Private Data Hub

Documentation

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Abstract

This documentation details the development of Private Data Hub, a platform that assist users to retrieve and analyze their personal data from various service providers, including Google, Spotify, and Netflix. The primary focus of this project was to provide users with an overview of the types of data being collected about them. This information is represented using graphs and lists that deliver interesting insights into their personal data. The platform is built with extensibility in mind, enabling future support for additional providers. It features a FastAPI and MongoDB-based backend for efficient data handling and a Svelte frontend for an intuitive user experience. Key features include data imports, data analysis, and efficient handling of large datasets. This document details the technical implementation, system architecture, and overall approach to delivering the Private Data Hub application.

Part I

Introduction

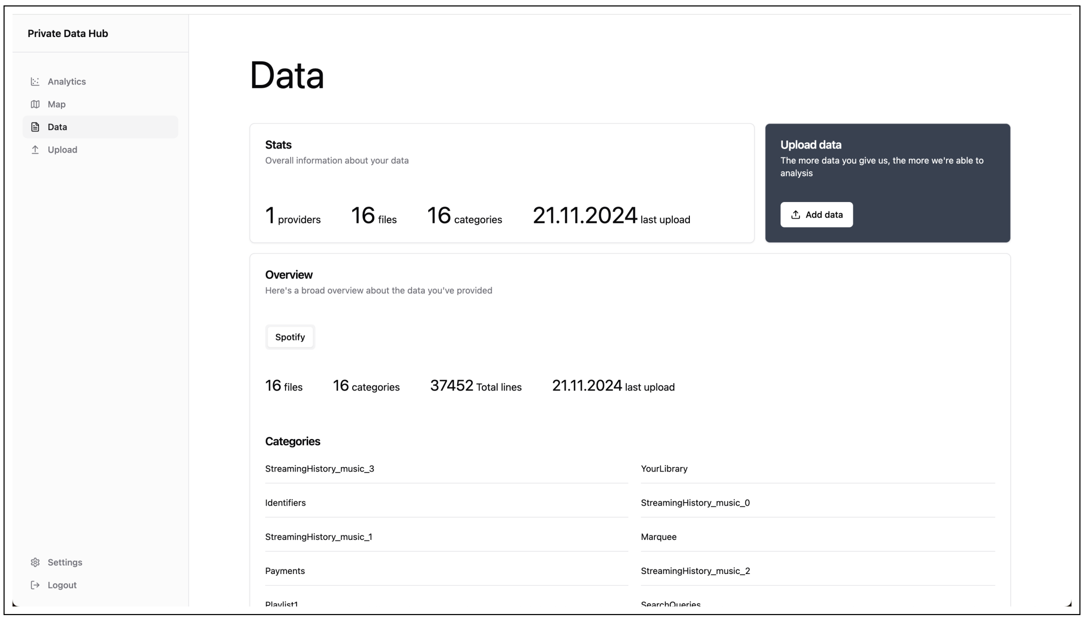
Management Summary

Thanks to data protection laws, individuals can request their personal data from companies. This includes data collected by companies like Netflix, Google, Facebook, Tesla, etc. Analyzing these data dumps is very time-consuming because there is no standard format, and the data exports are provided in different ways.
Private Data Hub is a web application that enables individuals to better understand their personal data requested from Google, Spotify or Netflix. Developed as part of a semester project at OST Eastern Switzerland University of Applied Sciences, the platform aims to provide users with an interesting insight into their collected data.

Purpose of the Platform

In today's digital age, users generate enormous amounts of personal data across various online platforms. However, understanding what data is collected and how it is used often remains unclear. Private Data Hub tries to provide users with the necessary tools to better understand their data dumps. Its key features for users include:

- Assistance with requesting personal data.
- An overview for personal data in a user-friendly dashboard.
- Clear visualizations, such as graphs, lists or maps to provide insights into the data.



Data

Stats
Overall information about your data

1 providers 16 files 16 categories 21.11.2024 last upload

Upload data
The more data you give us, the more we're able to analysis

Overview
Here's a broad overview about the data you've provided

16 files 16 categories 37452 Total lines 21.11.2024 last upload

Categories

StreamingHistory_music_3	YourLibrary
Identifiers	StreamingHistory_music_0
StreamingHistory_music_1	Marquee
Payments	StreamingHistory_music_2
Playlist1	SearchQueries

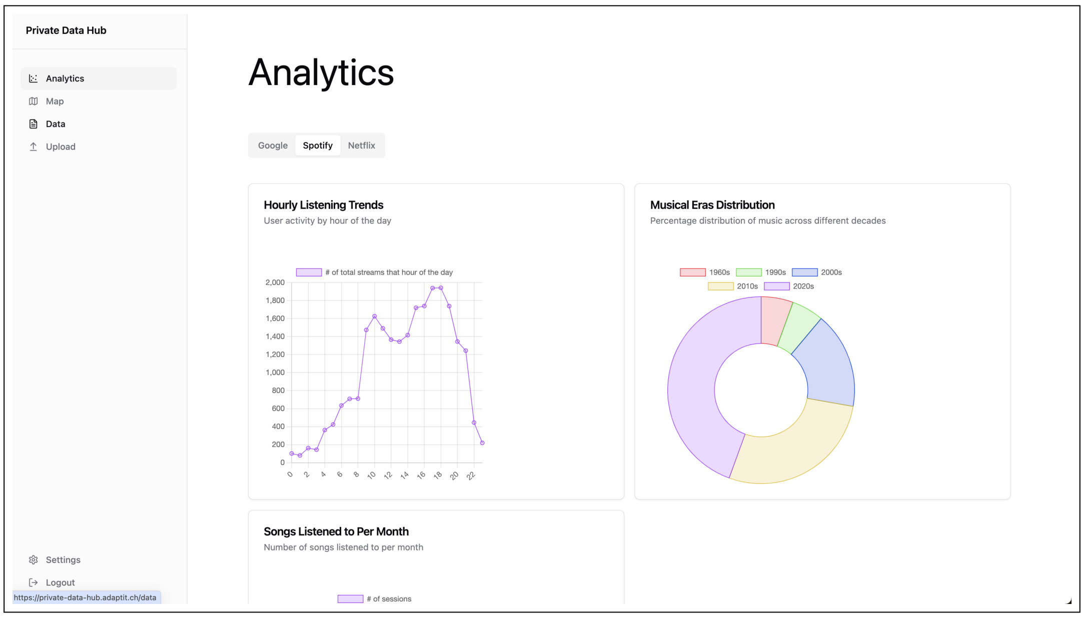
Figure 1: Dashboard of the Private Data Hub Application

Target Audience and Benefits

The platform is ideal for all individuals who are interested in gaining insights into their digital footprint. The app offers instructions on how to request user data and provides clear visualizations and insights.

Key Features and Results

- **Data Analysis and Visualization:** The platform provides graphs and charts that highlight usage trends, such as most-watched shows on Netflix or favorite music genres on Spotify.
- **Secure Data Handling:** All data uploads are encrypted.
- **Compatibility:** The application can analyse files in various formats (e.g., JSON, CSV, ICS).



Private Data Hub
Analytics
Map
Data
Upload
Settings
Logout
https://private-data-hub.adaptit.ch/data
Analytics
Google
Spotify
Netflix
Hourly Listening Trends
User activity by hour of the day
# of total streams that hour of the day
Musical Eras Distribution
Percentage distribution of music across different decades
1960s
1990s
2000s
2010s
2020s
Songs Listened to Per Month
Number of songs listened to per month
# of sessions

Figure 2: Analytics page of the Private Data Hub Application

Challenges Addressed

- **Data Diversity:** Developed flexible algorithms to process varying data formats from multiple providers.
- **Security:** Ensured encrypted communication and password protection to secure sensitive information.
- **Extensibility:** Designed to support additional data sources and analysis algorithms in the future.

Future Outlook

When designing the application architecture, emphasis was placed on ensuring extensibility to accommodate the analysis of additional data providers in the future. Potential next steps include:

- Expanding support to include additional data sources and formats.
- Introducing artificial intelligence to uncover patterns and relationships in datasets.

Contents

Part II

Product Documentation

Chapter 1

Starting point

1.1 Motivation

The motivation for this project comes from the growing attention to the personal data being collected by various platforms. This data is often vast and complex, making it difficult for individuals to understand what information is collected. With our application, we aim to empower users by helping them better understand what is being collected about them and the insights that can be derived from it.

1.2 Type of Work

This work focuses on the creation of a functional platform that empowers users to better understand their personal data.

1.3 Starting Point

When we began developing Private Data Hub, we started with a clean slate. This allowed us the freedom to design and build the system precisely according to our vision, free from the constraints of legacy decisions or existing structures.

1.4 Limitations and Differentiation

Private Data Hub currently includes a few generic algorithms designed to handle data from a wide range of potential providers. However, our primary focus is on analyzing the data of specific providers to deliver deeper and more meaningful insights. At present, the platform supports data from Google, Netflix, and Spotify.

However the platform has been designed with extensibility in mind, ensuring that new providers can be integrated in the future.

Chapter 2

Requirements

2.1 Functional Requirements

FR 1: Import of Relevant Data from Different Providers into a Database

The data can be imported via the upload page. The data formats JSON, CSV, MBOX and specific HTML files are currently supported. In addition, the source of the respective datadump is specified during import, which is currently limited to Netflix, Spotify and Google. However, the upload is not limited to these companies.

More details in the chapter 4.3.4.

FR 2: Account per User

We have implemented user management system that ensures secure and personalized data access. Users can create an account through the **/signup** page and authenticate via the **/login** page.

FR 3: User Support When Requesting Personal Data

Instructions for all supported providers are displayed on the "upload" page.

FR 4: Display of the Data on a Dashboard

On the "Dashboard" page, an overview of the uploaded data is displayed for all supported providers and interesting facts about the scope of the dumps are shown. For a more detailed and visual analysis, users can navigate to the "Analytics" page, where charts and graphs are displayed.

2.2 Optional Requirements

Integration of an LLM to Answer Questions About One's Own Data

We opted not to integrate a Large Language Model (LLM) due to the limited contextual scope and our primary focus on establishing the application's core functionalities and conducting manual data analysis. After evaluating the potential benefits, we concluded that adding an LLM would not significantly enhance the application's value at this stage.

Modern, Appealing, Responsive UI

We successfully developed a modern, visually appealing, and responsive user interface by utilizing ShadCN. This library enabled us to create an engaging and user-friendly design that seamlessly adapts to various devices and screen sizes.

Sharing Data and Comparing with Other Users

Due to the high sensitivity of the processed data, we consider this feature to be undesirable and have therefore not implemented it due to other priorities.

Import of Unstructured Data from Unknown Providers

As previously mentioned, the import is limited by the frontend to Netflix, Spotify and Google. However, the import algorithm in the backend can be used for any data dump. More details in the chapter 4.3.4.

AI for Identifying Relationships Between Different Datasets

We did not implement AI-driven methods to identify relationships between different datasets within the scope of this project. The complexity and resource requirements for such functionality exceeded our current project constraints.

Simple Data Analysis

After importing the data, a data analysis is performed for all available providers. We have created different algorithms for each provider to evaluate interesting statistics. The results can be viewed on the “analytics” page of our application. More details in the chapter 4.3.6.

Semi-Automated Import of Data Directly from the Provider

We developed a semi-automated data import feature by providing a detailed, step-by-step guide for retrieving data directly from the provider. This approach streamlines the data integration process, ensuring efficiency and accuracy while allowing users to manually intervene when necessary.

Data Visualization Using Charts

On the “analytics” page of our application, the results of the evaluations are predominantly represented in graphs. More details in the chapter 4.3.6.

Search Function: Full-Text Search Across All Imported Data

The implementation of a full-text search functionality was not pursued in the current phase of the project. Due to the limited scope of our data analysis, we determined that developing a comprehensive search feature would not provide substantial value.

Simple Data Categorization

During the import process, the data is assigned to specific categories by Google, for example, in order to simplify the analysis. More details in the chapter 6.4.

2.3 Non-Functional Requirements

NFR 1: Priority Implementation According to Customer Agreement

We successfully prioritized and implemented key features in accordance with the customer agreement.

NFR 2: Backend Should Handle 1,000 Requests per Minute

To ensure that our backend can handle the required load of 1,000 requests per minute, we performed load testing on the login route using the tool `siege`. This tool allows us to simulate multiple concurrent users and test how the server handles traffic under stress.

We specifically tested the login route with the following command:

```
siege -c 100 -r 10 -f login_form.txt https://private-data-hub.adaptit.ch/login
```

This command simulates 100 concurrent users (`-c 100`) who each perform 10 requests (`-r 10`), resulting in a total of 1,000 requests. The results from this load test are as follows:

- **Transactions:** 2000 hits
- **Availability:** 100.00%
- **Elapsed time:** 6.01 secs
- **Data transferred:** 8.51 MB
- **Response time:** 0.2 secs
- **Transaction rate:** 332.78 trans/sec
- **Successful transactions:** 2000
- **Failed transactions:** 0
- **Longest transaction:** 2.46 secs
- **Shortest transaction:** 0.08 secs

These results demonstrate that the backend can successfully handle 1,000 requests per minute, meeting the non-functional requirement for load handling. The availability and response times are within acceptable limits, and there were no failed transactions during the test.

NFR 3: Page Load Time Should Not Exceed 200ms

We have thoroughly tested the application's performance using Google Dev Tools, achieving an average page load time of approximately 60ms. This performance metric comfortably meets the requirement of not exceeding 200ms, ensuring a fast and responsive user experience.

