Design Document

Vibe Check

Date	**:**	**29/11/2024**
Version	**:**	**1.0**
State	**:**	**Finished**
Author	**:**	**Nuno Dias**

Version history

Version	Date	Author(s)	Changes	State
0.1	11/10/2024	Nuno Dias	First Draft	Finished
0.2	07/11/2024	Nuno Dias	Slight adjustments based on Feedback Expanded based on progress, new choices (MySQL) and changing structure	Finished
1.0	29/11/2024	Nuno Dias	Added CI pipeline Diagram & explanations; Added introductions to each chapters; Updated Backend Diagrams;	Finished

Contents

1. Architecture Constraints

Constraints imposed on the project technology, what choices were made and why? As well as an overview of the benefits, constraints and alternatives. For future reference.

1.1. Spring Boot

Spring Boot is a Java framework, an object oriented language and Java is still used in countless servers (2024) around the world due to its popularity, past and present. This means that Spring Boot, a very popular Java backend framework also has a lot of users and therefore support, the resources are countless. It also has built-in Dependency Inversion and Injection. Enforcing the SOLID principles in Spring is easily done as the framework provides tools that do more than half the work, keeping the codebase clean and maintainable.

From a consumer stand point it also has many advantages. It is a tried and true framework so there is little risk that your application will crash due to Spring. Being the most popular JVM (Java Virtual Machine) framework, it is also extremely unlikely to be abandoned, meaning the Application would need to be remade almost from scratch. It is also very fast, secure and easily scalable, serverless or otherwise.

All alternatives, even the closest to Spring Boot in popularity, fall behind by a large amount and their community/support will always be lacking compared to this behemoth. As the saying goes, do not fix what isn't broken. Spring Boot has withstood the test of time and people seem happy with it.

1.2. Vite + JavaScript + React

Vite was chosen because of its lightning-fast development experience. When running a development environment Vite doesn't bundle the whole code when changes are made but only updates the necessary parts of the code, meaning changes to code result in near instant updates. This is essential for the development of a single-page application (SPA) since all the code is tied together. Besides reducing wait time for developers, it also offers a lot of customization from the script type to popular libraries (React, Vue) to deployment options.

JavaScript was chosen due to better support for packages. The React library was chosen for its performance and modularity. React components are great for organizing code in SPAs, they provide the opportunity to apply some SOLID principles (e.g. Single Responsibility), as well as DRY (Don't Repeat Yourself). React Components allow code to be easily reused elsewhere. It also provides hooks which provide some template functions for certain uses and has a strong community & support due to its popularity.

While other options such as NextJS exist for the deployment, our client insisted on Vite. Besides while NextJS, as a new shiny framework, solves some problems, Vite has been around for longer and has better documented issues/answers.

1.3. Tailwind

Similar to other decisions, Tailwind was chosen due to its popularity, and therefore community, that developed easy ways to integrate it into any framework and many, many free and stylish components available online.

More generally, a CSS Framework was necessary since as a small team we could not afford to have someone spent all their time on the frontend design, and none of us are very skilled at CSS. Tailwind also integrates well with popular React component libraries like Shadcn and MagicUI.

While there are other CSS Frameworks, some like Bootstrap are more restraining and don't allow you to do everything you can with normal CSS without jumping through some hoops. As for the more simple and less restraining ones, Tailwind is simply the most popular and well documented/supported.

1.4. MySQL

MySQL was chosen as Database due to the demand of the client. However it does have a lot of advantages. As proven and popular choice it has a robust community with integrations for other tools like Flyway and most Database viewing tools (Azure, VS Code extensions, etc.). It also has a stable docker image that runs seamlessly with a simple docker run command that pulls from docker hub.

Compared to MS SQL witch runs best with proprietary Microsoft MS SQL Server and other tools, this option affords more liberty with supporting software.

1.5. Docker

Deployment to a Docker container was a no brainer. The App, both frontend and backend, can benefit immensely from being deployable anywhere. Each part of the App has a Dockerfile to containerize it and Docker itself is used to run CI/CD pipeline tests.

