UNIVERSITY of BRADFORD

Faculty of Engineering and Informatics
Department of Computer Science

Designing a visualization tool for adult social care data

Final project report

Iheb Bouzaiane

22040992

A thesis submitted in part fulfilment of the degree of BEng (Hons) in Software Engineering

16 June 2023

Declaration

The candidate confirms that the work submitted is his/her own and that appropriate credit has been given where reference has been made to the work of others. The candidate agrees that this report can be electronically checked for plagiarism.

Iheb Bouzaiane

Abstract

The amount of data collected has been increasing in every field of study. This is especially true for the health and social care sectors where patients' records have been stored and updated since their initial visit.

However, gaining useful insights from large and complex datasets that deal with real world data requires the combined efforts of multiple disciplines ranging from information visualization, data mining, machine learning and human-computer interaction (HCI) as well domain experts' knowledge. Despite advancements in other fields such as healthcare, finance… that led to innovative tools for tracing disease causes or market profits, social care still struggles to understand the population's journey through the care system. Information visualization and analytics can empower decision makers to gain insights into the effectiveness and performance of the social care sector. This will help them shape better policies for the population's needs.

In this report, the process of creating a social care data visualization application was described in three main steps. Initially, the most relevant features across the different social care datasets were extracted. This was based on the meetings with the analysts working on the Connected Yorkshire research database. This step provided the groundwork for the subsequent development process. The second step involved the design and construction of a Looker dashboard offering a holistic view of the dataset. This dashboard served as a powerful tool for developing an initial understanding of the dataset. Finally, a web application was developed, leveraging the power of Flask for the backend and both React and Redux for the frontend. The application enabled users to smoothly generate exploratory dashboards. These dashboards would be used for in-depth analysis and hypothesis generation about the social care dataset. The ultimate goal of the application was to facilitate the generation of quick insights about the social care dataset and assisting stakeholders in making informed decision.

Table of Contents

ABSTRACT ..... **I**

1. INTRODUCTION **1**

1.1 PROJECT DESCRIPTION 1
1.2 NOVELTY AND CONTRIBUTION 1
1.3 REPORT OVERVIEW 2

2. LITERATURE REVIEW **3**

2.1 VISUAL ANALYTICS 3
2.1.1 Definition *3*
2.1.2 Applications *3*
2.1.3 Workflow *3*
2.1.4 Challenges and solutions *4*
2.1.5 Tools used in visual analytics *6*
2.2 BACKEND DEVELOPMENT 7
2.3 FRONTEND DEVELOPMENT 8

3. REQUIREMENTS AND ANALYSIS **9**

3.1 REQUIREMENTS 9
3.2 ANALYSIS 10

4. DESIGN, IMPLEMENTATION, AND TESTING **13**

4.1 DATA ANALYSIS WORKFLOW 13
4.2 DESIGN 18
4.2.1 Architecture *18*
4.2.2 Backend *21*
4.2.3 Frontend *28*
4.2.4 User interfaces *30*
4.3 IMPLEMENTATION 34
4.4 TESTING 62
4.4.1 Integration testing *62*
4.4.2 System testing *64*
4.4.3 Unit testing *65*

5. RESULTS AND DISCUSSIONS **67**

5.1 MAIN ACHIEVEMENTS 67
5.2 LIMITATIONS OF THE SOLUTION 68
5.3 PROFESSIONAL, LEGAL, SOCIAL, ETHICAL AND ECONOMIC CONSIDERATIONS 69

6. CONCLUSION AND FURTHER WORK **70**

7. REFERENCES **71**

8. APPENDICES **74**

List of tables

Table 1. Visual analytics pipeline stages description....4
Table 2. Strategies used to deal with the data volume....5
Table 3. Description of the steps in the nine-stage methodology framework....6
Table 4 Comparison between Looker studio and Tableau....7
Table 5 Comparing SQLite, MySQL, and PostgreSQL....8
Table 6 User objectives....9
Table 7 User requirements....9
Table 8. List of functional requirements....10
Table 9. List of non-functional requirements....10
Table 10 Model classes description....23
Table 11 Controllers associated with the Models....26
Table 12 Controllers associated with the Dataset....26
Table 13 Mapping of the endpoints to the controllers....28
Table 14 Client directory structure....30

List of figures

Figure 1. Visual analytics pipeline proposed by Keim et al 3
Figure 2. Nine-stage design methodology framework (Sedlmair, Meyer and Munzner, 2012) 5
Figure 3 Use case diagram of the system 11
Figure 4 ER diagram for the User, Post, Visualization entities 12
Figure 5 Extracting the demographic characteristics 13
Figure 6 Joining contact information to the dataset 14
Figure 7 Joining GP visit information for each individual 15
Figure 8 Adding a new column to indicate the GP visit occurrence 15
Figure 9 Updating the gender column in the dataset 15
Figure 10 Adding the number of months between the contact and the visit occurrence 16
Figure 11 Spatial distribution of the individuals according to the reason of the contact 16
Figure 12. Visualizations for the ethnicity and gender distribution within the system 17
Figure 13 Reason of contact and the outcome of contact in the cohort 17
Figure 14 Common filters applied for the looker visualizations 18
Figure 15 representing the year of death in the cohort 18
Figure 16 Overview of the system architecture 19
Figure 17 Dashboard embedded in the explore dataset page 20
Figure 18 Server directory 21
Figure 19 Class diagram for the Backend Models 22
Figure 20 User schema 27
Figure 21 Post schema 27
Figure 22 Visualization schema 27
Figure 23 Client directory 29
Figure 24 Registration UI 30
Figure 25 Login UI 30
Figure 26 Explore dataset page UI 31
Figure 27 Create post page UI 32
Figure 28 All posts page UI 33
Figure 29 Registration flowchart 34
Figure 30 Login flowchart 36
Figure 31 Flowchart for adding a visualization 38
Figure 32 Input data normalization code 39
Figure 33 Input data transformation code 40
Figure 34 Input data serialization code 40
Figure 35 InitializeVisualizations() method code 40
Figure 36 GenerateUniqueIndex() method code 41
Figure 37 AppendVisualizationToLocalStorage() method code 41
Figure 38 InitializeEditVisualizations() method code 41
Figure 39 GenerateUniqueEditIndex() method 42
Figure 40 AppendEditVisualizationToLocalStorage() method code 42
Figure 41 Storing request data code 42