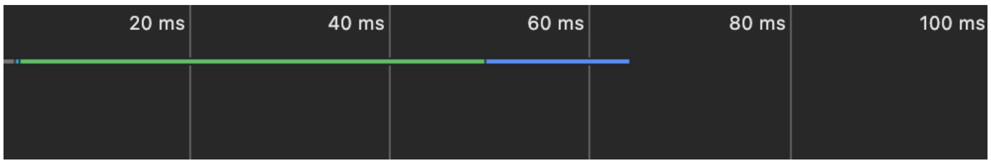


Figure 2.1: Page load test via Chrome Dev Tools

NFR 4: Responsive Design for Both Tablet and Desktop

We have successfully implemented a responsive design for both tablet, desktop and even mobile devices. By using modern design frameworks our interface seamlessly adapts to various screen sizes and resolutions.

More details in the chapter 5.2.

NFR 5: Cross-Browser Compatibility

We have ensured cross-browser compatibility, guaranteeing that the application functions seamlessly across all major web browsers, including Chrome, Firefox, Safari, and Edge.

NFR 6: Domain Accessibility

Our application is hosted and accessible through the dedicated domain `https://private-data-hub.adaptit.ch/`.

NFR 7: UI Evaluation by Test Users

We provided the application to five individuals for testing 7.4.

NFR 8: Database Capacity

Due to the limited 50GB storage space available on our DigitalOcean droplet, we were not able to test this configuration at full scale on the server. However, theoretically, the application can handle any number of datasets and users, as long as enough storage is available.

NFR 9: Error Handling Without System Crashes

We have implemented error handling strategies to minimize the risk of system crashes and ensure the application manages errors effectively. While we have designed the logic to anticipate and handle potential issues, the limited data available for testing means we cannot guarantee that the application will never experience crashes.

More details about the upload in the chapter 6.3.

NFR 10: Error Logging

We have implemented error logging mechanisms in both the frontend and backend of the application. This ensures that arising issues are effectively captured by the system and can be traced.

More details in the chapter 5.9 and 6.8.

NFR 11: Encrypted Communication

To protect sensitive information, we utilize Caddy as an HTTPS-enforcing reverse proxy, ensuring all client-server communication is encrypted. Caddy automatically manages SSL certificates and redirects all HTTP traffic to HTTPS, safeguarding user data during transit and maintaining compliance with security standards.

More details in the chapter 6.7.

NFR 12: Input Validation

We have implemented input validation mechanisms within the login and signup processes. Specifically, during user registration, the system checks whether the username already exists to prevent duplicate accounts. Additionally, we enforce password strength by ensuring that passwords meet minimum length requirements.

NFR 13: Password Security

We have implemented password security measures by storing all passwords as hashed values and ensuring their secure transmission from the client to the server. All password data is transmitted over encrypted channels, safeguarding it against interception and ensuring that user information remains confidential throughout the authentication process.

More details in the chapter 6.7.

NFR 14: Compliance with Data Protection Regulations

We provide users with the ability to access and delete their personal information. We also guarantee that the users data will not be used for any purpose other than those explicitly intended by the platform. Furthermore, the data will not be viewed by anyone other than the user.

NFR 15: Access Control

We have implemented access control mechanisms to ensure that each user can only view and interact with their authorized data. Upon logging into the web application, users are granted access exclusively to their own information.

NFR 16: Modular Backend Business Logic

We have structured the backend with a modular architecture, compartmentalizing business logic into distinct, independent components. By decoupling core functionalities, we facilitate easier updates and extensions, ensuring that the system can adapt to evolving requirements without significant overhaul.

More details in the chapter 6.1.

NFR 17: Backend API Testing

We have implemented tests for our backend API using Python's `unittest` library.

More details in the chapter 7.

NFR 18: Deployment of Implemented Functionality

We have successfully deployed the application on DigitalOcean, accessible via the domain `https://private-data-hub.adaptit.ch/`. Utilizing Docker for containerization, we ensured a consistent and scalable deployment environment. This approach simplifies the management and orchestration of application components.

More details in the chapter 6.6.

Chapter 3

Market Analysis

3.1 Market Overview

Today, people generate large amounts of personal data online (e.g., from streaming platforms, social media, and search engines). Many users have started to realize the importance of understanding what data is being collected and taking steps to control it. This growing awareness creates an opportunity for tools like *Private Data Hub*, which help individuals to better understand their personal data.

3.2 Industry Context

3.2.1 Data Awareness

As more people use online services, their data accumulates across multiple platforms. Users are becoming more concerned about what is collected, how it is used, and how to gain insights from it.

3.2.2 Regulatory Influence

Laws such as the GDPR in Europe give individuals more rights and control over their personal data. This encourages users to look for tools that help them exercise these rights easily.

3.2.3 Emerging Interest in Personal Analytics

Beyond just privacy, many users now want to extract value from their data — understanding their own habits, usage patterns, and trends.

3.3 Target Market

3.3.1 User Profile

- Interested in insights into own data.
- Comfortable with basic technology.

3.3.2 User Motivations

- Understanding where their data comes from and how it's used.
- Easily accessing meaningful data insights without technical expertise.
- Ensuring their information remains private and secure.

3.4 Market Needs

- **Data Transparency:** Users want a clear overview of the personal data they've shared.
- **Easy Insights:** Simple dashboards with easy-to-understand charts and summaries.
- **Privacy Assurance:** Confidence that their personal data is protected.

3.5 SWOT Analysis

3.5.1 Strengths

- Easy-to-use platform
- Supports multiple data sources
- Focus on privacy

3.5.2 Weaknesses

- Depends on users obtaining their own data dumps
- Requires updates for new data formats

3.5.3 Opportunities

- Increased interest in personal data analytics
- Potential partnerships with privacy-focused organizations

3.5.4 Threats

- Changes in data access policies
- New competitors entering the market
- Data breaches impacting user trust

3.6 Potential Risks and Mitigation

3.6.1 Risks

- Regulatory challenges
- Data security threats
- Platform changes affecting data access

3.6.2 Mitigation

- Implement strong encryption and regular security audits
- Continuously update the system for evolving data sources

3.7 Conclusion

As users become more aware of their personal data and the importance of privacy, the market for tools like *Private Data Hub* is expanding. By focusing on ease of use, privacy, and comprehensive analysis, *Private Data Hub* can attract and retain users. Through ongoing improvements, it can stand out in an increasingly data-conscious environment.

Chapter 4

Design/Architektur

4.1 C4 Diagram

The C4 diagram is included to provide a clear, structured view of our software architecture, breaking down the system into three levels of detail. This helps to understand how components interact within the larger system context.[c4m]

4.1.1 Context Diagram

Our application provides users with the necessary support to request their private data from companies. The requested data can then be uploaded to the application and analyzed to get an overview of what data is collected about them.

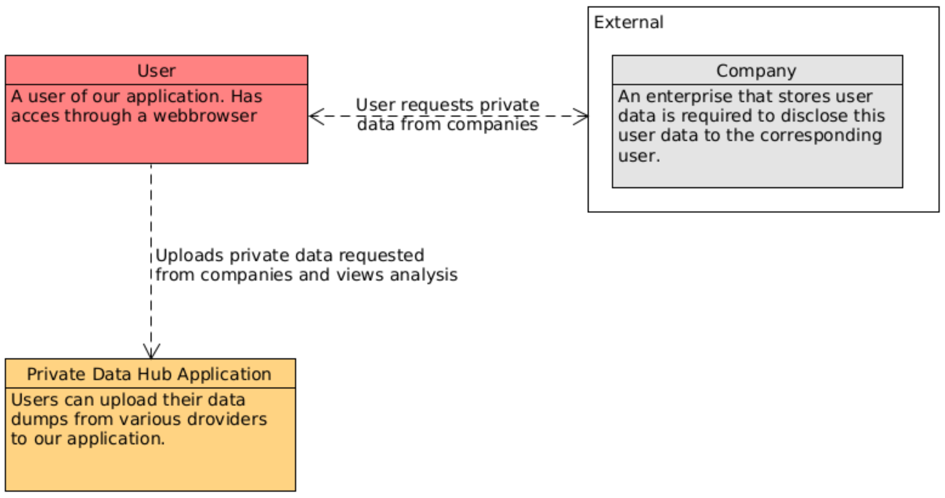

External
Company
An enterprise that stores user data is required to disclose this user data to the corresponding user.
User
A user of our application. Has acces through a webbrowser
User requests private data from companies
Uploads private data requested from companies and views analysis
Private Data Hub Application
Users can upload their data dumps from various droviders to our application.

Figure 4.1: Context Diagram

4.1.2 Container Diagram

Here you can see our application with front and backend. The frontend communicates with the backend via an api. The data processing takes place in the backend while the data is presented in an appealing way in the frontend.

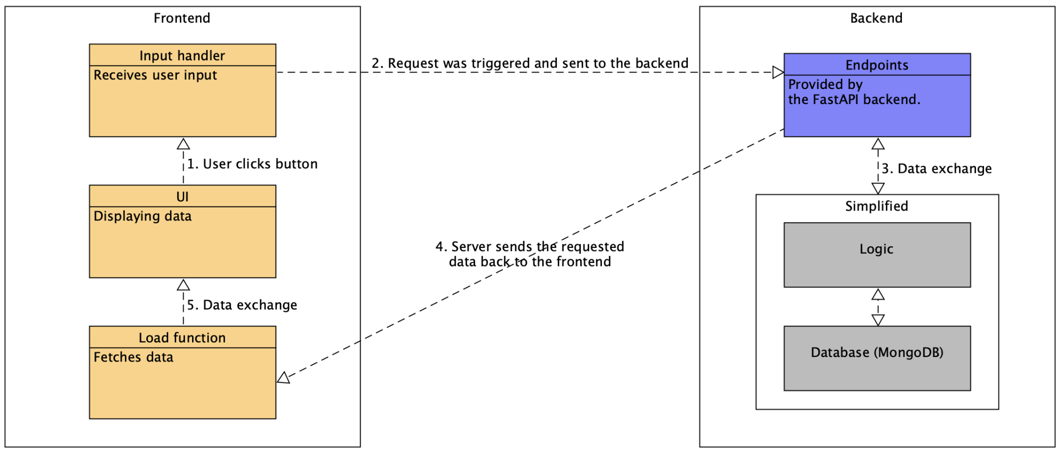


Figure 4.2: Container Diagram

4.1.3 Component Diagram

Detailing the main functional modules of our backend—such as authentication, user management, and data processing—the component diagram illustrates how these components interact with each other and with external services to deliver the application's core functionalities.

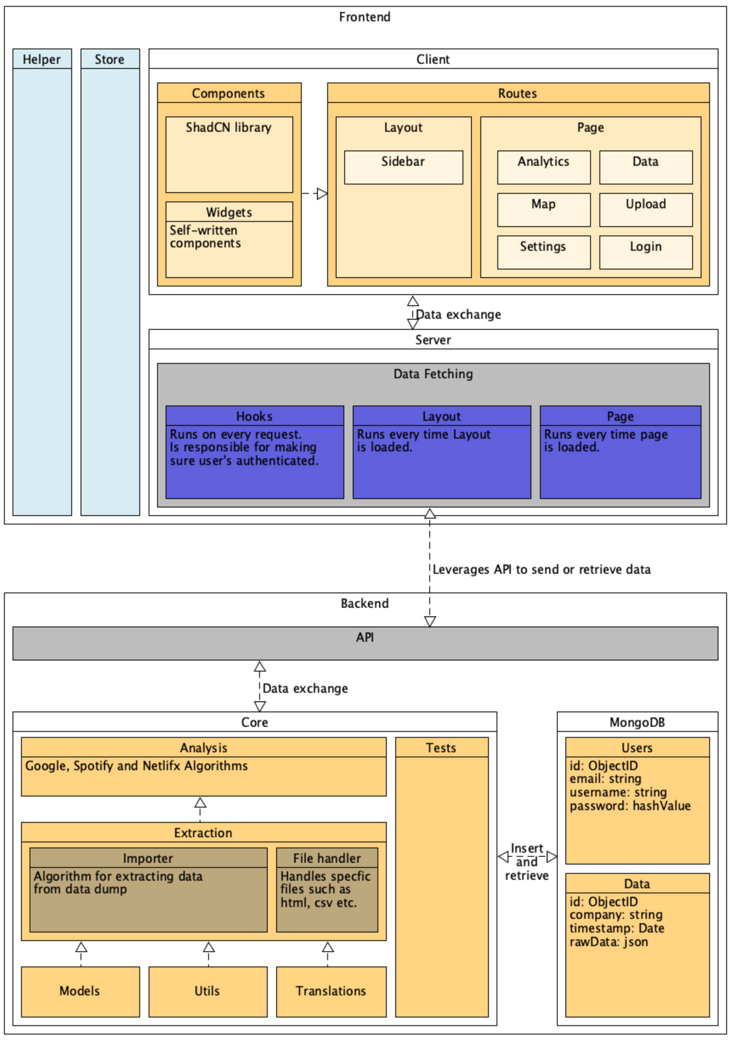


Figure 4.3: Component Diagram

4.1.4 Code Diagram

The code diagram illustrates the architecture of both the frontend and backend. This structured approach ensures maintainability, scalability, and logical organization, enabling seamless collaboration and efficient development workflows.