Since this project will has a CI/CD pipeline, for which docker is perfectly suited, by having the docker executor setup in our GitLab-runners the Application can be tested on multiple machines, always in a clean & neutral environment, then build as a docker image and finally deployed.

As for competitors Docker is basically unrivalled.

1.6. GitLab

As briefly mentioned before, GitLab was chosen as the git server. This was mainly due to a GitLab instance being hosted by our company. This means the project stays within company servers and costs are server costs are covered.

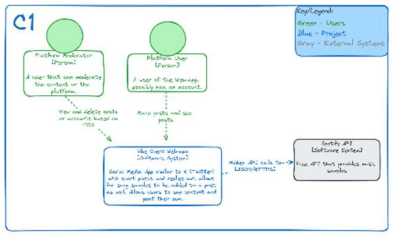
That being the main reason, GitLab does provide a lot of tools & integrations to maintain and test code bases. Although not as many as GitHub.

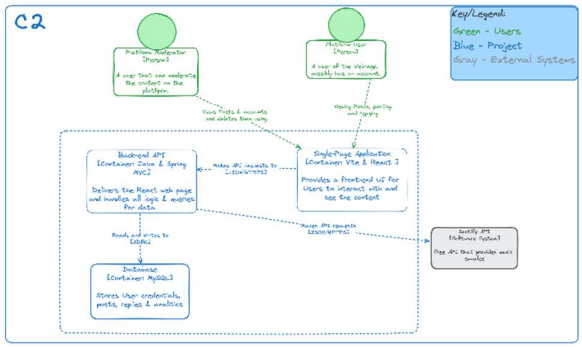
2. Diagrams

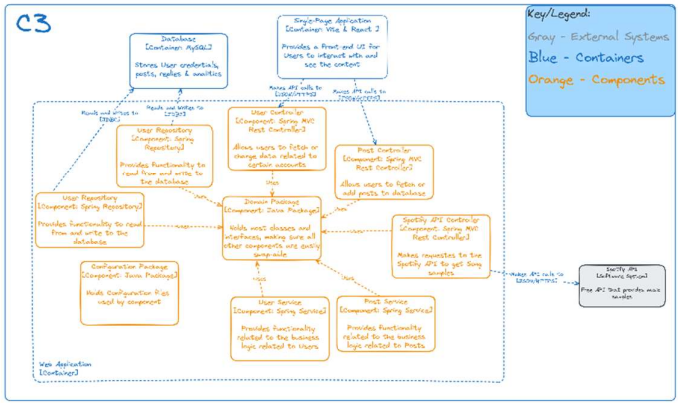
A series of diagrams that explain the various parts of the application visually. From overall design, to detailed structure to testing.

