Stack Underflow

SENG2021 - REQUIREMENTS AND DESIGN WORKSHOP 21T1

DELIVERABLE 2

Dimitri Tsardakas	z5259123
Reinier de Leon	z5257456
Jamie Clarisse Domingo	z5205571
Jiaqi Zhu	z5261703
Atharv Damle	z5232949

‘

Table of Contents

Short Summary

Motivation

Ever since the pandemic made travel almost impossible (also people who never had the ability to travel) people across the world felt lonely locked up in their houses. As a result, people who love travelling and other people in general felt disconnected from society and the world.

In addition, the importance of music and its true meaning to an individual is often overlooked in modern web applications such as soundcloud, spotify, facebook. Music brings out emotions and is often intertwined with past experiences such as travel, culture, history, and enjoyment.

Our platform is purely focused on providing users with a place to share music and explain its meaning to them from all around the world. Being dedicated to sharing these important personal thoughts, our users will get the chance to be heard without judgement that is often found in more popular websites.

Our Solution

Our application directly focuses on providing users with a place to share personal experiences through music. The main feature of our application is an interactive world map where a user will be able to see tags on multiple locations. These tags are posts people have shared from all around the world. Each tag contains an attached spotify song, an image and a personal message written by each user.

Within our application, a user can also explore different cultures and find new music from all around the world, as each song posted will provide its meaning and interpretation alongside. People are redundant to music types from other cultures that they are not interested in, our application makes it easier for users to find understanding within these new genres.

Part 1: Software Architecture

The Architecture diagram is available at:
https://github.com/AD9000/Stack_Underflow/blob/2a369cdc28ab51731460f8c4676f0c5df45bc6f3/Deliverables/Deliverable%202/architecture_diagram.svg

If there are errors in loading images, kindly take a look at
https://drive.google.com/file/d/1Ne0_6oBHXMm2xVSRbQJIYiKuwC-4gzKr/view?usp=sharing

To help describe and visualise the software architecture of our application, we have attached a software architecture diagram and detailed our chosen external data sources, software components, languages, libraries and machine requirements, as well as justifications for why we have chosen them, below.

What external data sources will your system be accessing?

The external data sources our application will access are the following application programming interfaces (API): Spotify API and Mapbox API.

Spotify Api

The Spotify API is a free web API that provides access to JSON metadata about music, artists, albums and a user's Spotify data. It also provides a JavaScript library that allows developers to create a music player and play Spotify audio tracks straight from the browser.

Mapbox API

The Mapbox API is a multi-functional web service API that provides four distinct services: Maps, Navigation, Search, and Accounts.

Software components: the selected Web stack showing major software components that comprise your solution. These will include both components that need to be developed and third-party components (e.g. web browser).

Our application will consist of the following software components: Browser, Frontend, Backend Database, and Web Hosting.

Browser

Users interact with the app using their browser of choice (Safari, Firefox, Google Chrome). Since the frontend (discussed later) is packaged as a web application, the UI is rendered in the user's browser, from where they can access the different features provided by our application.

Frontend

The frontend consists of the elements and technologies used to render the user interface in their browser. This is the visual component of the application, and is developed with user accessibility and aesthetics in mind for the best user experience possible.

Backend
The backend acts as a bridge between the frontend that renders the information and the data sources (the APIs) that provide that information. The backend itself is modelled as a RESTful API, and the frontend can access the features by simply making an appropriate request.

Database
The database is used to store the data that powers the website. This includes information about tags, likes, and data important for maintaining user accounts. This allows the user to customize their experience on the website and lets them access their data whenever they log into their account.

Web Hosting
This is the place (or the hardware) where the application itself is hosted. This is the place from where the frontend code is served to the user, where the backend runs and accesses the information archives, where the database lives and provides required data.
Note: The deployment for the frontend and the backend is done on *separate* servers to reduce coupling between them.

Relating choices to components: decide which language should be used for which component of the software architecture. This will be largely determined by the Web stack but at the same time you can make variations

Frontend

- *Typescript (https://www.typescriptlang.org/)*
  Typescript can be described as a superset of JavaScript that extends its functionality by providing static type checking.