Frontend

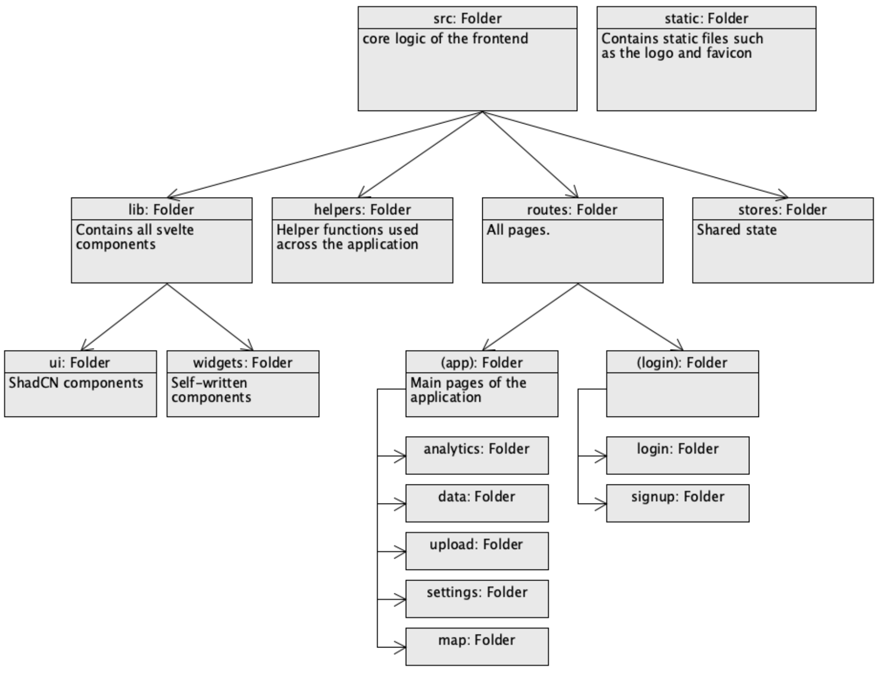


Figure 4.4: Code diagram for the frontend

Backend

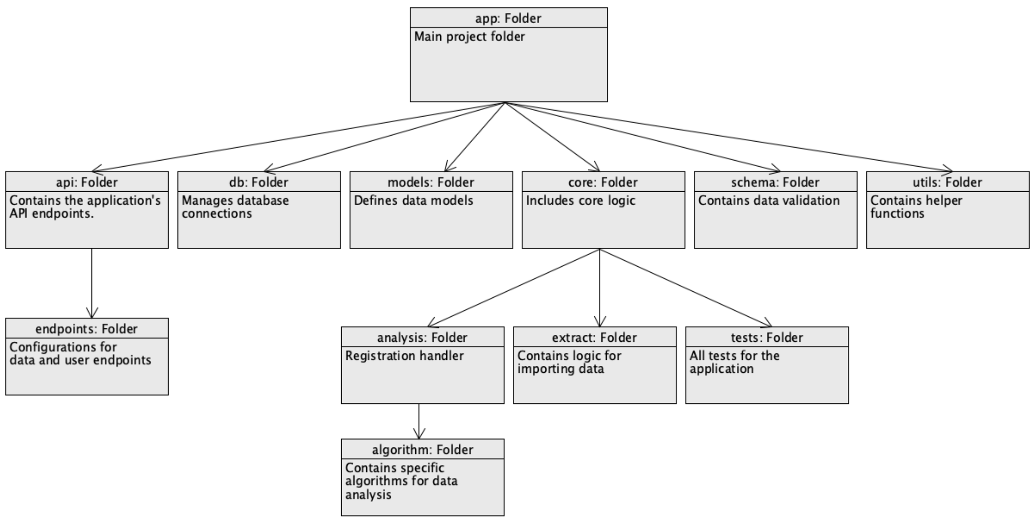


Figure 4.5: Code diagram for the backend

4.2 Frontend

4.2.1 Technologies

Svelte

Svelte is a front-end framework that compiles components into highly efficient, vanilla JavaScript code.

Why Svelte?: We chose Svelte for its simplicity and modern approach to frontend development, enabling a more straightforward and efficient build process.

ShadCN

shadcn is a collection of UI components built with Radix UI and styled with Tailwind CSS.

Why ShadCN?: We chose shadcn to accelerate development by leveraging pre-built, accessible components that integrate seamlessly with our existing stack.

Typescript

TypeScript is a superset of JavaScript that adds static typing. It helps developers catch type-related errors early in the development process, improving code quality and maintainability while still compiling to standard JavaScript.

Why Typescript?: Vanilla JavaScript can be daunting due to its error-prone nature and lack of static typing. TypeScript helps mitigate these issues by providing a type-safe development environment.

Tailwind

Tailwind CSS is a utility-first CSS framework that provides pre-built classes to quickly style components.

Why Tailwind?: Standard CSS can become difficult to manage, so we chose Tailwind for its utility-first approach, making styling more efficient and organized.

4.3 Backend

4.3.1 Technologies

Below, we briefly describe the technologies used in the backend and explain why they were chosen.

Python

Python is a versatile and easy-to-learn programming language known for its readability and wide range of libraries. It is commonly used for web development, scripting, data analysis, and automation. [pyt]

Why Python?: We chose Python for its simplicity, readability, and the extensive library ecosystem, which enabled us to develop robust backend functionality quickly and efficiently.

FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python. [fas]

Why FastAPI?: At first we wanted to implement the backend with django rest framework, but then we realized that when using mongodb, which is a prerequisite due to the nature of our data, we can only use the advantages of django in a cumbersome way. That's why we switched to FastAPI. We chose FastAPI for its speed, ease of use, and its powerful features like automatic API documentation and type-based validation, which helped us build a reliable backend.

MongoDB

MongoDB is an open source NoSQL database management program.[tec]

Why MongoDB?: Document databases store information in documents. Document databases have rich APIs and query languages that can be used to execute the CRUD (create, read, update, and delete) operations. They have flexible schemas, allowing developers to easily evolve their data models as their application requirements change. [Mon] Due to the flexible nature of the data to be processed, a document-based structure is best suited to our use case.

4.3.2 Overview

To provide a brief overview of how the backend operates, we outline all the steps performed when a user uploads data.

1. The user uploads data through a form on the frontend. The upload request contains the company name and actual data in chunks.
2. Once the data dump has been fully received by the backend, the backend extracts the file and invokes the import algorithm.
3. The import algorithm traverses the files recursively and stores all supported file formats in the database.
4. After the import is successfully completed, the analysis of the uploaded data is triggered. The analysis uses defined algorithms to examine the user data.

4.3.3 Upload

The upload mechanism must handle large files, up to 10 GB, especially from Google. To improve upload reliability, we implemented a chunking mechanism. This approach splits the uploaded file on the client side into smaller parts, which are sent individually. The backend then reassembles these parts, allowing the file to be used as if it were the original.

The the implementation is described in chapter [6.3].

4.3.4 Importing user data

The heart of our application is the import algorithm. To ensure the algorithm does not limit the app's extensibility, it was designed to be as general as possible. The algorithm is capable of importing supported file formats from any provider, regardless of the folder structure. It can also recognize data that has already been imported, allowing the same data dumps to be uploaded multiple times without duplicating unnecessary data in the database. Furthermore, when a new data dump is uploaded, only the data that is actually new is added to the database.

The implementation is described in chapter [6.4].

4.3.5 MongoDB Structure

Document databases store information in documents. Each document typically contains information about one object and any related metadata. Documents with similar contents are grouped together in collections. [Mon] We have collections for the users' data and for the users themselves. To keep it as simple as possible, we store the extracted content in the rawData field of the document. A graphical representation of the documents in the database can be found in the C4 Container diagram. [4.3]

4.3.6 Data Analysis

Another important part of our application is the analysis of user data. To make this easily extensible, we wanted a central place where every algorithm to be performed is registered. This separates the actual implementation into another file, making the analysis mechanism more modular. The implementation is described in this chapter [6.5]

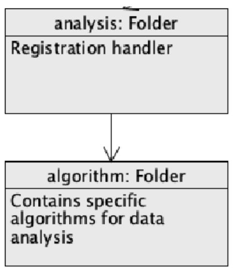


Figure 4.6: Code diagram zoomed in on analysis

Chapter 5

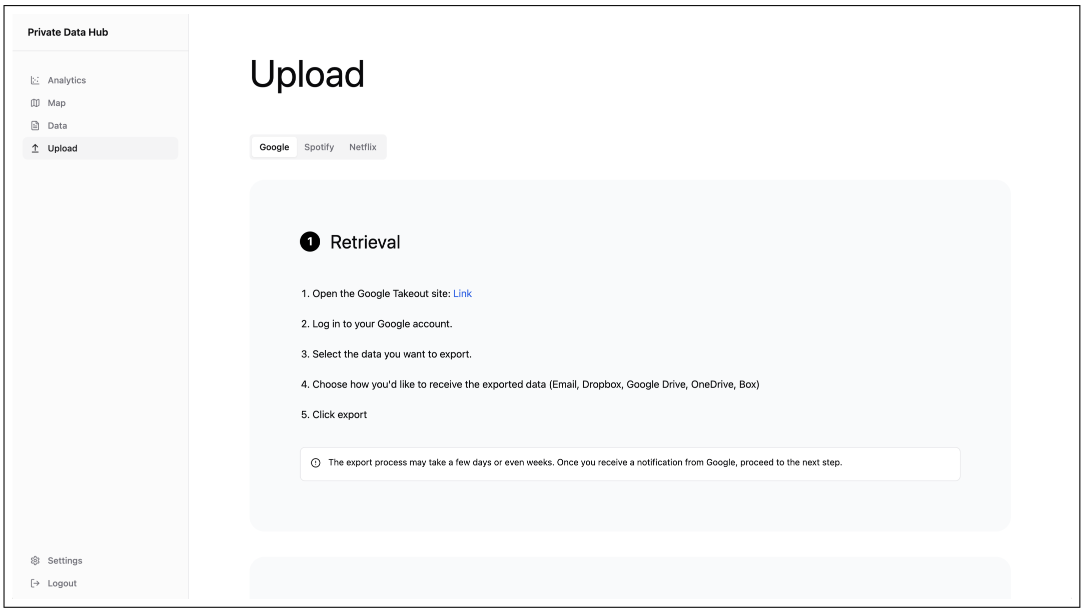
Frontend Implementation

5.1 Screens

Our goal was to create an intuitive experience that feels both modern and easy to navigate, ensuring a clean user interface that allows users to focus without unnecessary distractions.

5.1.1 Desktop

Upload



1. Open the Google Takeout site: Link

2. Log in to your Google account.

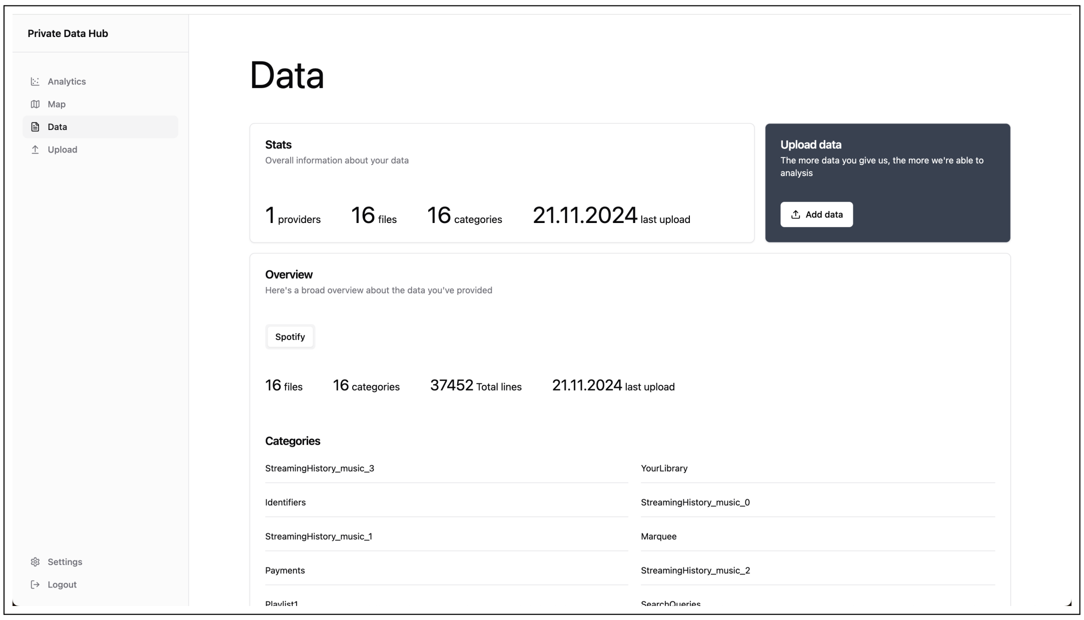
3. Select the data you want to export.

4. Choose how you'd like to receive the exported data (Email, Dropbox, Google Drive, OneDrive, Box)

5. Click export

The export process may take a few days or even weeks. Once you receive a notification from Google, proceed to the next step.