Figure 42 parse_input_string() function code 43
Figure 43 Get_bigquery_data() function code 43
Figure 44 Generate_query_string() function code 44
Figure 45 Extract_labels_an_data() function code 44
Figure 46 Handling the received data in the frontend 45
Figure 47 Pie chart visualization code 45
Figure 48 Bar chart visualization code 45
Figure 49 Flowchart for deleting a visualization 46
Figure 50 HandleDelete function code 46
Figure 51 DeleteCreatePostVisaulization reducer code 47
Figure 52 RemoveVisualizationFromLocalStorage function code 47
Figure 53 DeletePostVisualization reducer code 48
Figure 54 Flowchart for creating a post 49
Figure 55 Actions related to creating a post code 50
Figure 56 Create post in DB 50
Figure 57 Visualization parameters extracted code 51
Figure 58 Actions related to adding a visualization to a post code 51
Figure 59 Server-side code for adding a visualization to the DB 52
Figure 60 Function for removing the visualizations item from local storage 52
Figure 61 Update a post flowchart 53
Figure 62 The post information retrieved code 54
Figure 63 The visualization data retrieved code 54
Figure 64 Actions related to saving a visualization to a post code 55
Figure 65 Server-side code for adding a visualization to a post 55
Figure 66 checking if the visualization in the DB exists in the postInEditing visualizations array code 56
Figure 67 Actions related to deleting a visualization from a post code 56
Figure 68 Server-side code for delete a visualization from the DB 56
Figure 69 Actions related to updating a post code 57
Figure 70 Server-side code for updating a post in the DB 57
Figure 71 Function called to remove the edit_visualizations item from the local storage 58
Figure 72 View a post flowchart 59
Figure 73 Actions related to getting a post by id code 60
Figure 74 Server-side code for finding a post by id 60
Figure 75 Code for the successful response for fetching the post by id 61
Figure 76 Error message display code for the rejected response 61
Figure 77 Rendering the visualization list component 61
Figure 78 Testing relationship between the User model and the Visualization model 62
Figure 79 Testing CRUD operations on the User Model 63
Figure 80 Testing relationship between User model and Post model 63
Figure 81 Integration testing output 63
Figure 82 Testing registering a duplicate user 64
Figure 83 Testing user registration 64
Figure 84 Testing user login 64
Figure 85 System testing output 65

Figure 86 Testing the get biquery data function 66
Figure 87 Unit testing output 66

1. Introduction

Around 80% of care quality commission providers will have digital social care records by March 2024 (Association, n.d.). This is only a small part of the efforts and investments taken by the UK government to enhance the digitalization of patients' health and care records. Their aim is to make use of the data to improve the experience of the care providers and transform the performance of this sector.

Taking advantage of this abundant data can help the social care decision makers develop a better view of the existing system. It will also help shape more suitable policies for the needs of the population.

It is with this goal in mind that the idea of developing an advanced visualization tool for the social care data came to be. With the help of visual analytics, we can transform complex case studies and hypothesis into digestible visualizations. They can be utilized to better understand the specificities of the social care journey in the UK and understand its different components.

1.1 Project description

This project aims to design and develop a visual analytics tool that supports quick insight generation for the adult social care data in the connected Yorkshire research database.

During the initial stages of the project, I worked closely with the Bradford city council and the research analysts to try to figure out the relevant features in the dataset and understand some of the common visual exploration needs of the analysts.

My goal is to help them understand the social care journey of the people in order to get a grasp on the needs of the community. This will be achieved by developing a tool that is able to help trace the journey of individuals through social care and identifying relevant hidden pattern contained in the data. The final prototype should enable the quick generation of insights from the datasets.

1.2 Novelty and contribution

The tool developed will be used to assist data analysts and stakeholders in understanding and interpreting the complex datasets related to adult social care. It will also help them formulate new hypothesis about the data.

Through the use of interactive visualizations and drilling down capabilities, the tool generated will be able to provide a holistic view of people's care journey in the UK. This will allow the easy identification of the recurrent patterns and bias in the data. Another key feature is supporting fast real time analysis to quickly identify the areas of concern and take the appropriate actions.

Overall, this project will present a step forward towards the field of social care data analytics by providing an intuitive tool that can help better understand the social care data.

1.3 Report overview

Throughout the report, I will try to explain the entire development process of the visual analytics tool starting from the literature review described in chapter 2. Then, in chapter 3, I will define and analyse the requirements of the system. Afterwards, I will go through the data analysis workflow followed by a detailed description of the design, implementation, and testing of the solution. Finally, in chapter 5 and 6, I will finalize the report by mentioning the achievements of the project, its limitations, and LSEP considerations as well as the conclusion.

2. Literature Review

In this section of the report, I will be discussing the related work done within the field of visual analytics as well as the challenges and solutions found in each stage of the workflow. I will also mention the tools employed in both frontend and backend web development. This step is essential to formulate an understanding of what has already been achieved in the field in order to base my work on successful strategies.

2.1 Visual analytics

2.1.1 Definition

Visual analytics was initially defined as "the science of analytical reasoning facilitated by interactive visual interfaces" (Thomas, 2005). It combines both visualizations and analysis techniques within the field of human computer interaction to better understand complex and large datasets (Keim and Al, 2010).

2.1.2 Applications

It has been used in various fields to gain insights from existing data. Tumour tissue characterization is an example that illustrates how visual analytics can enhance patients' diagnosis and treatment (Raidou et al., 2015). More recently, IRVINE was developed to understand the reasons behind electric engines errors in car manufacturing (Eirich et al., 2022). Both of these VA-based design studies make use of domain experts' knowledge and existing computational strength to support informed decision making (Andrienko et al., 2017).

2.1.3 Workflow

Understanding the process of generating useful insights from raw data is a challenging task due to the complexity and interdisciplinary aspects of the visual analytics activity (Collins et al., 2018). **Error! Reference source not found.** represents the visual analytics pipeline proposed by Keim et al followed by a description of each step in Table 1.

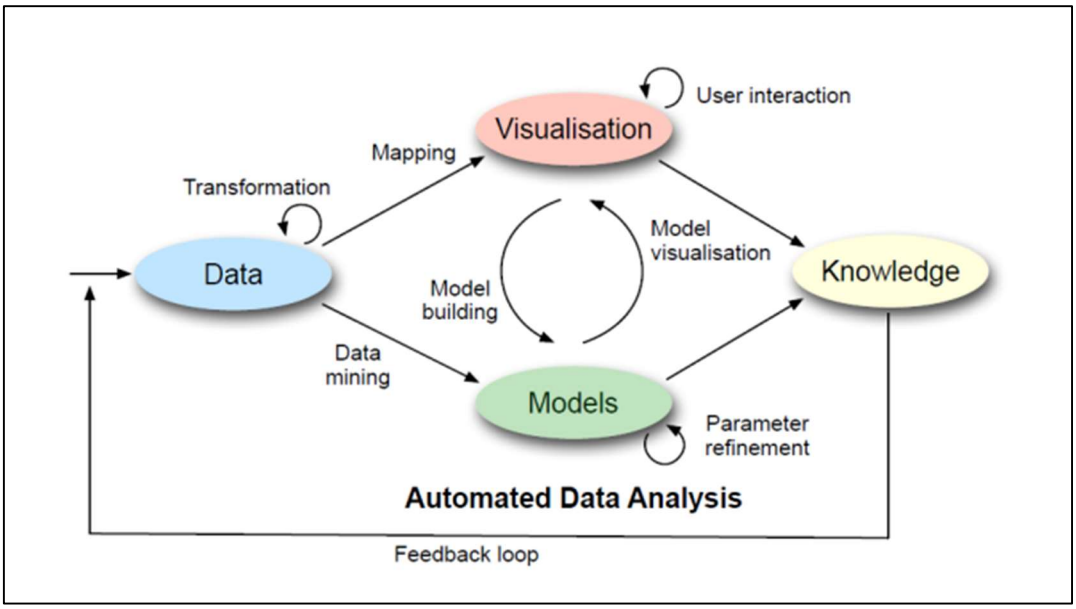


Figure 1. Visual analytics pipeline proposed by Keim et al

Step	Description
data transformation	Manipulate the raw data at hand to make it easier to analyse in order to reveal insights about the patterns and relationships.
visualization	Create visual representations to highlight the patterns in the data through appropriate visualizations.
Models	Create mathematical models that explain the patterns in the data and train them using a subset which can be used to classify the data, make predictions or identify clusters.
Knowledge	Understand complex datasets and use that knowledge to drive informed decision making

Table 1. Visual analytics pipeline stages description

2.1.4 Challenges and solutions

There are many challenges encountered in visual analytics. One of these challenges is the share volume of the data and its variety (Zikopoulos, 2012).