2.1. C4 Diagram

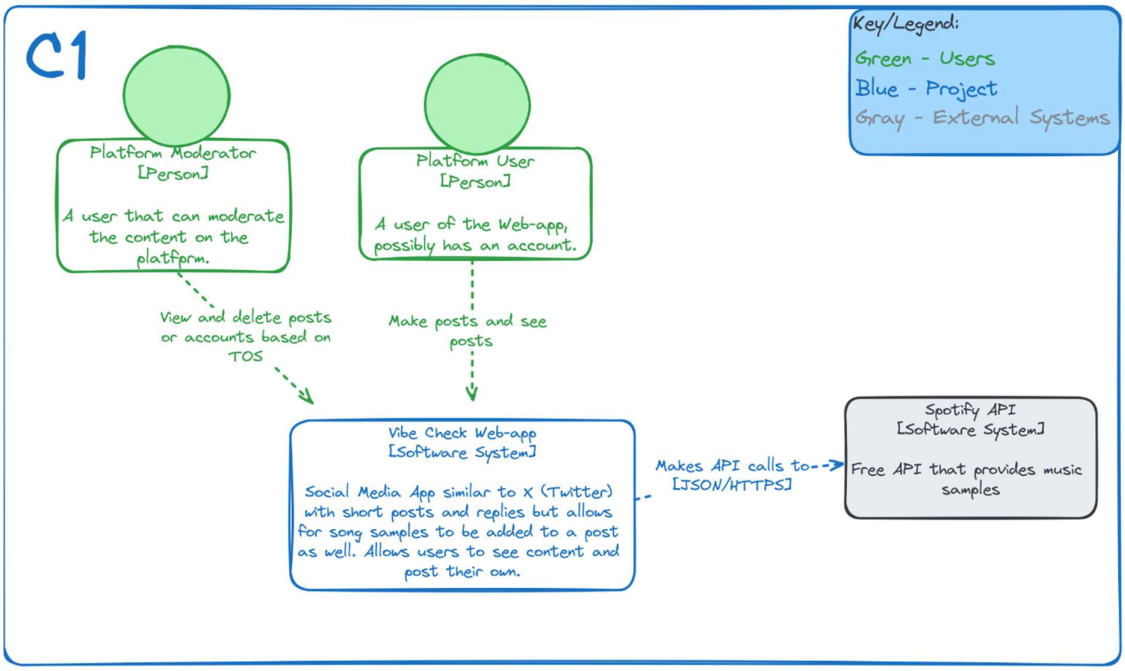
Vibe Check C4 Plan







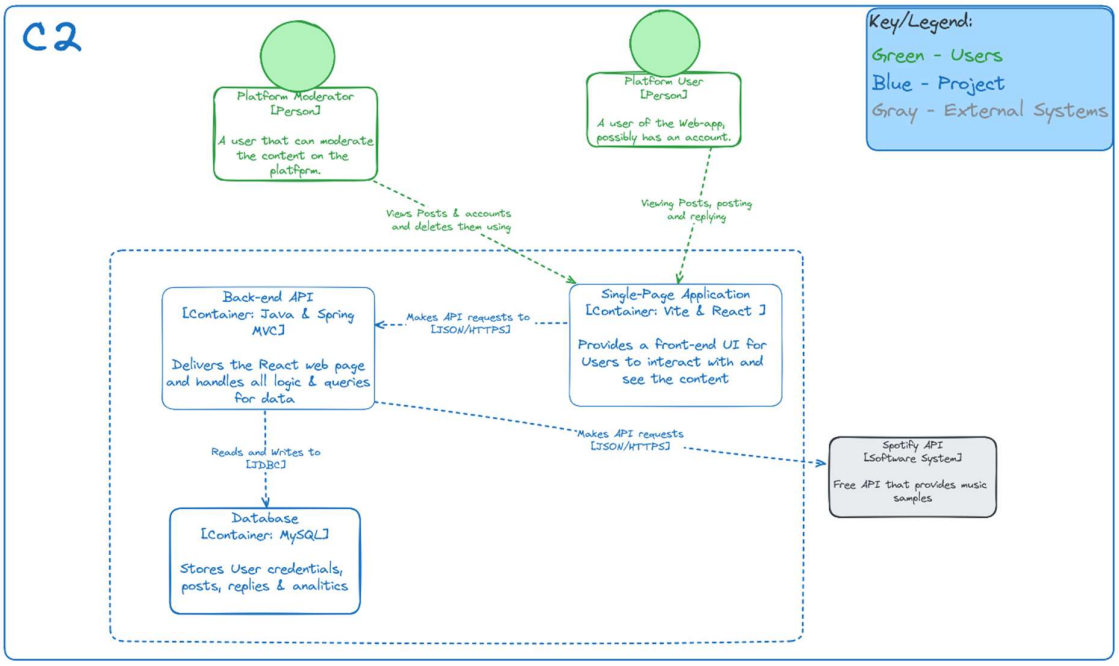
2.1.1. Level-1



This diagram shows the high-level context of the whole System.

The VibeCheck Web-App will have two types of Users, customers who use the app for recreational purposes and maintainers who can moderate the content and other Users. It will also interface with one External system, by default the Spotify API, through API calls to get song samples to show in combination with posts.