Data


Private Data Hub

Analytics
Map
Data
Upload

Settings
Logout

Data

Stats
Overall information about your data

1 providers 16 files 16 categories 21.11.2024 last upload

Upload data
The more data you give us, the more we're able to analysis

Add data

Overview
Here's a broad overview about the data you've provided

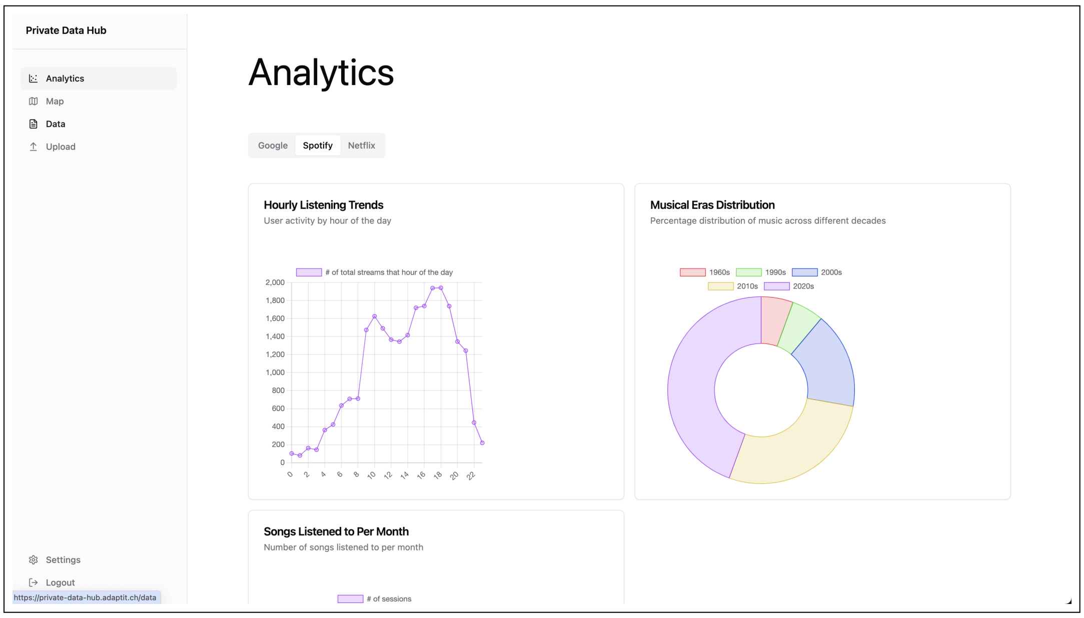
Spotify

16 files 16 categories 37452 Total lines 21.11.2024 last upload

Categories

StreamingHistory_music_3	YourLibrary
Identifiers	StreamingHistory_music_0
StreamingHistory_music_1	Marquee
Payments	StreamingHistory_music_2
Playlist1	SearchQueries

Analytics


Private Data Hub

Analytics
Map
Data
Upload

Settings
Logout

https://private-data-hub.adaptit.ch/data

Analytics

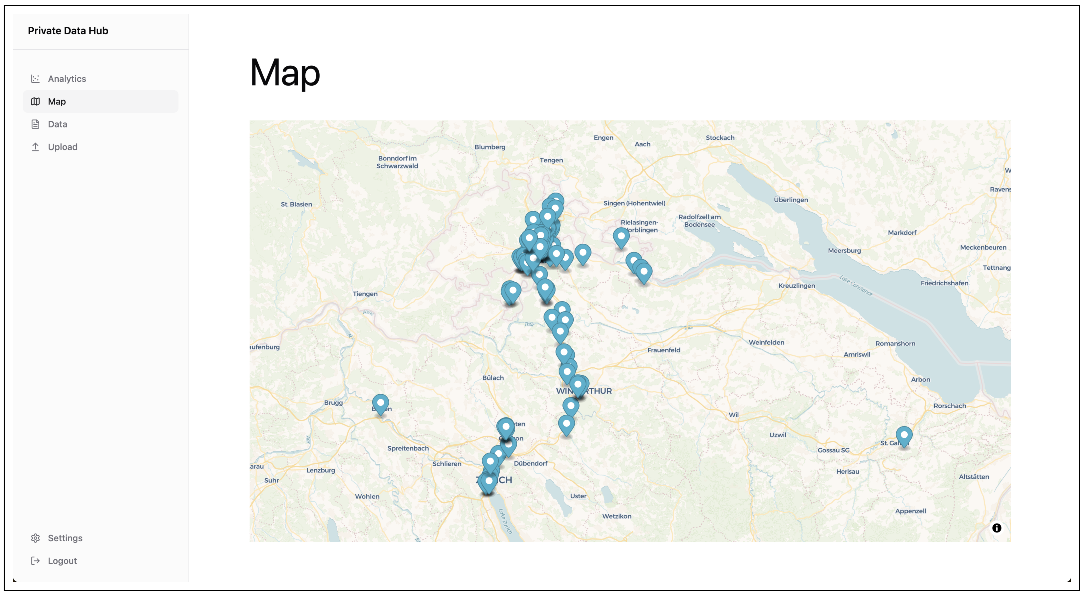
Google Spotify Netflix

Hourly Listening Trends
User activity by hour of the day

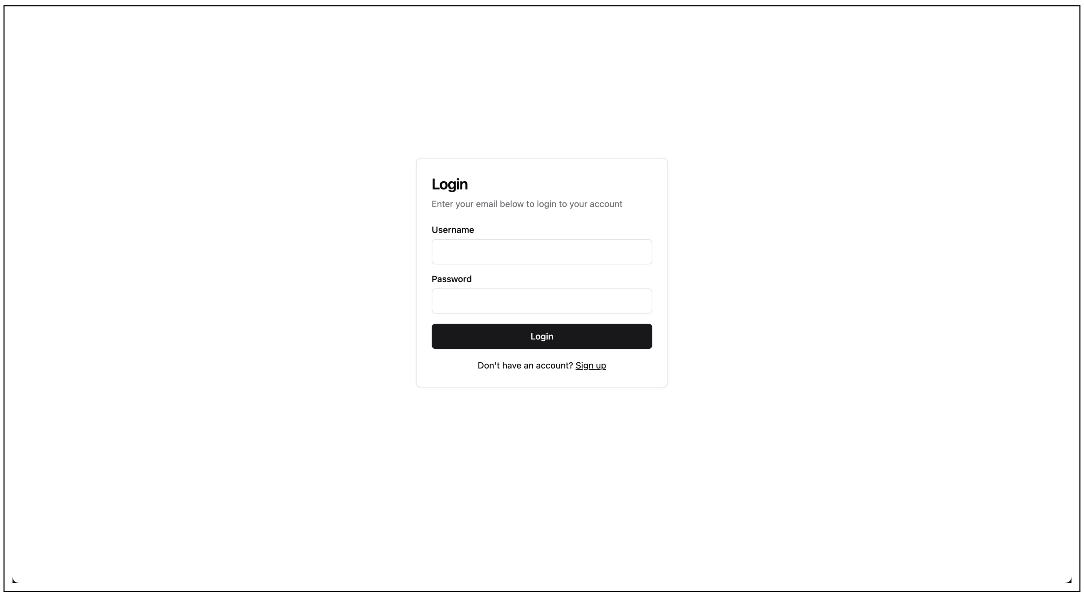
Musical Eras Distribution
Percentage distribution of music across different decades

Songs Listened to Per Month
Number of songs listened to per month

Map



Login // Signup

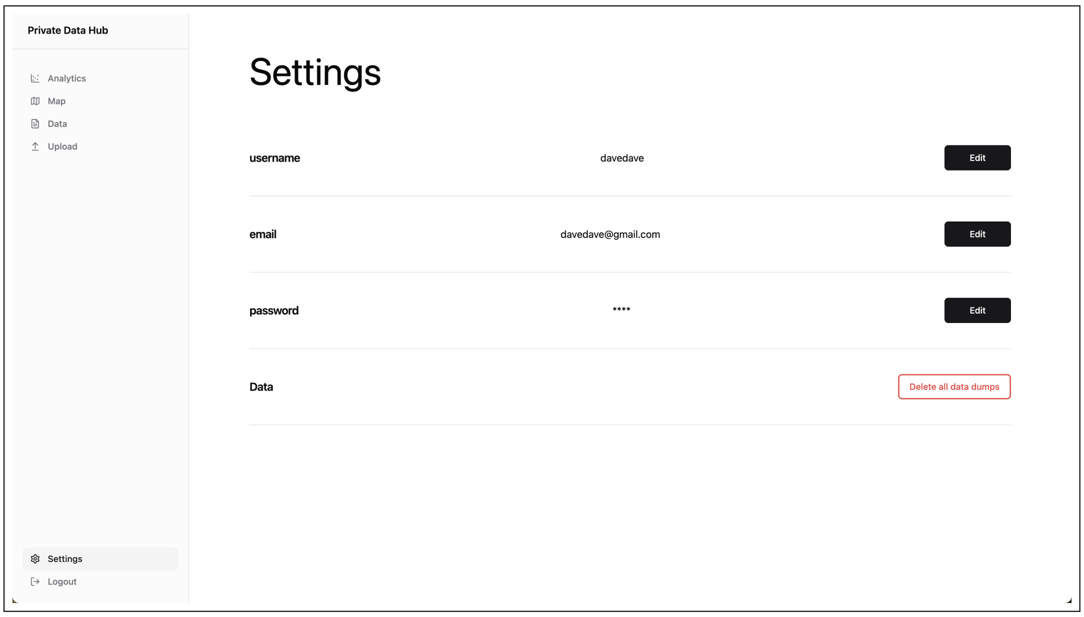

Login

Enter your email below to login to your account

Username

Password

Settings


Private Data Hub

Analytics
Map
Data
Upload

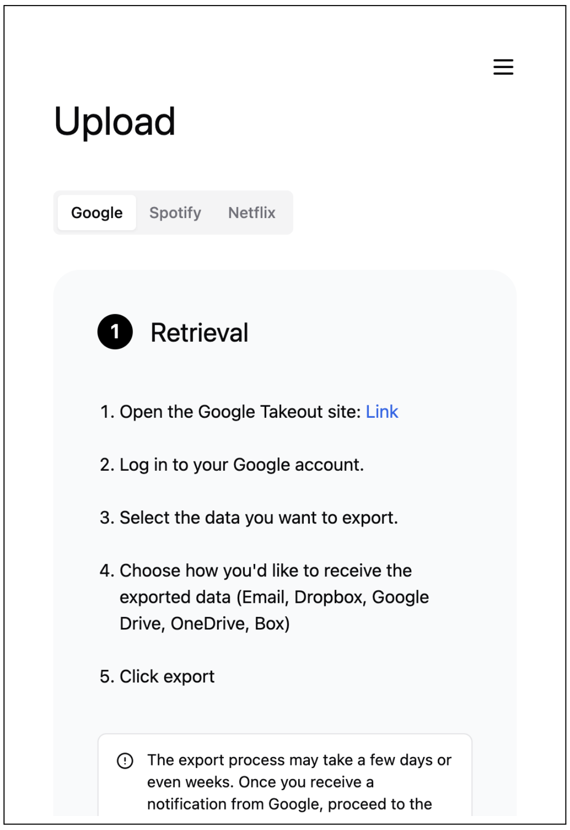
Settings

username	davedave	Edit
email	davedave@gmail.com	Edit
password	••••	Edit
Data		Delete all data dumps

Settings
Logout

5.1.2 Phone

Upload


Upload

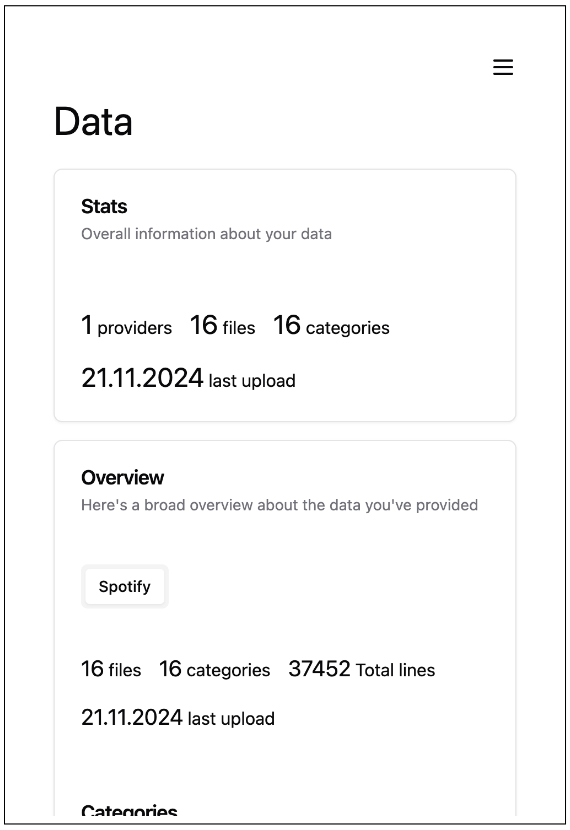
Google Spotify Netflix

1 Retrieval

1. Open the Google Takeout site: Link
2. Log in to your Google account.
3. Select the data you want to export.
4. Choose how you'd like to receive the exported data (Email, Dropbox, Google Drive, OneDrive, Box)
5. Click export

The export process may take a few days or even weeks. Once you receive a notification from Google, proceed to the