In healthcare (and social care) for example, millions of patients' records are collected over time generating huge databases of event sequence data (Guo et al., 2022). While the abundant healthcare data constitutes an opportunity for a wider analysis of the population, it can also be crippling for the data analysts. That is why different strategies have been developed to reduce the pattern variety and volume of the data (Du et al., 2017). These strategies are described in Table 2.

Strategy	Description
Data cleaning	Remove the duplicates in the data, fill any missing values, and correct errors (Hyunmo Kang et al., 2008).
Data wrangling	Change the structure of the data to make it more suitable for analysis in order to explore latent data abstractions (Bigelow, Williams and Isaacs, 2021).

Data focusing	Reduce the volume of the data by using subsets extracted from the main datasets (Reinartz, 1999) which might also yield better results for scalable machine learning algorithms (W. Aha and L. Bankert, 1995).
Apply analytic focusing	For temporal event sequence data, we can create a view that combines records with similar temporal patterns (Monroe, 2014) or apply a common filter to one of the attributes in a relational DB.
Measuring volume and variety	Define the scalability of the visualization by measuring the number of different elements (visualizations) on the dashboard/screen (Eick and Karr, 2002).

Table 2. Strategies used to deal with the data volume

Another common challenge is the design of appropriate visualization systems and evaluating them to effectively support the stakeholders' needs (Dimara et al., 2021).

For this purpose, a methodology was put forward to help guide design studies that deal with real-world problems called the "nine-stage design study methodology" (Figure 2) (Sedlmair, Meyer and Munzner, 2012).

I will be using this framework to guide my project and make sure the visualizations generated adhere to the stakeholders' requirements. Each stage of the methodology is described in table 3.

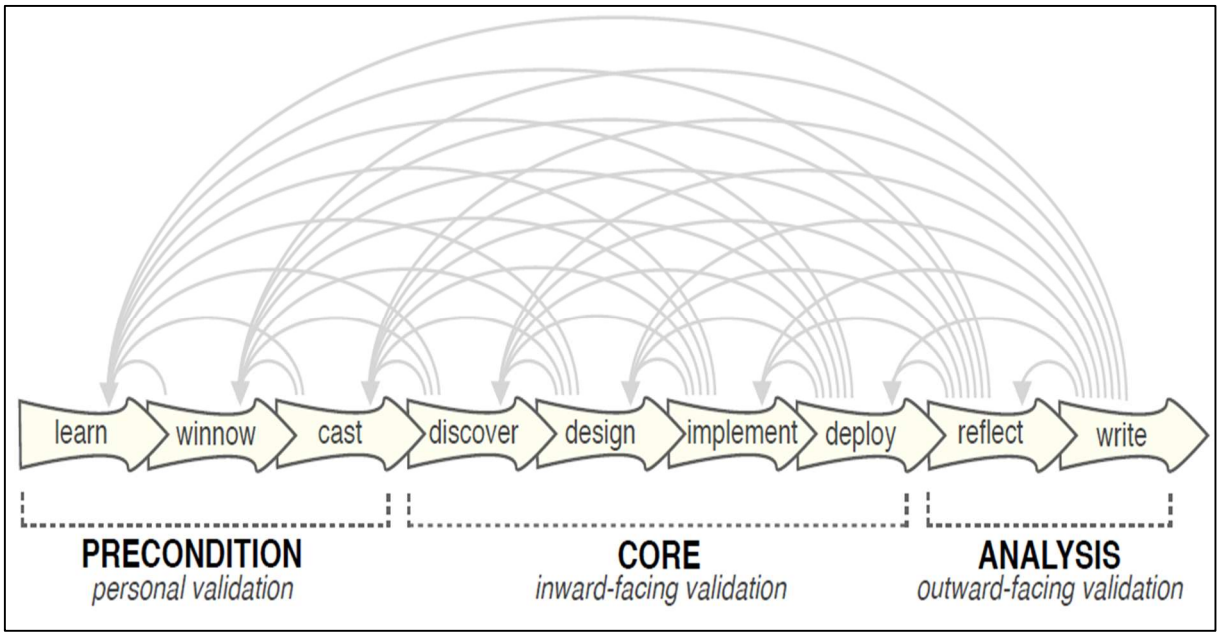


Figure 2. Nine-stage design methodology framework (Sedlmair, Meyer and Munzner, 2012)

Phase		Brief description
Precondition phase	Learn	Learn about the visualization literature
	Winnow	Select the appropriate collaborations
	Cast	Identify the roles of the collaborators selected
Core phase *(in the scope of this project)*	Discover	Characterize the problem and abstract it
	Design	Create visualizations that comply with the problem abstraction
	Implement	Develop visualization prototypes and test their usability
	Deploy	Release the final prototype and gather feedback
Analysis phase	Reflect	Critically reflect on the design study conducted and its contributions
	Write	Write the design study paper

Table 3. Description of the steps in the nine-stage methodology framework

2.1.5 Tools used in visual analytics

In the field of visual analytics, two leading tools are used across different industries due to their robust data analysis and visualization capabilities. These tools are Looker studio and Tableau.

The following table provides a comparative analysis between these two tools. This comparison aims to showcase each tool's strengths and limitations across various domains. The information is based on the features provided in the official Google Cloud website as well as the Tableau documentation. The criteria of comparison were based of the characteristics mentioned in the research conducted by Passlick et al. (2023) on self-service business intelligence and analytics.

Feature	Looker studio	Tableau
Data exploration	Provides a robust platform for data exploration using the lookML modelling language	Offers an intuitive drag-and-drop interface for data exploration
Data visualization	Offers a wide range of pre-existing visualizations as well as custom visualizations in JavaScript	Offers a wide range of pre-existing visualizations as well as custom visualizations
Data integration	Supports a wide range of data sources	Supports a wide range of data sources
	Allows real time data exploration	Provides options for live connections and in memory extracts
Collaboration	Allows sharing dashboards across the organization	Allows sharing, commenting and publishing dashboards
User entry skills	Requires a certain level of expertise especially for creating LookML models	Does not require any expertise or prior technical knowledge
Data management requirements	Requires robust data management practices	Offers flexibility and allows users to connect to the data where it resides
Development collaboration	Supports development collaboration	Supports development collaboration

Table 4 Comparison between Looker studio and Tableau

2.2 *Backend development*

Relational databases are a crucial part of any system that handles large amounts of diverse data types. In fact, they provide a well-structured environment for managing and storing application data. According to Connolly and Begg (2020), one of the most common approaches in backend development is to handle the data related operations through API calls. These calls are sent to the backend resulting in CRUD (Create, Retrieve, Update, Delete) operations.