- *React (https://reactjs.org/)*
  React is a JavaScript library that is primarily used for building UI components and developing user interfaces. In our application, we will use React to implement reusable and professional front end components.

- *Material UI (https://material-ui.com/)*
  This simple and customizable component library will be used together with React to develop an accessible and aesthetically pleasing interface for our users.

Backend

- *Python3 (https://www.python.org/)*
  Python3 is an interpreted, high-level and general-purpose programming language that everyone in our team is very experienced with. It will be used to implement the majority of our backend.

- *FastAPI (https://fastapi.tiangolo.com/)*

FastAPI is a fast, high-performance web framework for building API's with Python 3.6+. It will enable us to use a REST interface to implement the main building blocks of our application.

Database

- *SQL*
  SQL (Structured Query Language) will allow us to access and manipulate the data in our database.

- *SQLite (https://docs.python.org/3/library/sqlite3.html)*
  SQLite is a fast, light relational database management system.

- *SQLAlchemy (https://www.sqlalchemy.org/)*
  SQLAlchemy is an object-relational mapping tool that will allow us to easily access and interact with our database as objects rather than relational database tables, abstracting away the SQL code itself, and making it more maintainable.

Web Hosting

- *Docker (https://www.docker.com)*
  Docker is an open source containerization platform. Docker enables developers to package applications into containers—standardized executable components that combine application source code with all the operating system (OS) libraries and dependencies required to run the code in any environment.

- *Google Cloud (https://cloud.google.com/)*
  Google cloud is a cloud computing platform, which in simple words, means it provides the hardware for the application to run on, but with zero maintenance costs. We use the compute engine in particular, which is then used to run the Docker containers.

The choice of a platform: decide on machine or machines requirements (Linux, Windows etc.) for the final system

Client Platform

Our application will support Windows, Mac and Linux operating systems. Though the latest version of Google Chrome will be the primary web browser we will test our application on, we will also provide support for the latest version of Safari, Mozilla Firefox, and Microsoft Edge 13.

Website Hosting

All the components of the website will run on a lightweight Linux virtual machine. The applications themselves will be containerized using Docker which means there are no fixed requirements for the processing power and RAM, and can be scaled up or down easily using a container management system such as Kubernetes if needed. We will use Google Cloud to set up and deploy these containers as it makes the process of deployment much easier.

Make a summary of the key benefits/achievements of your architectural choices.

External Data Sources

We chose to use the Spotify API since Spotify is one of the most used music streaming services in the world, thus allowing us to reach a multitude of users who already own a Spotify account. Utilising this data source in our application will give our users access to Spotify’s vast range of audio content from songs, audio books to podcasts. The Spotify API also has extensive capabilities that will allow us to fulfil the requirements stated in our user stories, particularly its information and playback features. Our users will be able to access songs in their playlists and favourite albums, including the respective album cover images, and will also have the ability to play these songs in their own web browser. Furthermore, the Spotify API is free to use and it has detailed, comprehensive documentation with many easily-accessible tutorials, thus making it easy to learn and implement.

We chose the Mapbox API as it provides all the functionality our application requires, providing services such as: Maps, Navigation, Search, and Accounts. The services we will be using are the Search and Maps services.
The Mapbox Search Service implements geocoding operations that, when employed in our application, will allow users to search for a wide variety of locations by ‘country’, ‘region’, ‘district’, ‘place’ or ‘address’, which they can then attach to a Spotify song when creating a new post.
The chosen API also provides a Maps Service that can be used to create and request maps. Our application will utilise the Mapbox Maps Service, in conjunction with the Search Service, to render map tiles on the browser. This allows us to achieve and implement multiple user stories including allowing users to view the posts of others, and visualise their respective locations. Other similar API’s were considered, such as the Google Maps API and OpenStreetMaps API, however, we found Mapbox to be the cheapest to use, with readable and easy to understand documentation. Due to the limited time restraints of this project, we need to prioritise efficiency, and thus employing a simple, manageable API is ideal.