Data


Data

Stats
Overall information about your data

1 providers 16 files 16 categories

21.11.2024 last upload

Overview
Here's a broad overview about the data you've provided

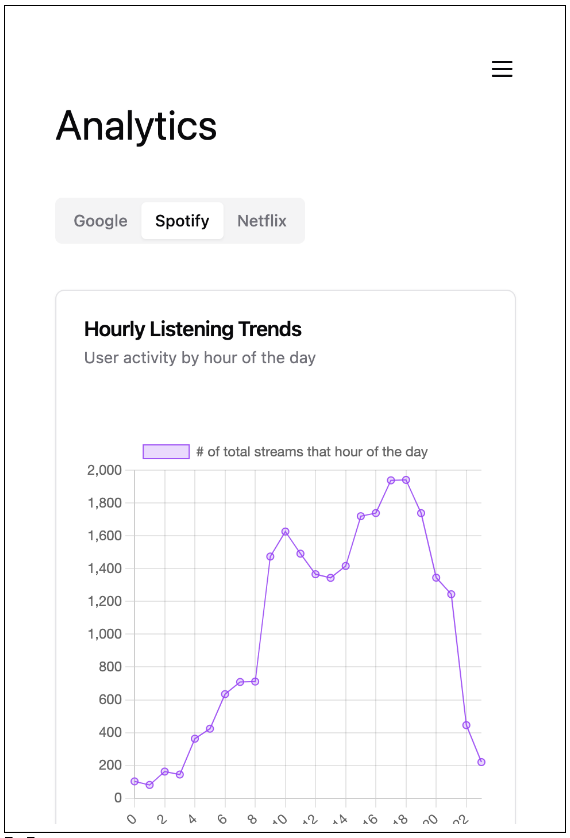
Spotify

16 files 16 categories 37452 Total lines

21.11.2024 last upload

Categories

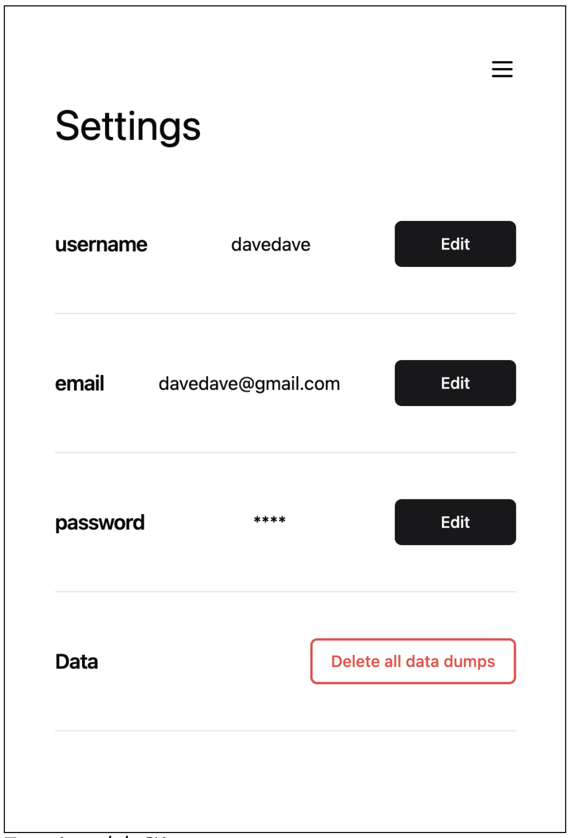
Analytics


Analytics

Hourly Listening Trends

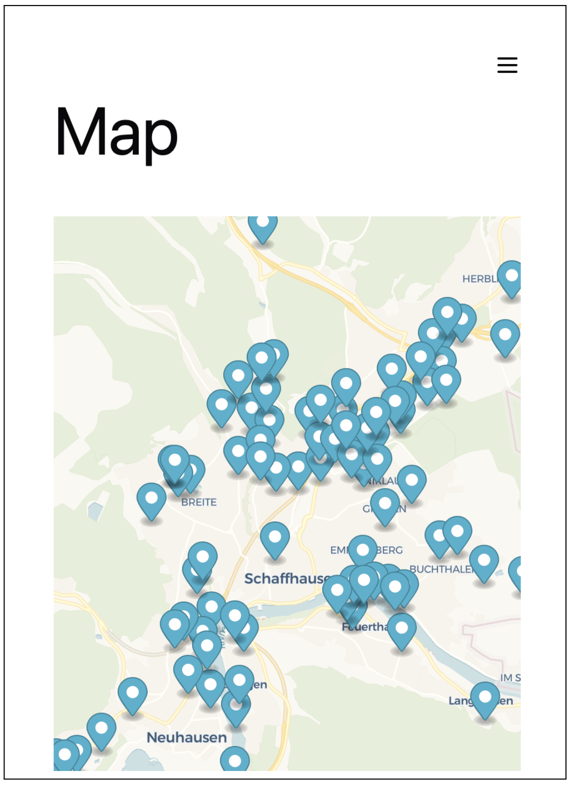
User activity by hour of the day

Settings

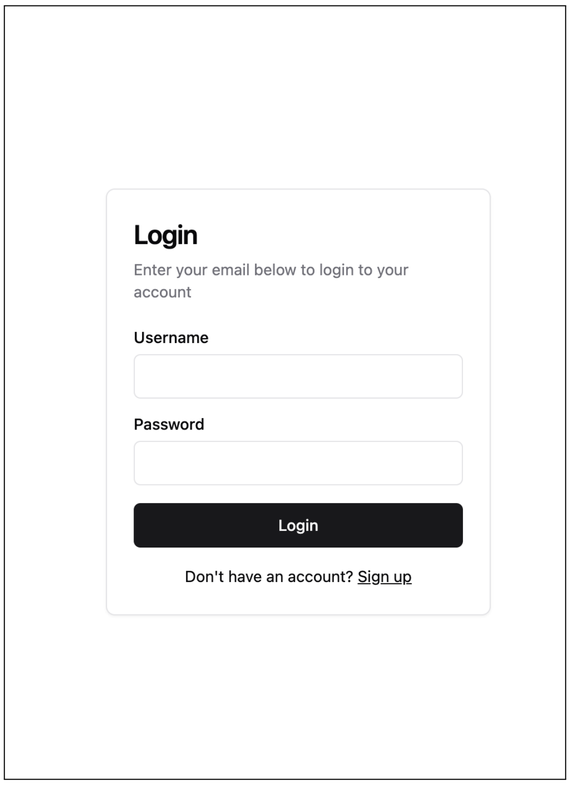

Settings

username	davedave	Edit
email	davedave@gmail.com	Edit
password	****	Edit
Data		Delete all data dumps

Map


Map

Login // Signup


Login

Enter your email below to login to your account

Username

Password

Login

Don't have an account? Sign up

5.2 Responsive Design

Our responsive design approach leverages Tailwind CSS classes that adapt layout and typography to different screen sizes. By adjusting classes based on breakpoints (e.g., `sm`, `md`, `lg`), our UI fluidly scales across devices.

Here's a simplified Svelte component to showcase this mechanism:

```
<script>
    let title = "Dashboard";
</script>

<h1 class="text-base sm:text-lg md:text-xl lg:text-2xl">
    {title}
</h1>
```

5.3 Authentication

User authentication is handled through a simple login and signup flow. When a user submits their credentials, the backend validates them and returns a signed token, which is stored in a cookie on the client.

See Section 6.2 for a deeper understanding of how token creation is managed in the backend.

5.3.1 Protected Pages

A dedicated SvelteKit `handle` hook checks for a valid token on every request. If the token is invalid or missing, the user is redirected to a login page. This ensures only authenticated users can access protected routes, while login and signup pages remain publicly accessible.[Svea]

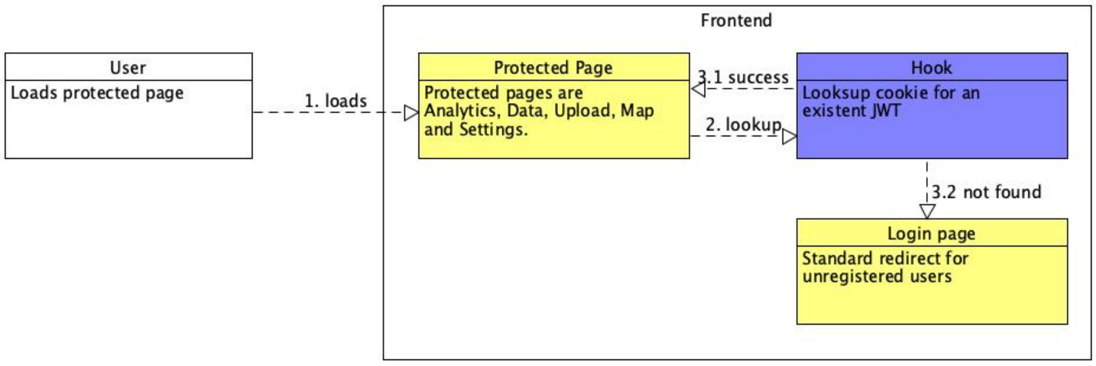

Frontend
User
Loads protected page
1. loads
Protected Page
Protected pages are Analytics, Data, Upload, Map and Settings.
3.1 success
2. lookup
Hook
Looksup cookie for an existent JWT
3.2 not found
Login page
Standard redirect for unregistered users

Figure 5.1: Diagram of how protected routes work

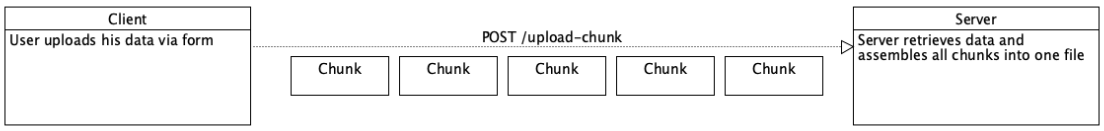
The implementation can be found in the **hooks.server.ts** file in the frontend.

5.4 Upload

Managing large file uploads requires careful handling to prevent excessive memory use on the server side. By processing uploads in smaller segments, we ensure a more stable and scalable experience.

5.4.1 Chunking

We implement a client-side chunking strategy that splits large files into manageable parts before sending them to the server. This approach minimizes server RAM usage and helps maintain consistent performance, even as file sizes grow.


Client
User uploads his data via form
POST /upload-chunk
Chunk
Chunk
Chunk
Chunk
Chunk
Server
Server retrieves data and assembles all chunks into one file

5.5 Map

Our application integrates a map component powered by `svelte-maplibre`, enabling an interactive visualization of geographical data. [Sveb]

5.5.1 Spatial Clustering

To maintain smooth performance, we apply spatial clustering to bundle nearby coordinates into aggregated markers. This ensures that the map does not become overburdened when displaying large datasets, allowing users to zoom and navigate efficiently without performance degradation.

5.6 Data Fetching

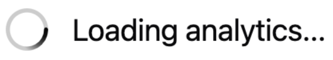
Data fetching in SvelteKit often involves using a `load` function on the server to retrieve required data before the page is rendered. This ensures data is available at load time, improving the initial user experience. [Svea]

```
export const load: PageServerLoad = ({ cookies }) => {
    const token = cookies.get("token");
    return {
        streamed: {
            dashboard: db.getUserData("data/dashboard", token)
        }
    };
};
```

5.6.1 Asynchronous Loading

To enhance user experience, our frontend uses asynchronous loading to ensure pages begin rendering even if the data isn't fully fetched yet. This approach gives users immediate feedback, avoiding blank screens during data retrieval. [Svea]

We implemented a Preloader component that provides a smooth visual cue whenever data is still being loaded.


Loading analytics...

Svelte's await syntax makes this seamless, as shown in the example snippet. By wrapping asynchronous data operations, we can show a loading message while waiting, render the data once available, or handle any errors gracefully.

```
{#await dataPromise}
    <Preloader>Loading data...</Preloader>
```

```
{:then data}
    <Dashboard data={data} />
{:catch someError}
    System error: {someError.message}.
{/await}
```

5.7 Chart.js

For our frontend graphics, we utilized Chart.js, a lightweight JavaScript library that enables the creation of responsive and visually appealing charts with ease. Chart.js was selected for its simplicity and versatility, allowing us to efficiently render various chart types—including bar, line, and pie charts—without extensive configuration. [Cha]

5.7.1 Implementation Example

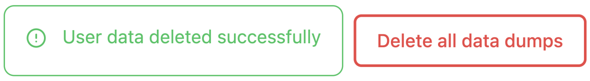
To illustrate how Chart.js enhanced our project, consider the following code snippet used to create a dynamic line chart:

```
const ctx = document.getElementById('dataChart').getContext('2d');
const dataChart = new Chart(ctx,
{
    type: 'line',
    data: { labels: ['January', 'February', 'March', 'April'],
        datasets: [{
        label: 'Purchase History over Time',
        data: [150, 200, 175, 225],
        }]
    },
});
```

This implementation allowed us to present data interactively, providing users with clear and engaging visual insights.

5.8 Data Deletion

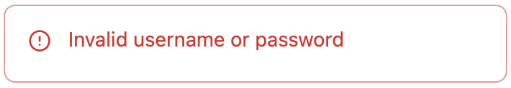
Data deletion is handled through the `Settings` page, providing users with full control over their personal data. Users can easily remove their data by navigating to the settings, initiating the deletion process via an intuitive interface.


User data deleted successfully

Delete all data dumps

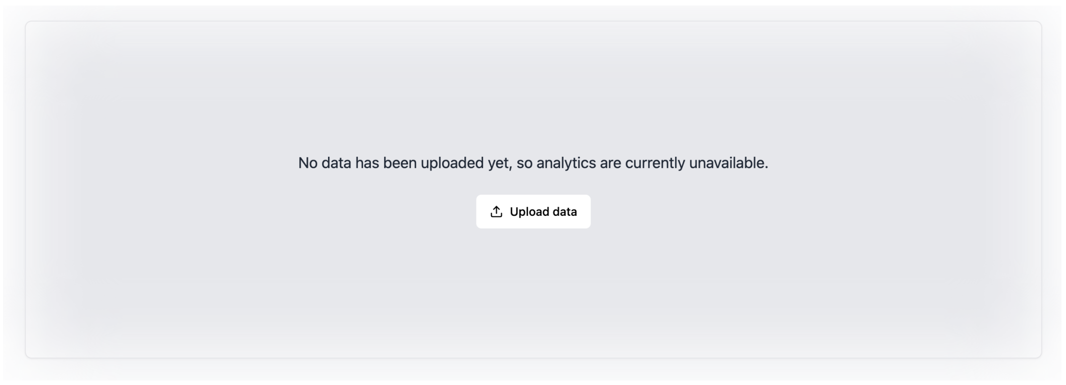
5.9 Error Handling

Our frontend employs simple yet effective error-handling techniques to maintain a smooth user experience. If a request fails or an unexpected state occurs, the UI displays a clear message.