In the context of backend development, a highly iterative approach is often preferred especially when the functionalities of the system are initially unclear. For that reason, Flask constitutes a viable choice for such systems. Its modular design offers adaptability to the ever-evolving needs of the developer. It also supports various extensions that can add several features to the system as described in the Flask official documentation. In fact, applications that use Flask range from single module-based systems to large scale web applications with multiple features.

Amongst the different relational databases that can be connected to Flask, PostgreSQL stands out amongst all other candidates. The table below provides a comparison of the 3 main databases considered for the solution namely: SQLite, MySQL, and PostgreSQL. The comparison focuses on features such as SQL compliance, concurrency, extensibility, speed

and finally use case which were retrieved from the comparative analysis conducted by Ostezer and Drake (2014).

Feature	SQLite	MySQL	PostgreSQL
SQL compliance	Low	Medium	High
Concurrency	Low	Medium	High
Extensibility	Low	Medim	High
Speed	High	High	Medium
Use case	Embedded applications	Web applications	Large scale web applications

Table 5 Comparing SQLite, MySQL, and PostgreSQL

2.3 Frontend development

Since the project will heavily rely on user interactions within an interface, the frontend development side constitutes an integral part of the solution. Amongst the various frontend technologies used, React is by far one the most popular. This is backed by KOMODO digital (2021) where it was voted the second most popular web framework.

React is a JavaScript library mainly used for building user interfaces. It is based on reusable components that can be rendered in any part of the application. This component-based architecture helps making the frontend code easier to maintain (Meta Open Source, 2023).

In addition to React, Redux, an open-source JavaScript library, can be employed to manage the application state. It is commonly used in conjunction with a framework (ex: React, Angular) to build user interfaces. One of its core features includes centralizing component state management in a store. This helps making the application state predictable and easier to manage (Js.org, 2015).

3. Requirements and Analysis

In this section of the report, I will be going through the objectives and requirements set by the stakeholders to generate the initial set of functional and non-functional requirements for the project. Additionally, I will conduct a comprehensive analysis of the requirements to identify the extract the system's features.

3.1 Requirements

Based on the discover stage of the nine-stage framework (Sedlmair, Meyer and Munzner, 2012) mentioned above, it is essential to work closely with the stakeholders and domain experts to make sure the visualizations generated are insightful. Thus, weekly meetings were held with both parties to help identify the project needs and requirements.

Initially, we discussed the general goals of the data scientists currently working on the adult social care data to formulate research hypothesis and identified 3 main objectives:

No	Objectives
1	Examine characteristics of individuals referred via the police and compare them to those referred by other means
2	Explore means to quantify the longitudinal trajectories of service involvement with adult social care users
3	Overlay place-based policing data around vulnerabilities and social care referral data

Table 6 User objectives

In the subsequent meetings, the domain experts identified more targeted requirements for the cohort of individuals referred to the adult social care system by the police. Based on that, here is the list of user requirements that should be validated in the development process:

No	User requirements
1	The demographics characteristics of individuals namely age, gender, ethnicity and any other relevant information available
2	Spatial distribution of individuals in terms of home of LSOA
3	Trajectories of service involvement within the social care system
4	Trajectories of service involvement with the broader linked systems

Table 7 User requirements

In accordance with the initial user needs, we can derive some initial functional and non-functional requirements. It is important to note that the development process of this design solution is ***iterative***. In fact, the requirements of the system will evolve and might change in the future.

Having that in mind, Table and Table describe the functional and non-functional requirements of the system.

No	Functional requirements
1	The system should be able to visualize temporal event sequence data
2	The system should provide a variety of visualization options
3	The system should allow the user to interact with the visualizations
4	The system should enable the user to explore the data and discover patterns
5	The system should support geospatial map visualizations
6	System should integrate with the google cloud adult social care dataset

Table 8. List of functional requirements

No	Non-functional requirements
1	The system should have an intuitive user-friendly interface
2	The system should be able to handle large amounts of data and perform visualizations without any significant delays
3	The system should be maintainable and easy to update
4	The system must ensure the integrity of the individuals' social care data and protect it from unauthorized changes or deletions
5	The system should protect the data from unauthorized access an comply to the regulations regarding the social care data privacy

Table 9. List of non-functional requirements

3.2 Analysis

In order to achieve a better understanding of the system, the functionalities must be properly defined based on the requirements gathered in the previous section. The goal here is to provide a higher-level view on how the system is intended to be used.

The primary actor in this context is the ***user or analyst*** who will create, view and edit post containing data driven visualizations.

The main ***use cases*** filtered according to the requirements would be:

- Register/login/logout
- View/edit/delete/create posts.
- Add/delete visualization (Add/Delete visualizations to the post)
- View dashboard

These use cases are illustrated by the use case diagram below.

