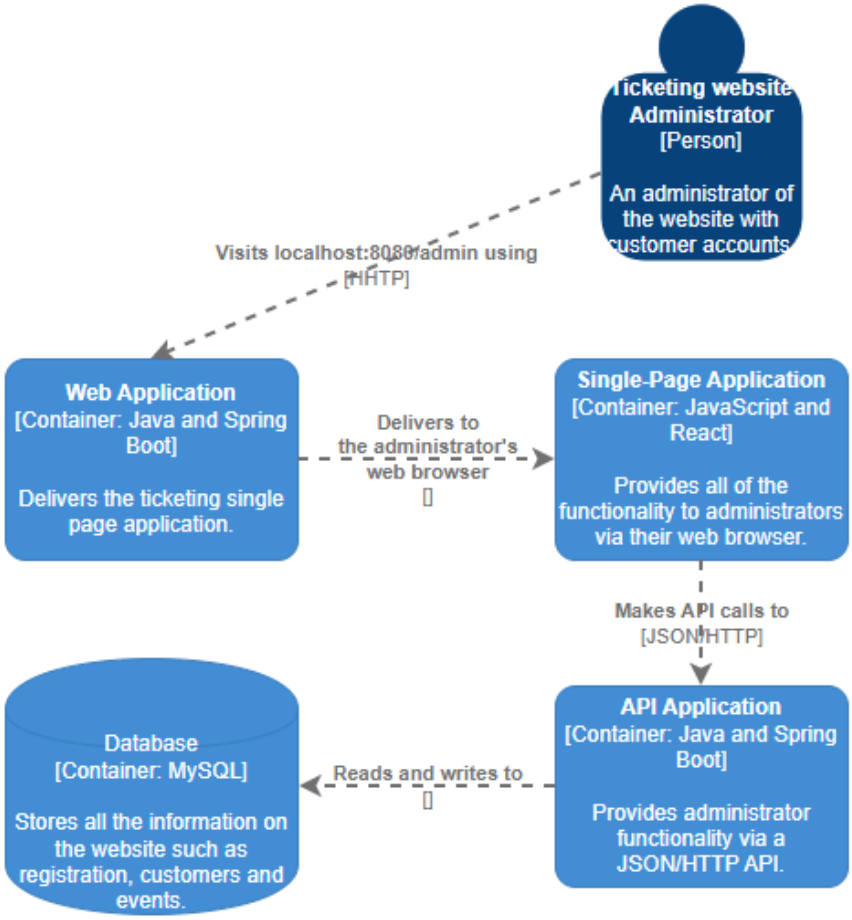
Separation of concerns is a design principle which refers to separating an application like mine for example into distinct sections, in such way that each section addresses a separate concern. For example, the user interface is a concern of an application, and the business logic is another concern. Adding changes to the user interface should not require changes to the business logic and the other way around applies. In our case, the way the backend application is layered, as explained in the chapter above, is a great example of separation of concerns since each layer handles a different concern.