Frontend

When deciding on our web stack for the frontend, both Typescript and JavaScript were initially considered. However, although we are more familiar with JavaScript, we eventually settled on Typescript since it is an overall ideal choice for large software projects due to its advanced type system and extra features. Implementing our system in Typescript will improve the readability and maintainability of our code, help us to debug our code effectively and efficiently, and minimise errors that may be caused by type ambiguity during compile time. Furthermore, since it is significantly similar to JavaScript, it will be easy to pick up for those in the team who have not used it before.

We have chosen to use the React JavaScript library as most of our team have used this language this past and are quite familiar with it. Using this language will help us to build reusable components quickly, improving our productivity and enabling us to achieve more in a short amount of time.

As mentioned previously, fast development is a priority of ours. Using the components in the Material UI library will give our frontend a sleek, professional look with very minimal effort. Though we want to prioritise functionality over aesthetics, using Material UI will allow us to achieve both within time restraints.

Backend

Python was our first choice of language to employ in our backend as every member of the team is experienced with it. It's clear and straightforward syntax helps with readability and maintainability. Employed in our application, it will expose a RESTful API to the frontend, which then enables us to abstract away all the complexity, while providing a simple, and well-documented interface for the frontend to interact with. This also allows us to seamlessly add more features to the APIs, such as user account management, by simply adding another endpoint for the frontend to call.

FastAPI was chosen for our application due to its high performance, as it is said to be one of the fastest Python frameworks available. Unlike Flask, it can detect invalid data types at runtime, thus reducing the possibility for bugs, and compared to Django, takes minimal time and effort to implement. FastAPIi is easy to learn and use, facilitates fast, simple API creation, can generate documentation on the go, and also provides a testing page for quicker development. Overall, it is an efficient, productive choice for the implementation of our backend.

We decided that it was necessary to use an object-relational mapping (ORM) to help us access the database, especially since Python is an object-oriented language. For an ORM, we settled on SQLAlchemy as it operates smoothly with Python and has a similar style, thus reducing the complexity and learning curve of our project. Furthermore, it will simplify our database queries, help us to keep our database organised, and make our code more readable and maintainable.

Database

We chose to use SQL for our database as it is a familiar language to our team and we have previous experience developing SQL databases. It is an effective and efficient language, enabling us to query and manipulate the data as we see fit.

SQLite was our chosen database management system as it is simple and uncomplicated to use, robust as it encapsulates our data into a single file, and is simple to employ since there is no need for setup or configuration. Since we are only required to implement a minimum viable product, SQLite is suitable for our needs. However, if we were to scale our application up and deploy it, we may then consider PostgreSQL.

Web Hosting
We chose Docker as our containerization solution to decrease coupling with the operating system itself. Docker installs what is essentially a lighter version of a linux virtual machine on the server it is running before it runs the application, making it platform independent. The frontend, backend and database are also deployed in separate containers for decoupling purposes. If by any chance any one of these three containers go down, it would not affect the functioning of the other two. Furthermore, if we use Kubernetes for container management, the traffic from the user can be simply redirected to another container running the frontend, which allows for even lower downtime.

Each of the docker containers will then be hosted in google cloud using the cloud compute engine by google. Cloud computing is the best hosting solution for the app as there are no hardware maintenance costs, along with high scalability combined with the docker containers, since the hardware can also be easily scaled up using the google cloud console. It also allows us to easily add in kubernetes management for the docker containers in the future using the Google Kubernetes Engine (GKE).

Client Platform Requirements
The choices for providing support to Mac, Linux and Windows, and browser support for Safari, Mozilla Firefox, and Microsoft Edge 13 were decided based on the operating systems and browsers supported by our external data sources, Spotify API and Mapbox API. We wanted to ensure that our application can accommodate and fulfil the user's needs without any issues.

Part 2: User Stories & Sequence Diagrams

See PDF file in the folder - Deliverable 2 on Github.
(https://github.com/AD9000/Stack_Underflow/blob/a0a2788fa7448e47cae3c5cc8bc82c747bac2840/Deliverables/Deliverable%202/User%20Stories%20&%20Sequence%20Diagrams.pdf)