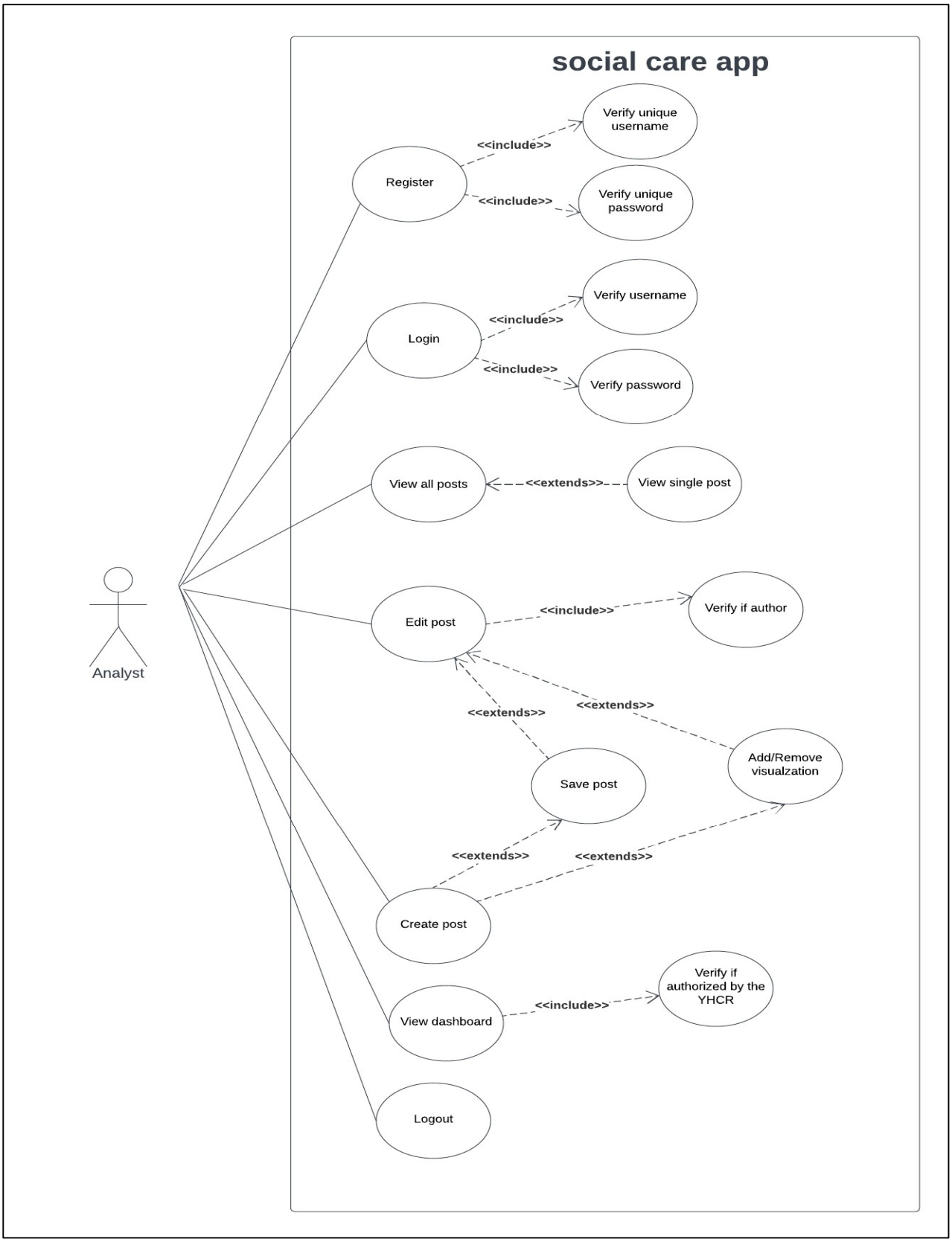


Figure 3 Use case diagram of the system

One of the major aspects of the system is the ability to store and manage the different entities created. The main entities in question will be User, Post and Visualization. Below is a simplified ER diagram representing the main relationships between these entities.

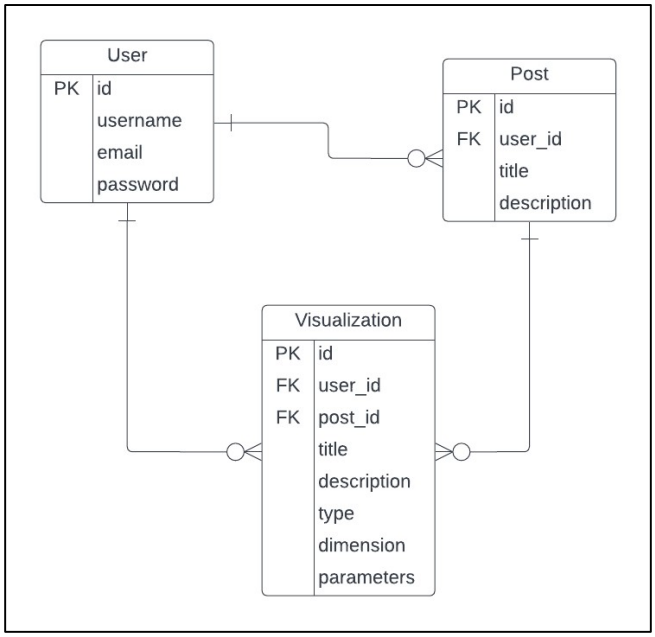


Figure 4 ER diagram for the User, Post, Visualization entities

The application's business logic will be handled by a Flask backend system, a lightweight and flexible python web framework well suited for *RESTful* APIs. In order to efficiently manage database objects and their relationships in flask, I will make use of *SQLAlchemy*, a popular ORM (Object Relational mapping) library that provides a high-level API for the application database operations. In this case the database in question is a *PostgreSQL*.

Each database table in the system *(As presented by the ER diagram above)* is represented by a class where instances of these classes correspond to rows in the respective tables. Therefore, the code responsible for handling database operations becomes more readable and maintainable since the database tables operations become centralized in classes representing the tables and their relationships. This concept is key in ensuring the backend uses the MVC pattern which I will talk about in more details in the design section of the report.

4. Design, Implementation, and Testing

In this section of the report, I will describe the data analysis workflow that led to the creation of the filtered social care dataset. Subsequently, I will explore the design of the solution in depth and highlight the implementation and testing strategies followed throughout the development process.

4.1 Data analysis workflow

Before developing the final prototype for the social care visual analytics solution, I spent a significant amount of time refining the social care dataset. My goal was to preserve the most relevant features as identified by the research analysts during the initial interviews. The data filtering stage guarantees that the final prototype would supply its users with valuable insights.

The data filtering process was achieved using BigQuery; a powerful and fully managed data Wearhouse in the Google Cloud ecosystem which allowed me to conduct the data analysis on the Connected Yorkshire research database seamlessly.

Initially, I started by extracting the fields relevant to the data exploration needs of the analysts working on the database. These fields included the demographic characteristics of all individuals that were contacted in the dataset. These characteristics include the postcode, age, year of death (if applicable), gender, and ethnicity.

```
CREATE OR REPLACE TABLE `CB_1937_IB.social_care_all_demographic_characteristics` AS
SELECT
    c.person_id,
    c.src_PostCodeDistrict AS postcode,
    DATE_DIFF(
        IFNULL(
            p.death_datetime,
            (
                SELECT extract_date
                FROM `yhcr-prd-phm-bia-core.CB_LOOKUPS.tbl_Dataset_ExtractDateRef`
                WHERE DataSetName = 'Adult social care'
            )
        ),
        DATE(p.birth_datetime),
        YEAR
    ) AS age,
    EXTRACT(YEAR FROM TIMESTAMP(p.death_datetime)) AS year_of_death,
    p.gender_source_value AS gender,
    p.ethnicity_source_value AS ethnicity
FROM
    `CB_FDM_AdultSocialCare.src_Contacts` AS c
LEFT JOIN
    `CB_FDM_AdultSocialCare.person` AS p
ON
    (c.person_id = p.person_id)
```