Invalid username or password

5.10 Empty data state

In cases where no data is available, our application handles this gracefully by displaying a clear message to inform users of the absence of data.


No data has been uploaded yet, so analytics are currently unavailable.

Chapter 6

Backend Implementation

6.1 Endpoints

Below is a list of API endpoints that were implemented in the process of writing the application.

6.1.1 User

- **POST /api/user/register:** Registers a new user by providing necessary credentials.
- **POST /api/user/login:** Logs in an existing user by verifying credentials and returning a JWT.
- **GET /api/user/me:** Retrieves information about the currently authenticated user.
- **PUT /api/user/update:** Updates user details for the currently authenticated user.

6.1.2 Data

- **POST /api/data/start-upload:** An upload id is returned which is used in upload-chunk.
- **POST /api/data/upload-chunk:** Uploads chunks of a file to ensure large files are uploaded efficiently.
- **GET /api/data/info:** Retrieves user data analytics and summary information.
- **GET /api/data/dashboard:** Fetches detailed dashboard information for the user.
- **GET /api/data/coordinates:** Retrieves geographic coordinate data for visualizations.
- **DELETE /api/data/delete_user_data:** Deletes all user-uploaded data securely.

6.2 Authentication

We implemented token-based authentication using JSON Web Tokens (JWT) with FastAPI. This approach ensures secure and stateless user verification.

6.2.1 Implementation Highlights

- **JWT Handling**: Tokens are generated with essential user claims and have an expiration time. Verification middleware ensures that only requests with valid tokens can access protected routes.

- **Password Security**: User passwords are hashed using secure algorithms before being stored in the database, preventing plain-text password storage.
- **Dependency Injection**: FastAPI's dependency injection is used to retrieve and validate the current user based on the JWT.

6.2.2 Security Considerations

- **Secret Management**: JWT secret keys are stored securely as environment variables to prevent unauthorized access.
- **HTTPS Enforcement**: All authentication-related communications occur over HTTPS to protect against data interception.
- **Token Expiration**: JWTs have a limited validity period to reduce the risk of token misuse if compromised.

Refer to Section 5.3 for a detailed explanation of the frontend's role in the authentication process.

6.3 Upload

Relevant to the upload are the following two endpoints:

6.3.1 POST /api/data/start-upload

Since our application is hosted for multiple users, parallel uploads must be supported. To enable this, we implemented the start-upload endpoint. The frontend calls this endpoint before the actual upload, receiving an upload ID in response. This ID is then included with the file uploads. The backend stores the uploads with their corresponding IDs in a list (uploads_in_progress) and creates a temporary folder to store the uploaded files temporarily. The list is used to identify canceled uploads that need to be cleaned up.

6.3.2 POST /api/data/upload-chunk

With the retrieved upload ID, the frontend can begin uploading the chunks. The chunk size is set to 100 MB to minimize unnecessary overhead. The frontend then sends each chunk sequentially to the backend:

Upload Logs

```
[...]
INFO:     172.22.0.1:38156 - "POST /api/data/start-upload HTTP/1.1" 200 OK
            09:40:27 root INFO Uploading chunk 0 of 21 for Takeout.zip.
INFO:     172.22.0.1:38156 - "POST /api/data/upload-chunk HTTP/1.1" 200 OK
            09:40:28 root INFO Uploading chunk 1 of 21 for Takeout.zip.
INFO:     172.22.0.1:38156 - "POST /api/data/upload-chunk HTTP/1.1" 200 OK
            09:40:28 root INFO Uploading chunk 2 of 21 for Takeout.zip.
INFO:     172.22.0.1:38156 - "POST /api/data/upload-chunk HTTP/1.1" 200 OK
            09:40:28 root INFO Uploading chunk 3 of 21 for Takeout.zip.
INFO:     172.22.0.1:38156 - "POST /api/data/upload-chunk HTTP/1.1" 200 OK
[...]
```

Aftermath

After the upload is complete, the backend reassembles all the files and extracts the zip folder.

```
[...]
09:40:36 root INFO Writing part 18 to final file.
09:40:36 root INFO Writing part 19 to final file.
09:40:36 root INFO Writing part 20 to final file.
09:40:36 root INFO Extracting zip file /tmp/chunks_5cc58292-c2e1-4aae-
ae04-5bc3b640640f/Takeout.zip
[...]
```

6.3.3 Cleanup

The application includes a periodic cleanup process to remove expired upload sessions that have been inactive for a specified duration (default: 30 minutes). The cleanup_expired_uploads function identifies expired sessions, deletes their temporary directories, and removes them from the session tracker. The process runs periodically using AsyncIOScheduler. The logic is located in /app/utils/upload and the scheduling of the task is initialized in the main.py file at application start up.

6.4 Data Extraction

After a successful upload, the data is unpacked and can now be processed. The algorithm for extracting the data has a recursive structure. It moves through the uploaded file structure and saves every file that is in a compatible format as a document in our database. The provider of the data is recorded during the upload. So if someone uploads a data dump from google, for example, we are able to treat this data more specifically.

To achieve this we have chosen an object-oriented approach for simplification and clarity. Each file encountered by the algorithm is initially stored in a "Document" object in python. For each provider there is a "Provider-Document" which inherits from "Document". This allows us to extract provider specific data. For an unknown provider, the superclass is used.

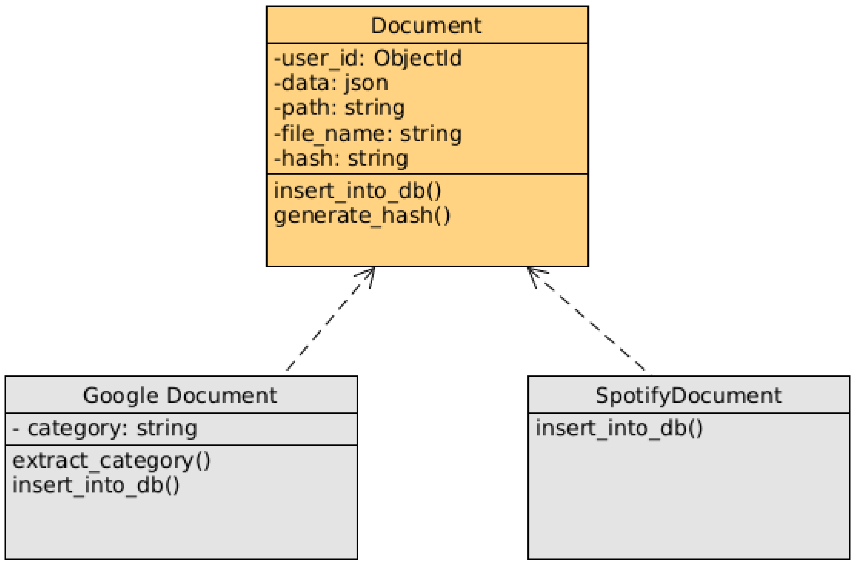


Figure 6.1: Document Objects for import algorithm

6.4.1 Objects

Until now, we have provider-specific objects for Spotify, Google, and Netflix. These objects contain member variables that store information about the data and include logic used to store the data in the database.

Hashing

To prevent the import of already stored data and to use memory efficiently, an md5 hash is created from the content of rawData and the file_name. When saving to the database, the system first checks whether the hash already exists; if it does, the file is not saved.

6.4.2 Implementation of the algorithm

The extraction algorithm is designed to recursively traverse a directory structure, identify supported files, and store their contents in a MongoDB collection. The algorithm follows these steps:

1. **Initialization**: The algorithm initializes logging and sets up the MongoDB client and GridFS for handling large files.
2. **Recursive Traversal**: The `store_files_recursively` function traverses the directory structure starting from a given root directory. For each entry:
   - If the entry is a directory, the function calls itself recursively.
   - If the entry is a supported file, it checks its size and moves on to the file handling.
3. **File Handling**:
   - **Small Files**: If the file size is within the MongoDB document size limit (16MB), it calls the corresponding file handler 6.4.3, which reads the file, converts it to json and finally creates document object (`GoogleDocument`, `SpotifyDocument`, or a generic `Document`), and calls the insert_into_db() method on it.
   - **Large Files**: If the file size exceeds the limit, it stores the file in GridFS 6.4.4.

6.4.3 File Handler

To store files in our database, we created file handlers for different formats. These handlers convert the various formats into JSON, allowing the data to be used in our company-specific objects and stored in the database. Currently, the following handlers are implemented:

- csv
- json
- html (only for specific files)
- mbox
- ics

6.4.4 Storing large files with GridFS

GridFS is a specification within MongoDB for storing and retrieving files that exceed the BSON document size limit of 16MB. Instead of saving the entire file in a single document, GridFS breaks down the file into smaller chunks and stores each chunk as a separate document. The metadata for each file, such as filename and upload date is stored in an associated fs.files collection. This was particularly necessary for files from Google, as they are sometimes quite large. More details can be found in Chapter 6.5.1.

6.4.5 Multilingual Data Handling

To accommodate language-specific data sources, such as Google data dumps, we integrated language-specific logic ensuring that our data extraction processes function seamlessly across multiple languages. This was achieved by defining language-specific keywords and implementing dynamic query building to handle variations in data paths and service names.

For example, the LANGUAGE_KEYWORDS dictionary maps different language variations for various services and paths:

```
LANGUAGE_KEYWORDS = {
    "activity": [
        "Meine Aktivitäten",
        "My Activity"
    ],
    "google_search": [
        "Google Suche",
        "Search"
    ],
    ...
}
```

The ‘check_keywords‘ function verifies if a given path contains any of the relevant keywords for a specified category:

```
def check_keywords(path, keyword_category):
    keywords = LANGUAGE_KEYWORDS[keyword_category]
    return any(keyword in path for keyword in keywords)
```

Additionally, the ‘build_user_data_query‘ function constructs appropriate queries by iterating through the possible language translations to ensure data is fetched from the correct database collection.

```
def build_user_data_query(user_id, service, path):
    for index in range(len(LANGUAGE_KEYWORDS[service])):
        translated_service = LANGUAGE_KEYWORDS[service][index]
        translated_path = LANGUAGE_KEYWORDS[path][index]

        query = {
            "userId": user_id,
            "service": translated_service,
            "path": {"$regex": translated_path}
        }

        if user_data_collection.count_documents(query):
            return query, 'collection'

        gridfs_query = {
            "user_id": user_id,
            "filename": {"$regex": translated_path}
        }

        if grid_fs.exists(gridfs_query):
            return gridfs_query, 'gridfs'

    return None, None
```

This implementation ensures that our system accurately identifies and processes user data regardless of the language, maintaining the integrity and reliability of our data extraction workflows.

6.5 Data analysis

The analysis.py file is the core of the analysis process. In this file, there is a configuration object where each algorithm is registered:

```
ALGORITHMS_CONFIG = {
    "google": {
        "google_purchase_history": google_algorithms.google_purchase_history,
        "google_calendar_analysis": google_algorithms.google_calendar_analysis
    },
    "spotify": {
        "musical_eras": spotify_algorithms.musical_eras,
        "listening_sessions": spotify_algorithms.listening_sessions,
    },
    "netflix": {
        "payments": netflix_algorithms.total_payments,
        "viewing_activity": netflix_algorithms.viewing_time_analysis,
    },
    "general": {
        "coordinates": general_algorithms.extract_coordinates,
    }
}
```

This file contains general methods to trigger specific analyses and cache the results.

6.5.1 Google

We implemented an analysis for the google data dump by developing custom algorithms that query and process user data from various Google services.

The size of each user's data dump varies significantly based on their usage of the provided services. Consequently, some files are stored in GridFS when they exceed the standard database size limits, while smaller files remain in the regular database. To ensure our algorithms function correctly, we implemented the "fetch_user_data" function that dynamically directs queries to the appropriate database collection. This approach guarantees efficient and accurate data processing tailored to each user's specific data volume.

```
def fetch_user_data(query, storage_type, specific=None):
    if not query:
        return

    if storage_type == 'collection':
        for doc in user_data_collection.find(query):
            raw_data = doc.get('rawData', [])
            if specific:
                for entry in raw_data[specific]:
                    yield entry
            else:
                for entry in raw_data:
                    yield entry

    elif storage_type == 'gridfs':
        for grid_file in grid_fs.find(query):
            content = BytesIO(grid_file.read())
            data = json.loads(content.read().decode('utf-8'))

            if specific:
                for entry in data[specific]:
                    yield entry
            else:
                for entry in data:
                    yield entry

    else:
        return []
```

Implemented Analyses

- Most visited domains
- YouTube watchtime
- Email usage analysis
- Google search analysis
- Google rating analysis
- Google purchase history
- Google calendar analysis

6.5.2 Spotify

Our Spotify analysis algorithms leverage the Spotify API, enabling us to combine data from user data dumps with additional information retrieved directly from the API. By utilizing these two data sources, we developed customized algorithms to process the data and generate meaningful metrics.

Implemented Analyses

- Most streamed songs
- Most streamed artists
- Top genres
- Listening time analysis
- Most streamed podcasts
- Time spent
- Musical eras
- Listening sessions

6.5.3 Netflix

For the analysis of Netflix data, we primarily used aggregation pipelines in MongoDB. While these pipelines can be challenging to understand at first glance, they offer significant performance advantages. The evaluations were mainly based on viewing activity.