Diagrams

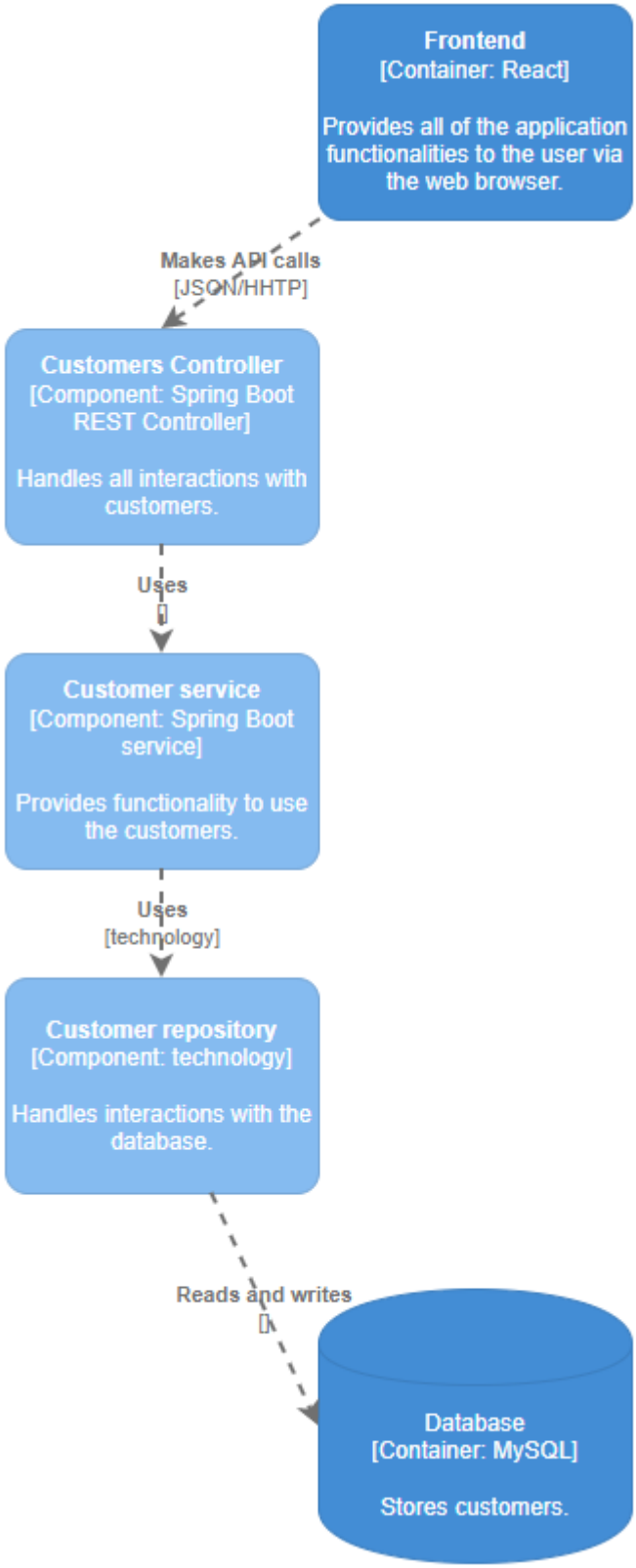
C1 Diagram – System context



C2 Diagram – Container



C3 Diagram – Component



References

Oloruntoba, S. (2021b, November 30). *SOLID: The First 5 Principles of Object Oriented Design*. DigitalOcean Community. https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

Millington, S. (2022, September 19). *A Solid Guide to SOLID Principles*. https://www.baeldung.com/. https://www.baeldung.com/solid-principles

Hutsulyak, O. (2022b, August 10). *10 Key Reasons Why You Should Use React for Web Development*. Blog | TechMagic. https://www.techmagic.co/blog/why-we-use-react-js-in-the-development/

Peerbits. (2022, February 7). *The benefits of ReactJS and reasons to choose it for your project*. https://www.peerbits.com/blog/reasons-to-choose-reactjs-for-your-web-development-project.html

Olawanle, J. (2022, May 17). *Axios React – How to Make Get, Post, and Delete API Requests*. freeCodeCamp.org. https://www.freecodecamp.org/news/axios-react-how-to-make-get-post-and-delete-api-requests/

GeeksforGeeks. (2022, April 29). *Axios in React: A Guide for Beginners*. https://www.geeksforgeeks.org/axios-in-react-a-guide-for-beginners/

Introducing JSX –. (n.d.). React. https://reactjs.org/docs/introducing-jsx.html

Arancio, S. (2022, January 5). *What is JSX? - Stephen Arancio*. Medium. https://medium.com/@sjarancio/what-is-jsx-e3dda0af3490

[Tom Gregory]. (2019, August 23). *Gradle Tutorial - why you should use it and how to get started* [Video]. YouTube. https://www.youtube.com/watch?v=ojx49J1JCdQ

Spring Boot - Introduction. (n.d.). https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm

Kushnir, A. (2022, June 8). *Pros and Cons of Using Spring Boot*. Insights. https://bambooagile.eu/insights/pros-and-cons-of-using-spring-boot/

GeeksforGeeks. (2022a, March 11). *Spring Boot - Architecture*. https://www.geeksforgeeks.org/spring-boot-architecture/

Java Dependency Injection - DI Design Pattern Example Tutorial. (2022, August 3). DigitalOcean Community. https://www.digitalocean.com/community/tutorials/java-dependency-injection-design-pattern-example-tutorial

SOLID Principles : The Dependency Inversion Principle | Making Java Easy To Learn. (2022, September 21). Making Java Easy to Learn. https://javatechonline.com/solid-principles-the-dependency-inversion-principle/