Figure 5 Extracting the demographic characteristics

Then I joined the contact information (*age at contact, contact route, contact reason, contact outcome)* details from the src_Contacts table to the previously generated table in order to gain some insights about the contact information of each individual in the dataset (route, source, reason, date of contact…).

```
CREATE OR REPLACE TABLE `CB_1937_IB.social_care_all_demographic_characteristics_and
_contact_info` AS
SELECT
    c.person_id,
    EXTRACT(YEAR FROM TIMESTAMP(p.death_datetime)) AS year_of_death,
    c.src_PostCodeDistrict as postcode,
    DATE_DIFF(
        IFNULL(
            p.death_datetime,
            (
                SELECT extract_date
                FROM `yhcr-prd-phm-bia-core.CB_LOOKUPS.tbl_Dataset_ExtractDateRef`
                WHERE DataSetName = 'Adult social care'
            )
        ),
        DATE(p.birth_datetime),
        YEAR
    ) AS age,
    p.gender_source_value AS gender,
    p.ethnicity_source_value AS ethnicity,
    ABS(
        DATE_DIFF(
            PARSE_DATE('%b-%y', c.src_MonthAndYearOfBirth),
            CAST(c.src_ContactDate AS DATE FORMAT 'dd/mm/yyyy'),
            YEAR
        )
    ) AS contact_age,
    c.src_contactSource as contact_source,
    c.src_ContactDate as contact_date,
    c.src_ContactRoute as contact_route,
    c.src_ContactReason as contact_reason,
    c.src_ContactOutcome as contact_outcome,

FROM
    `CB_FDM_AdultSocialCare.src_Contacts` AS c
LEFT JOIN
    `CB_FDM_AdultSocialCare.person` AS p ON (c.person_id = p.person_id)
```

Figure 6 Joining contact information to the dataset

In the next filtering cycle, I created a table that linked the cohort of individuals obtained from the previous operations to their GP visits information (duration of visit, location of visit…)

```
CREATE OR REPLACE TABLE `CB_1937_IB.social_care_all_demographic_contact_visit_info`
 AS
SELECT
      m.person_id,
      postcode,
      age,
      year_of_death,
      gender,
      ethnicity,
      contact_source,
      contact_age,
      contact_route,
      contact_reason,
      contact_outcome,
      v.care_site_id as visit_care_site_id,
      v.src_visitduration as visit_duration,
      v.src_visitlocation as visit_location,
      v.src_visitdesc as visit_desc,
      v.src_visitstartdate as visit_start_date,
      v.src_visitenddate as visit_end_date,
      v.src_visitprovider as visit_provider,
      v.src_visitstartplace as visit_start_place,
      v.src_visitendplace as visit_end_place,
      v.visit_concept_id as visit_concept_id,
      v.visit_type_concept_id as visit_type_concept_id,
      v.visit_source_concept_id as visit_source_concept_id,
      v.admitted_from_concept_id as visit_admitted_from_concept_id,
      v.discharge_to_concept_id as visit_discharge_to_concept_id
FROM
    `CB_1937_IB.social_care_all_demographic_characteristics_and_contact_info` AS m
LEFT JOIN
    `CB_FDM_PrimaryCare_v5.tbl_visit_occurrence` AS v
ON
    (m.person_id = v.person_id)
```

Figure 7 Joining GP visit information for each individual

Consequently, I added a new column to the dataset labelled 'visit_occurence' by calculating the time difference between the contact date and the visit date.

```
CREATE OR REPLACE TABLE `CB_1937_IB.social_care_all_demographic_contact_visit_info` AS
SELECT *,
       CASE
           WHEN cast(contact_date as date FORMAT 'dd/mm/yyyy') > visit_start_date THEN 'Before contact'
           WHEN cast(contact_date as date FORMAT 'dd/mm/yyyy') < visit_start_date THEN 'After contact'
           ELSE 'On contact'
       END AS visit_occurrence
FROM `CB_1937_IB.social_care_all_demographic_contact_visit_info`
```

Figure 8 Adding a new column to indicate the GP visit occurrence

I also updated the current gender column to only include 3 values 'M' for Male, 'F' for female and 'U' for Unspecified since there was a significant number of null values in that column which would affect the accuracy of the of the representations.

```
UPDATE `CB_1937_IB.social_care_all_demographic_contact_visit_info`
SET gender = CASE
                 WHEN gender IN ('M', 'Male', '1', '|', 'l') THEN 'M'
                 WHEN gender IN ('F', 'Female', '2') THEN 'F'
                 ELSE 'U'
             END
WHERE TRUE;
```

Figure 9 Updating the gender column in the dataset

In addition, I also calculated the absolute value of the number of months between the contact and the visit occurrence to get an idea about the elapsed time between both operations in the system and whether the contact outcomes or reasons are linked to that relationship.

```
CREATE OR REPLACE TABLE `CB_1937_IB.social_care_all_demographic_contact_visit_info`
 AS
SELECT
    *,
    ABS(
        DATE_DIFF(
            CAST(contact_date AS DATE FORMAT 'dd/mm/yyyy'),
            visit_start_date,
            MONTH
        )
    ) AS abs_months_between_contact_and_visit
FROM
    `CB_1937_IB.social_care_all_demographic_contact_visit_info`;
```

Figure 10 Adding the number of months between the contact and the visit occurrence

The next step in the data exploration phase included generating informative dashboards linked to the dataset obtained from the previous queries. For that aim, I took advantage of the google cloud ecosystem that I had access to and connected my dataset to Looker studio which provided interactive visualization capabilities (Google Cloud, n.d.). The final dashboard generated before starting the prototype design and implementation phase, included a bubble map representing the spatial distribution and contact reason of the individuals in the cohort.