Implemented Analyses

- Total streaming hours
- Average streaming hours
- Weekdays
- Most watched
- Most watched movies
- Payments
- Viewing activity

6.5.4 General

These algorithms perform analysis on all datasets. So far, we have implemented an algorithm for detecting coordinates, which searches all datasets using multiple regular expressions for coordinates. For Google, we have a more specific algorithm that additionally extracts coordinates from the location history.

Implemented Analyses

- Coordinates extraction

6.5.5 Other

These algorithms are responsible for the information on the dashboard.

Implemented Analyses

- List Categories
  - Retrieves all unique categories of data files associated with a specific user and data provider.
- Last Upload Date
  - Determines the most recent upload date for a user, optionally filtering by a specific provider.
- Count Lines
  - Counts the total number of lines of data stored for a specific user and provider.

6.5.6 Analysis implementation

The actual analysis algorithms are located in the algorithms module, where there is a separate file for each provider. The analysis is based on queries of the data in MongoDB. Aggregation pipelines were used for some of the analyses, which make the queries more efficient by processing the data in stages and reducing the amount of data that needs to be transferred.
The analysis can follow different approaches, but the output must adhere to the same structure for the frontend to recognize and display the data correctly. The structure must follow this scheme:

```
return {
        "title": "Hourly YouTube Viewing Trends",
        "description": "User viewing activity by hour of the day",
        "type": "line",
        "metrics": analytics,
    }
```

Types that can be used are: line, doughnut, bar and customList.

Data format

The ‘analytics‘ variable can store data in different formats, depending on whether the data is grouped or not. These approaches ensure that the frontend can correctly process and display the data.

- **If the data is not grouped:** The data is included directly in the ‘analytics‘ variable as a list of key-value pairs. Each dictionary in the ‘data‘ list represents an individual data point, containing relevant key-value pairs (e.g., URL and count).

  Example:

  ```
  analytics = [{
      "labelName": "# of visits",
      "data": [{"url": url, "count": count} for url, count in most_visited_urls]
  }]
  ```

  In this case, each dictionary within the ‘data‘ list represents a unique data point, with keys like ”url” and ”count” mapping to their respective values.

- **If the data is grouped (e.g., by profiles):** When the data is grouped (e.g., by profiles), the ‘analytics‘ variable contains multiple datasets. Each dataset is represented as a dictionary with the relevant data for each group (e.g., profile).

  Example:

  ```
  analytics = [{
      "labelName": "User Profile Data",
      "data": {
          "profile_1": round(profile["total_duration"] / 3600, 2),
          "profile_2": round(profile["total_duration"] / 3600, 2)
      }
  }]
  ```

  In this case, each key in the ‘data‘ dictionary represents a group (e.g., profile), and the corresponding value represents the metric (e.g., total duration in hours) for that group.

- **Using `customList` when the data is grouped but has differing data names:** When the data is grouped by profiles or categories and contains differing data points (e.g. most watched movies per profile), the ‘customList‘ type is used to structure the response.

  Example:

  ```
  data = [{"profile": profile["_id"], "data": [
      [rank, content["title"], round(content["total_duration"]/60, 1)]
      for rank, content in enumerate(profile["top_content"], start=1)
  ]} for profile in result]

  return {
      "title": "Top 5 Most Watched",
      "description": "Top 5 most watched content across profiles. Ranked by hours watched
      "type": "customList",
      "labels": ["Rank", "Title", "Hours"],
      "metrics": data
  }
  ```

  In this case, each profile’s data is grouped and displayed with custom labels (e.g., ”Rank”, ”Title”, ”Hours”). The ‘customList‘ type is particularly useful when displaying ranked content or varied data for each group.

6.5.7 Caching

To make the analysis more efficient, we store the results of the analysis in the database. When a user requests an analysis, the results are fetched from the database. The results are only updated when new data is uploaded.

6.6 Deployment

Note: All deployment-specific changes are only present in the production branch.

We deployed our application on a DigitalOcean Droplet using Docker Compose. The application consists of a frontend (SvelteKit), backend (FastAPI), and a MongoDB database, all running in separate containers. Caddy acts as the reverse proxy, providing secure HTTPS connections by automatically handling SSL certificates. Docker Compose simplifies the management of these services, ensuring easy deployment, scaling, and consistent environments across development and production.

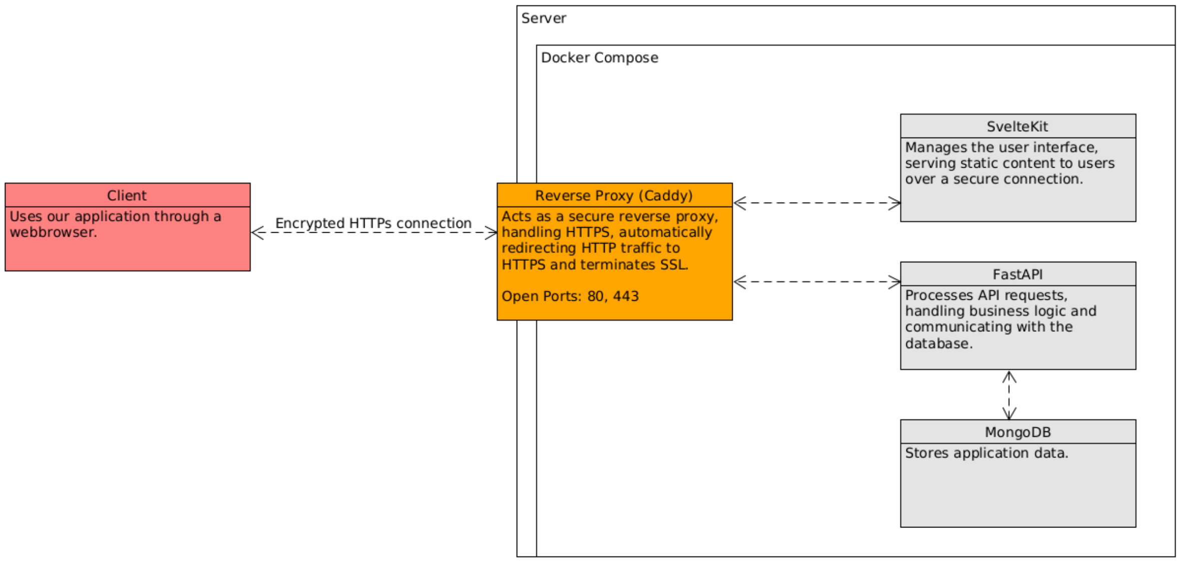

Server
Docker Compose
Client
Uses our application through a webbrowser.
Encrypted HTTPs connection
Reverse Proxy (Caddy)
Acts as a secure reverse proxy, handling HTTPS, automatically redirecting HTTP traffic to HTTPS and terminates SSL.
Open Ports: 80, 443
SvelteKit
Manages the user interface, serving static content to users over a secure connection.
FastAPI
Processes API requests, handling business logic and communicating with the database.
MongoDB
Stores application data.

Figure 6.2: Docker compose overview

6.6.1 Docker Compose

Docker Compose is a tool for managing multi-container applications, enabling efficient development and deployment. With a simple YAML file you can configure and monitor all services in your application stack with one command.[Doc]

6.6.2 Digital Ocean Droplet

In the droplet on digital ocean we have cloned the git repository and adjusted the environment variables. With the command "docker compose up –build" we build and start our configuration.

Caddy Configuration Overview

```
private-data-hub.adaptit.ch {
    reverse_proxy /api/data/upload-chunk fastapi:8000 {
        header_down Strict-Transport-Security max-age=31536000;
    }
    reverse_proxy /api/data/start-upload fastapi:8000 {
        header_down Strict-Transport-Security max-age=31536000;
    }
    reverse_proxy frontend:3000 {
        header_down Strict-Transport-Security max-age=31536000;
    }
}
```

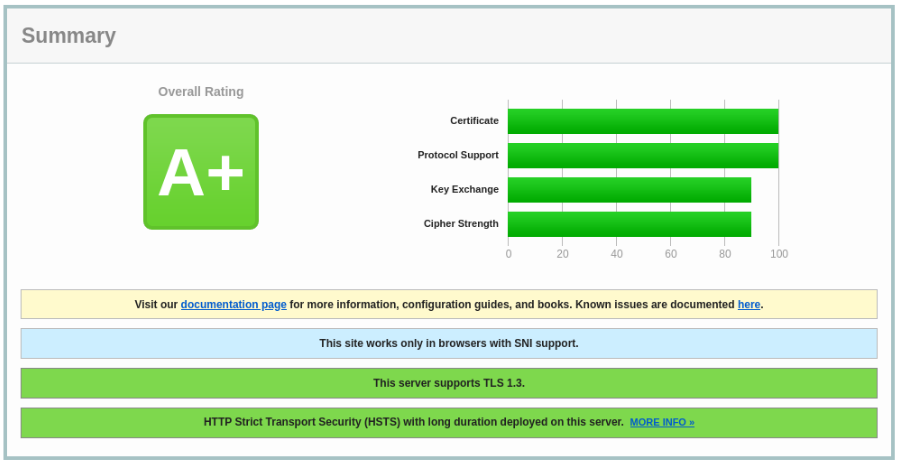
This Caddy configuration sets up `private-data-hub.adaptit.ch` as a secure reverse proxy, directing requests based on their paths:

- **API Traffic**: Requests to `/api/data/upload-chunk` or `/api/data/start-upload` are routed to the `fastapi` service on port `8000`.
- **Frontend Traffic**: All other requests are directed to the `frontend` service on port `3000`.
- **Security Enforcement**: The `Strict-Transport-Security` header enforces HTTPS connections with a one-year duration (`max-age=31536000`), ensuring all traffic remains encrypted.

6.7 Security

6.7.1 SSL

Since the information processed on our application is very sensitive, encrypted communication between client and server is essential. To secure sensitive information, Caddy serves as a HTTPs-enforcing reverse proxy, automatically managing SSL certificates to ensure encrypted communication. This configuration redirects all HTTP traffic to HTTPS, which protects the user data. SSL is crucial in protecting data during transit by establishing a secure channel between client and server. Through these protocols, Caddy simplifies encryption setups with automation and secure defaults, helping us ensure that all communications meet security standards.[Cad][SSLb]



Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This site works only in browsers with SNI support.

This server supports TLS 1.3.

HTTP Strict Transport Security (HSTS) with long duration deployed on this server. MORE INFO »

Figure 6.3: SSL Report with ssllabs.com

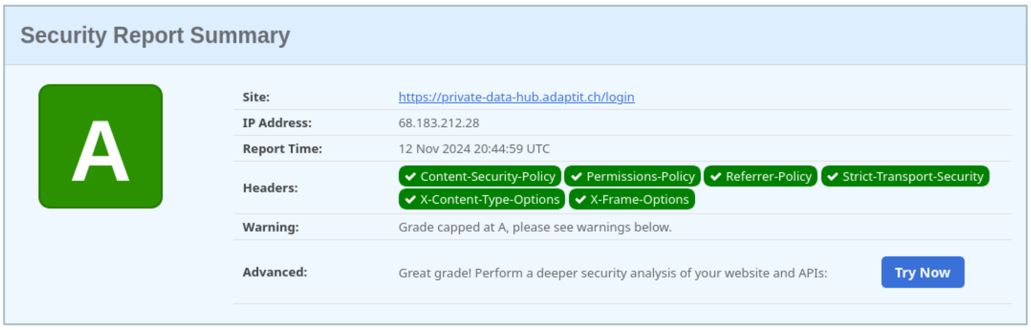
We used ssllabs.com to automatically test the https connection to our application. The service performs an analysis of the configuration of any SSL web server on the public Internet.[SSLa]

6.7.2 Ports

In the Compose file, the ”ports” keyword is used to expose specific ports for connections. To secure the server and API from unauthorized access, only ports 80 and 443 are open for HTTP and HTTPS traffic. All requests are routed through Caddy, which acts as a reverse proxy for these ports. There is no other way to communicate with the frontend or backend, ensuring that all traffic is securely handled by Caddy within the Docker network.

6.7.3 CORS and Security Headers

To further enhance the security of our application, we implemented strict CORS (Cross-Origin Resource Sharing) policies and essential security headers. These measures ensure that only authorized origins can access our resources while protecting against a range of common web security vulnerabilities.



Security Report Summary

A

Site:	https://private-data-hub.adaptit.ch/login
IP Address:	68.183.212.28
Report Time:	12 Nov 2024 20:44:59 UTC
Headers:	✓ Content-Security-Policy ✓ Permissions-Policy ✓ Referrer-Policy ✓ Strict-Transport-Security ✓ X-Content-Type-Options ✓ X-Frame-Options
Warning:	Grade capped at A, please see warnings below.
Advanced:	Great grade! Perform a deeper security analysis of your website and APIs: Try Now

Figure 6.4: Security header scan with securityheaders.com

6.7.4 CORS Configuration

CORS policies specify which domains can interact with our server, protecting the application from unauthorized cross-origin requests. In our configuration, we restrict cross-origin requests to approved origins, allowing only trusted sources to access the resources on our server. This limitation helps prevent potentially harmful cross-origin requests, safeguarding sensitive data and application functionality.

In Caddy and SvelteKit, we manage CORS headers to enforce this policy at the server level, setting Access-Control-Allow-Origin to only permit requests from our client's origin. By restricting this header to specific, trusted origins, we prevent malicious sites from making requests to our application, reducing the risk of CSRF (Cross-Site Request Forgery) and other similar attacks.