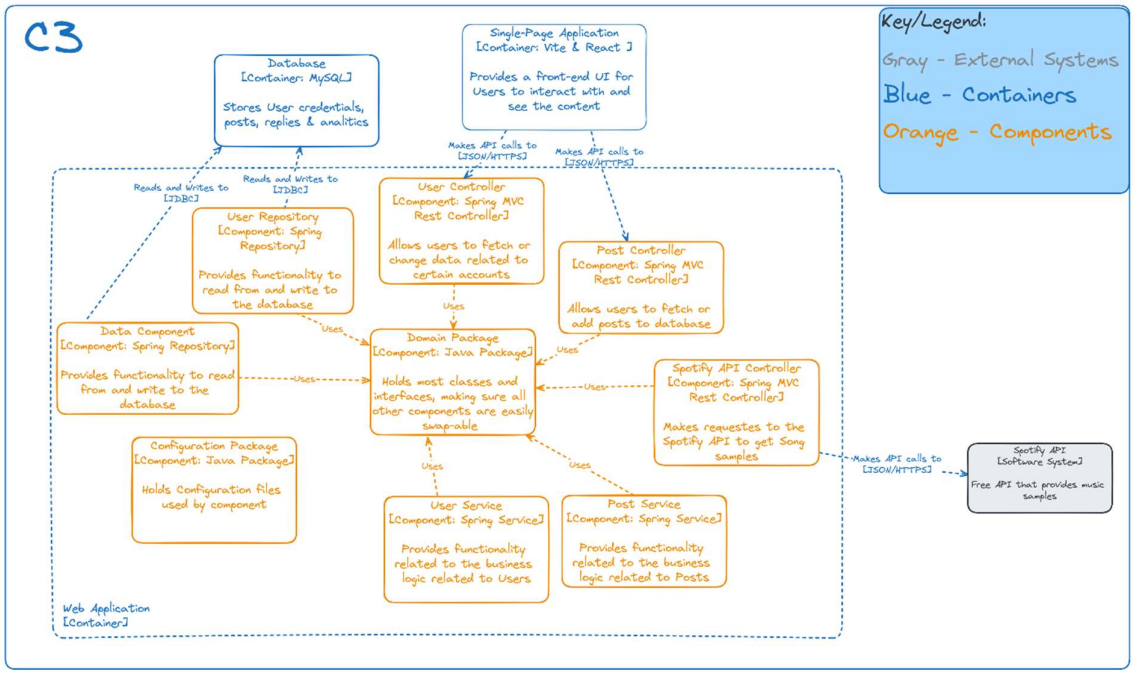
2.1.2. Level-2



This diagram shows a slightly more detailed version of the whole System, broken down into containers that represent parts of the Web-App.

The VibeCheck Web-App will further be broken down into 3 containers: Backend, Frontend and Database. They are separated by concerns, the frontend Single-Page Application (SPA) is in charge of the UI and does not need to know how to process data, by following the SOLID principles here we get a modular, maintainable and scalable application. In the future, if desired, we can add a second frontend in the form of a desktop app or mobile app, these can be integrated seamlessly without touching the rest of the containers.

2.1.3. Level-3 Backend



This diagram shows a detailed view of a container from the system, in this case the backend, broken down into components that represent important parts of the container. These components are semi-arbitrary.

The Spring MVC app will have an onion architecture, meaning most java packages point to the Domain. This architecture is focused on separating core business logic from external dependencies, such as databases, APIs, or user interfaces by organizing the system into layers that form an onion-like structure. This is great for maintainable, scalable and loosely-coupled applications such as this one. It means that the outer layers can be easily swapped or modified without modifying the other layers.

This architecture also encourages dependency inversion and other SOLID principles as seen by the fact all components are using (dependent on) Domain instead of other components being dependent on them, meaning changes to this component will affect the others. It also follows the DRY and KISS principle since classes and interfaces are not spread everywhere, they are simply kept in one place.