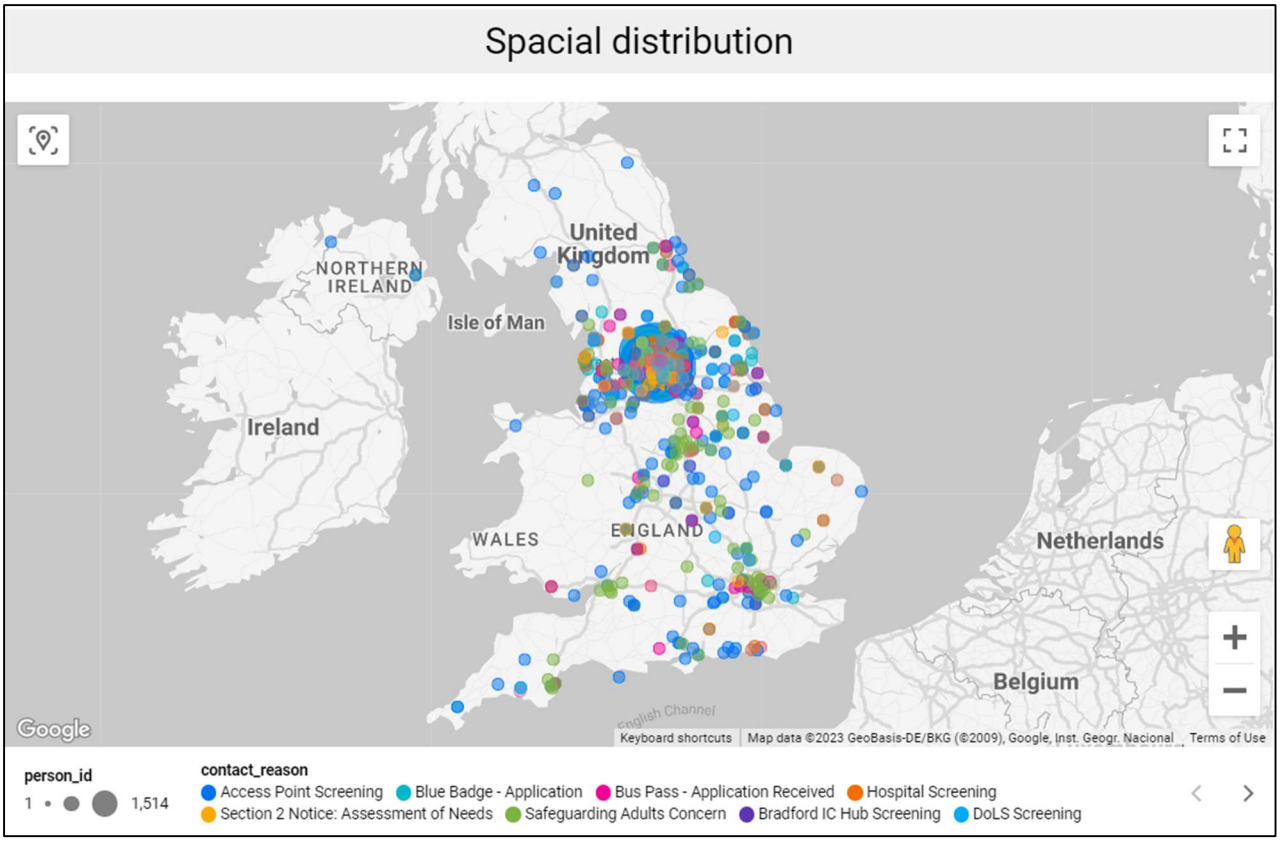


Figure 11 Spatial distribution of the individuals according to the reason of the contact

In order to represent the Genders and ethnicities in the cohort, I created a stacked bar chart for the gender and created an age range x-axis. On the other hand, I created a donut chart which represented the ethnicities' distribution. Finally, I added a table for the year of death and GP visit description followed by a pie chart representing the reason of contact as well as a bar chart containing the information related to the outcome of contact and the GP visit description.

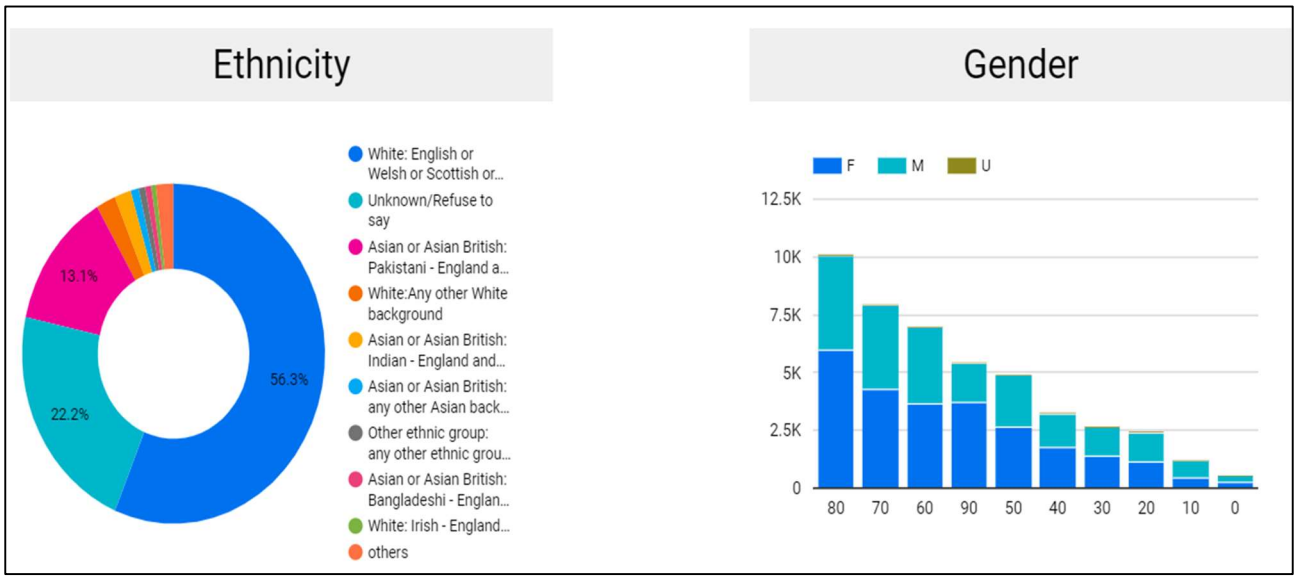


Figure 12. Visualizations for the ethnicity and gender distribution within the system

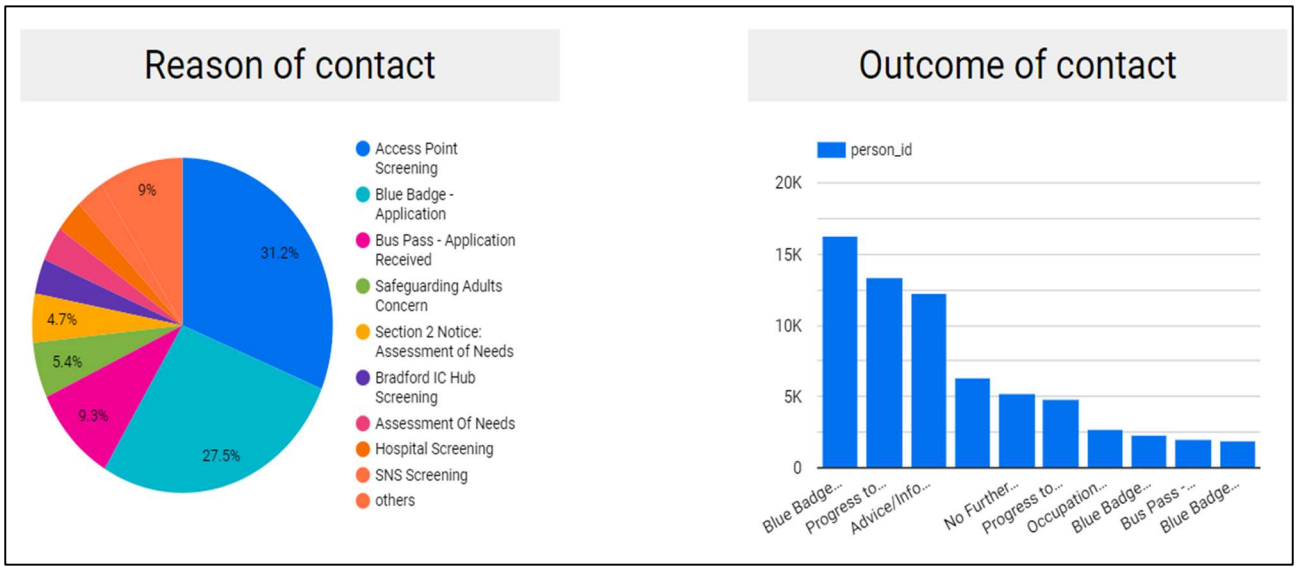


Figure 13 Reason of contact and the outcome of contact in the cohort

An important feature in Looker that I utilized during the development of the dashboard prototypes is that the visualizations created are connected therefore common filters can be applied interactively which enables drill down capabilities. The common filters I created were the age range, contact source, and visit occurrence.