6.7.5 Security Headers

In addition to CORS, several HTTP security headers are configured to further protect the application:

- Content-Security-Policy (CSP)
- X-Frame-Options
- Strict-Transport-Security (HSTS)
- Referrer-Policy
- Permissions-Policy

6.8 Logging

Logging in the backend is implemented using Python's logging module. Logs are written to a file (backend_logs.txt) and are also displayed in the console. The logs include timestamps, log levels, and messages. The logging setup is initialized at the application startup, ensuring consistent logging across all components in the backend.

Logging was implemented by manually inserting log entries at key points in the application's algorithms and actions. This makes it easier to identify and troubleshoot issues when something goes wrong. Each log entry provides information about the application's state at the time of execution.

6.9 Local Development

Information about how to develop the project locally can be found in the README files in our repository. There is a README in both the frontend and backend directories that describe the steps that need to be performed.

Chapter 7

Testing Concept

We have developed a test concept to ensure the functionality of our application. This concept includes automated testing of individual components or functions, application testing through acceptance tests, and functionality testing from an end user's perspective (usability tests).

7.1 Pipelines

In our repository, we use gitlab pipelines to run automated tests.

7.2 Automated Tests

E1: Upload: Multiple files

The import function is tested to evaluate its behavior when multiple files are uploaded. The test ensures that no errors occur during successive file uploads and that all files are correctly saved in the database.

E2: Upload: Duplicates

In this test, two identical files are processed. The test is considered successful if the system recognizes during the import that the second file is identical to the first and, as a result, does not import it.

E3: Upload: User assignment

This test ensures that the correct user is assigned to each imported file and that no mismatches occur.

E4: Upload: Paths and file names

This test verifies that the path and filename are correctly assigned and stored in the database.

E5: Upload: Company assignment

This test verifies that the company field of imported files is correctly assigned to the specified company.

E6: Endpoints: Info

This test verifies the response returned by the endpoint /api/data/info when no data is uploaded. The response must be an empty array.

E7: Endpoints: Dashboard

This test verifies the response returned by the endpoint /api/data/dashborad when no data is uploaded. The response must be an empty array.

E8: Endpoints: Coordinates

This test verifies the response returned by the endpoint /api/data/coordinates when no data is uploaded. The response must be an empty array.

E9: Endpoints: Delete user data

This test verifies the response returned by the endpoint /api/data/delete_user_data when no data is uploaded. The response must return status code 200 and the message "User data deleted successfully".

7.3 Acceptance Tests

An acceptance test is conducted after the implementation of each new feature. At the start of every acceptance test, the following steps are performed: build and run the application using Docker Compose, and sign in or register with a new account and/or delete any existing data.

7.3.1 Analysis

In general, when implementing a new analysis or modifying an existing one, the structure of the response is verified. The developer utilizes the `/docs` page of FastAPI (backend) for this purpose. This allows specific endpoints to be called, and the corresponding responses to be examined. If the structure matches the defined guidelines [6.5.6], the next steps in the process can be executed.

Spotify analysis

This test involves uploading Spotify data through our upload interface. After the upload, we perform the following checks:

1. Verify that the dashboard displays the correct information.
2. Confirm that the analysis page shows the information accurately and that everything is displayed correctly.

Netflix analysis

This test involves uploading Netflix data through our upload interface. After the upload, we perform the following checks:

1. Verify that the dashboard displays the correct information.
2. Confirm that the analysis page shows the information accurately and that everything is displayed correctly.

Google analysis

This test involves uploading Google data through our upload interface. After the upload, we perform the following checks:

1. Verify that the dashboard displays the correct information.
2. Confirm that the analysis page shows the information accurately and that everything is displayed correctly.
3. Confirm that the map the shows expected locations with the filenames attached.

7.3.2 Upload

After any changes affecting the upload functionality, a manual verification is performed to ensure that the upload process operates correctly and efficiently. This includes measuring the time required to upload data. For testing, Google data dumps larger than 2GB are used. During the upload, the following metrics are recorded:

- **Duration of the upload**
- **Internet speed** at the time of the upload
- **Size of the zip file**
- **Upload success status** (whether the upload was successful)

The recorded values are documented and compared with previous results.

7.3.3 Sign up and Login

After changes to the authentication system, this test is conducted to verify that the account creation functions correctly and that requests are authorized as expected. To perform this check, the developer creates a new account via the frontend. Upon successful registration, the developer logs into the application to confirm proper functionality.

7.3.4 Other

Deletion of user data

After completing the previous acceptance tests, all user data is deleted through the settings and its deletion is verified in the database.

7.4 Usability Tests

Based on a questionnaire, we asked people in different age groups about the use of our application. Below you will find an evaluation of the questions in the test. The original questionnaire can be found here 9.4.

1. **How would you rate the overall layout of the platform (e.g., organization, spacing)?**

   - 9
   - 10
   - 7
   - 8
   - 10

   We note that the experience could be rated on a scale of 1 to 10.

2. **How well does the platform adapt to your screen size?**

   - 10
   - 10
   - 10
   - 9
   - 9

   We note that the experience could be rated on a scale of 1 to 10. All tests were conducted either on a laptop or a tablet; unfortunately, we were unable to obtain feedback regarding mobile devices.

3. **How visually appealing is the color scheme of the platform in general?**

   - 10
   - 8
   - 7
   - 6
   - 10

   We note that the experience could be rated on a scale of 1 to 10.

4. **How is the content presented on the analytics page (e.g., text, labels, data visualizations)?**

   - 10
   - 10
   - 8
   - 8
   - 9

   We note that the experience could be rated on a scale of 1 to 10.

Chapter 8

Results and findings

8.1 Problems we encountered

8.1.1 Upload

We encountered an issue when uploading content through server-side functionality consumed excessive RAM, causing the application to crash. This problem was due to uploading all data simultaneously. To address this, we transitioned the upload process to the client side and implemented data chunking, which effectively resolved the issue.

8.1.2 Webservice

Developing a web service with robust authentication, data transfer, and security features proved to be quite challenging. A significant portion of our efforts was dedicated to building the web service, which limited our ability to focus on data analysis. This proved to be exhausting; however, we eventually managed to allocate substantial time to data analysis, which we found enjoyable and rewarding.

8.1.3 Many languages

Working with Python and JavaScript (TypeScript) proved to be quite challenging due to our limited proficiency in both languages. We had to dedicate substantial time to learning language-specific features and best practices, especially with JavaScript. This learning curve added significant complexity and exhaustion to the development process, hindering our overall productivity.

8.1.4 Caching

Initially, the endpoints responsible for fetching analysis results processed the datasets every time they were called. This approach was highly inefficient and resulted in significant loading times when accessing analytics. To address this issue, we implemented a caching mechanism to store the analysis results. Since the results remain unchanged unless new data is added, this optimization greatly improved performance and reduced loading times 6.5.7.

8.2 Open Issues

8.2.1 Upload Speed

The data from Spotify and Netflix can be uploaded without any problems due to their small size, but Google dumps are considerably larger (sometimes over 10GB). Depending on the internet connection, the upload can take a very long time. We measured 30 minutes for an 8GB data dump at a speed of around 40 Mb/s.

To reduce such long waiting times, other approaches could be chosen for the upload. Only a small part of the uploaded Google data is analyzed. For example, only the metadata of photos is required and not the entire image. Such files could be removed before the upload on the client side. As we unfortunately do not have enough time to implement such a mechanism, this point remains open and could be implemented in the future.

8.2.2 Analysis

With each upload, all data dumps currently in the database are analyzed again. To make the process more efficient, information about previous analyses could be stored. This would allow us to know what data has already been analyzed and then only analyse the new data.

Chapter 9

Conclusions and outlook

9.1 Short Term

The project provided us with an opportunity to expand our knowledge and skills. By separating the front-end and back-end, we were able to learn about both sides and gain a better understanding of how such split applications interact. There were areas where we had limited prior knowledge, which motivated us to acquire and apply new concepts.

9.2 Long Term

9.2.1 Local Application

The nature of user data is extremely sensitive. It includes everything a company stores about a person, such as health information, addresses, and much more. For this reason, we believe there is little interest in sharing such data with a third-party provider. Instead, we see potential in a local application that does not require an internet connection. This approach would not only eliminate long upload times but also allow users to maintain full control over their data.

9.2.2 Open Source

Additionally, we believe it is absolutely essential to make the source code of this application publicly available. This would foster trust and increase user interest.

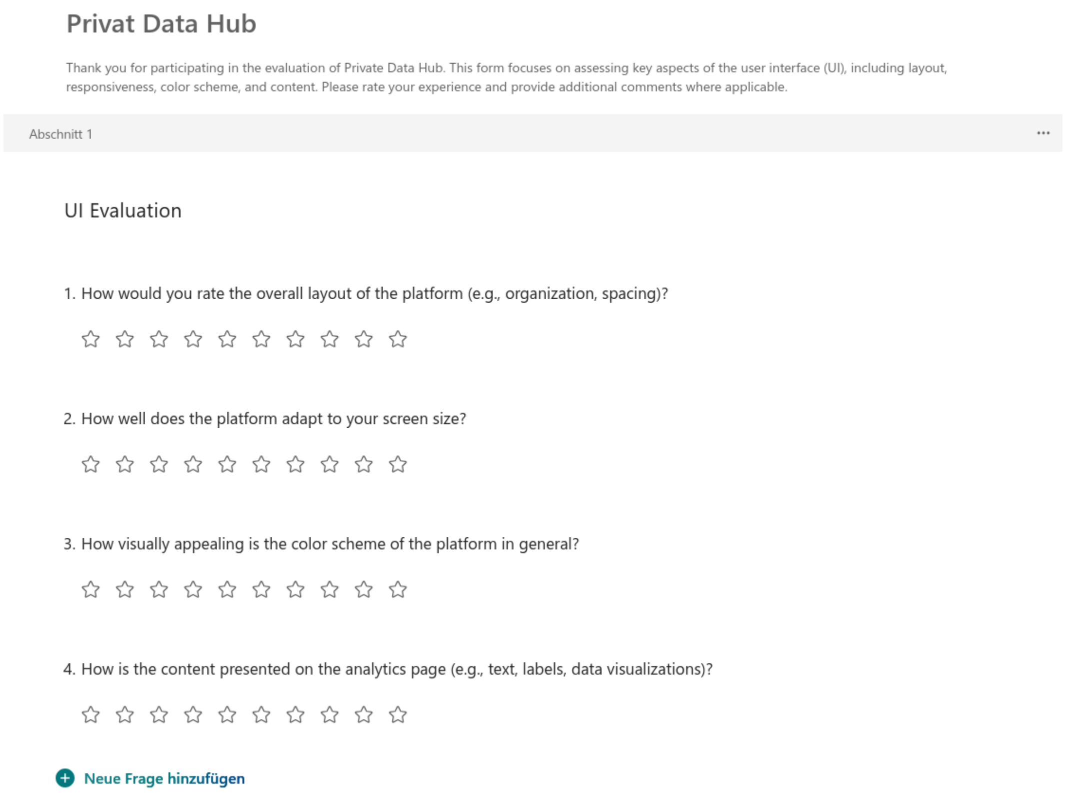
9.3 Conclusion

Overall, we recognize the value of this application and the importance of informing users about the data collected from them. The development of such an app has been challenging, particularly because user data is often poorly structured, varies across providers, and can change over time. To address these challenges, we implemented an import algorithm that loads all data into the database, with the analysis performed based on patterns and file names. This approach worked well for us, but we believe there is still room for improvement.
If we were to do it again, we would change the division between the front-end and back-end. The decision to use a separate technology for the front-end made the development process more difficult. We believe that if the application had been developed using a different technology stack, we could have focused more on the core aspects, namely the analysis and presentation of the data.

Part III

Appendix

9.4 Usability Tests Questions



Privat Data Hub

Thank you for participating in the evaluation of Private Data Hub. This form focuses on assessing key aspects of the user interface (UI), including layout, responsiveness, color scheme, and content. Please rate your experience and provide additional comments where applicable.

Abschnitt 1

UI Evaluation

1. How would you rate the overall layout of the platform (e.g., organization, spacing)?

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

2. How well does the platform adapt to your screen size?

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

3. How visually appealing is the color scheme of the platform in general?

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

4. How is the content presented on the analytics page (e.g., text, labels, data visualizations)?

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

Neue Frage hinzufügen

Bibliography

[c4m] c4model introduction. https://c4model.com/introduction. Accessed: 2024-10-29.

[Cad] Caddy: Reverse-proxy in einfach? https://blog.ordix.de/caddy-reverse-proxy-in-einfach. Accessed: 2024-11-11.

[Cha] Chartsjs. https://www.chartjs.org//. Accessed: 2024-11-11.

[Doc] Docker compose overview. https://docs.docker.com/compose/. Accessed: 2024-11-11.

[fas] fastapi. https://fastapi.tiangolo.com/. Accessed: 2024-10-29.

[Mon] How do document databases work? https://www.mongodb.com/resources/basics/databases/document-databases. Accessed: 2024-10-29.

[pyt] Python executive summary. https://www.python.org/doc/essays/blurb/. Accessed: 2024-10-29.

[SSLa] Ssl server test. https://www.ssllabs.com/ssltest/. Accessed: 2024-11-11.

[SSLb] Was ist ssl. https://www.cloudflare.com/de-de/learning/ssl/what-is-ssl/. Accessed: 2024-11-11.

[Svea] Svelte documentation. https://svelte.dev/docs. Accessed: 2024-11-11.

[Sveb] Svelte maplibre. https://svelte-maplibre.vercel.app/. Accessed: 2024-11-11.

[tec] What is mongodb? https://www.techtarget.com/searchdatamanagement/definition/MongoDB. Accessed: 2024-10-29.