As seen in the diagram the Backend will have three Controllers, two of them responsible for the main types of content of the application, Posts and Users. Which will communicate with their respective Service components to perform any logic needed by the API request, which in turn will communicate with their respective Repository components to read from and write to the database. The third controller and service are essentially a wrapper around the Spotify API which will perform any processing required to the data from Spotify and return it to then display it to the User.

2.1.4. Level-3 Frontend

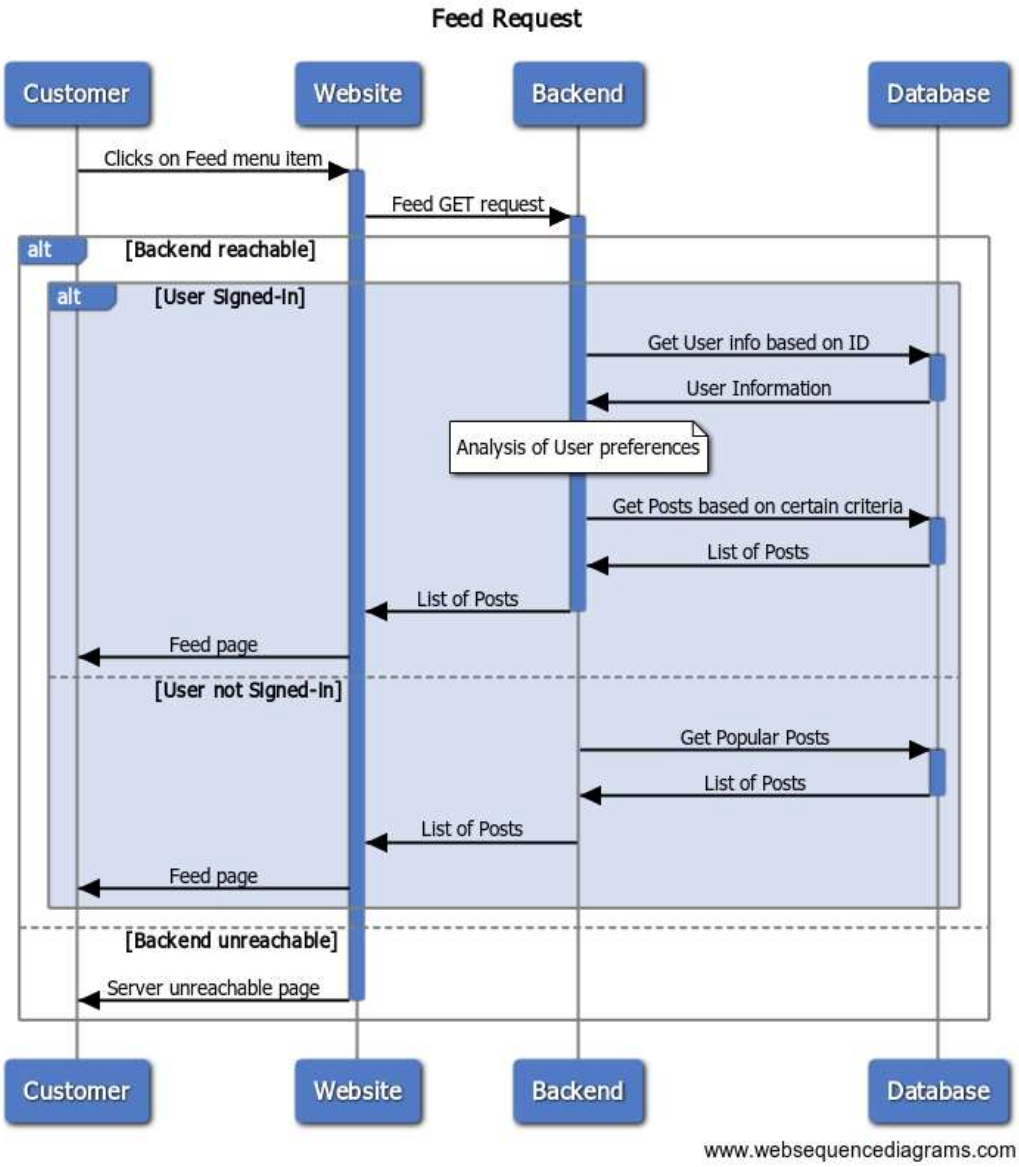
(TBA)

2.1.5. (Partial) Level-4 Repository

(TBA)

2.2. Sequence Diagrams

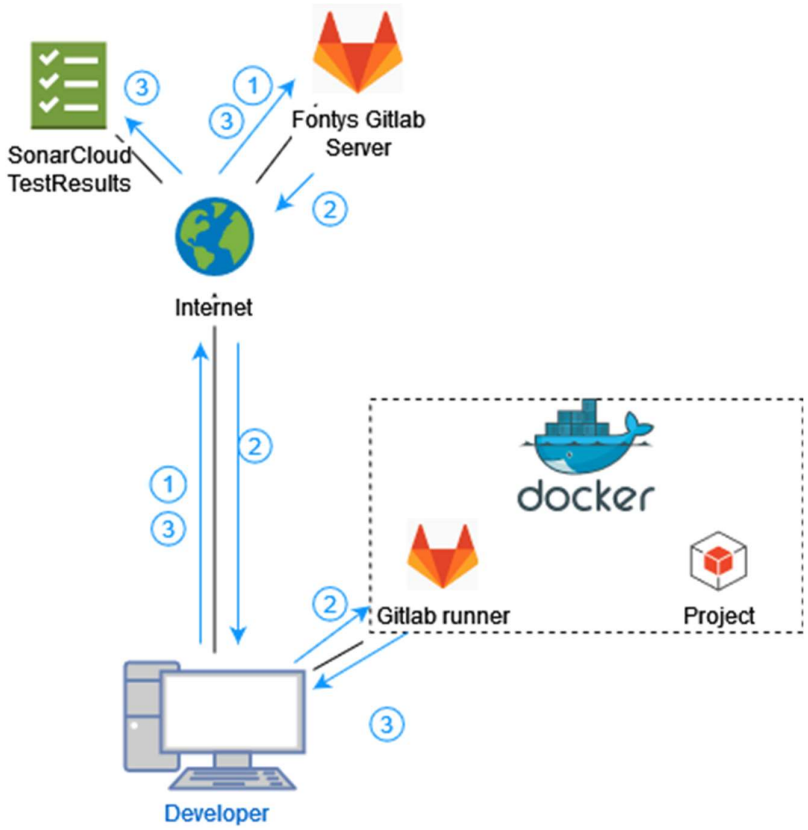
2.2.1. Feed Request



Feed requests will start when a User accesses the correct page on the Website, then Website will initiate a GET request to the backend which will process the request, fetch User information from the Database, analyse the user data to determine the parameters the query for Posts should have (find relevant posts for the user), get those posts from the database, process them, and return them to the Website.

Alternatively if the User is not signed in they will receive generic Popular Feeds and the request for the users data from the database is skipped.

2.1. CI/CD Pipeline



This diagram shows the current CI/CD pipeline separated in 3 steps.

1. A developer pushes a commit to the GitLab server (Git repository) that is hosted at Fontys. Then the server executes the steps in the pipeline.
2. It looks for an available GitLab Runner (in this case the computer of the same developer) and starts the stages. The pipeline file is written for a runner running inside a docker container, if the Runner is not running inside docker then the 3 stage, *Docker build*, will fail.
3. In the Runner the stages execute one after the other
    a. The first one builds the project with Gradle and saves the .jar file
    b. The second one runs the project tests
    c. The third runs SonarQube task and sends the results to a Sonar Cloud instance hosted by them.
    d. The final stage tries to build a Dockerfile.

    Of these, only SonarQube is allowed to fail since it requires 80% coverage and the tests are not quite there yet.