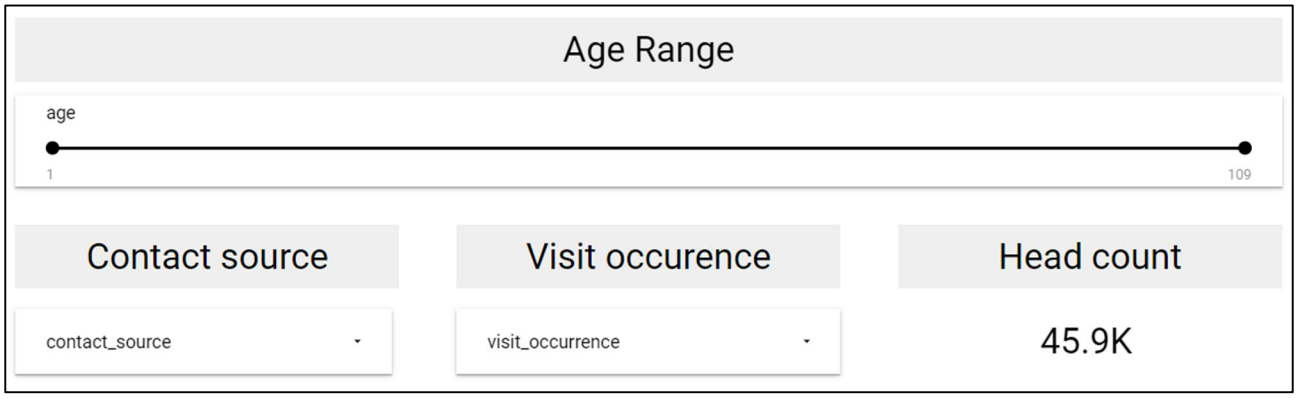


Figure 14 Common filters applied for the looker visualizations

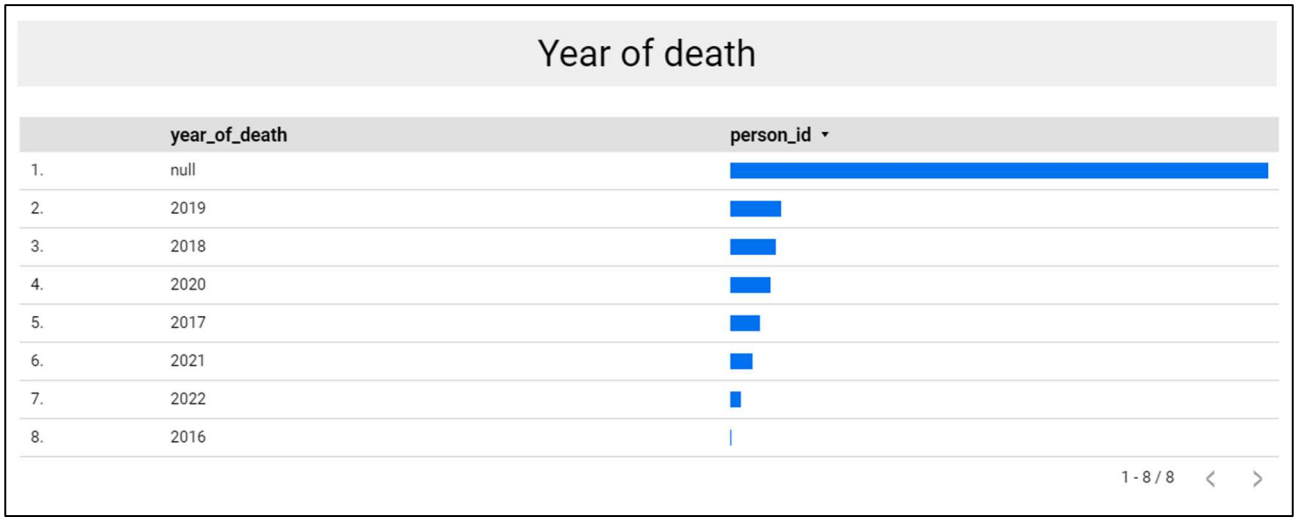


Figure 15 representing the year of death in the cohort

By the end of this stage, the newly filtered social care dataset was prepared and made accessible for API integration, enabling the rapid creation of dashboard prototypes. The access to the filtered dataset was enable and approved by the research centre.

4.2 Design

In this segment of the report, the system's architecture is discussed alongside the frontend and backend structures. At the end of the chapter, I included an overview of the user interfaces portraying the primary pages with the application.

4.2.1 Architecture

For the final prototype implementation, I opted for an overall *client-server* architecture that effectively separates concerns during the development process. Here is an overview of the system architecture components and their connections:

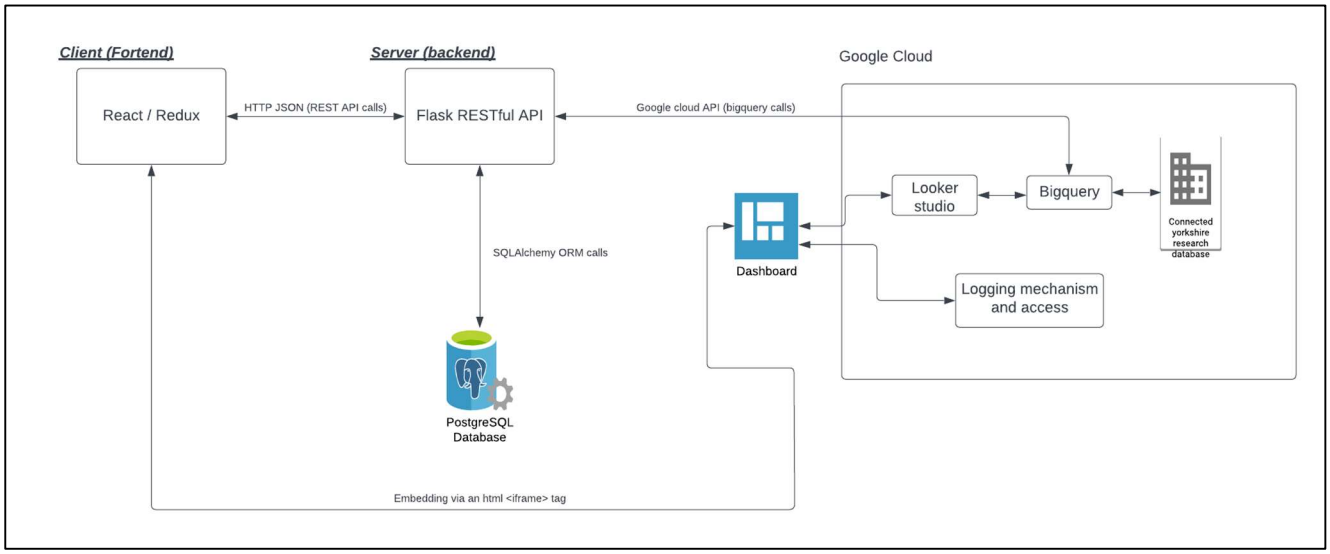


Figure 16 Overview of the system architecture

In this architecture, a Flask backend is responsible for managing all incoming request calls. It also establishes a connection with the filtered social care dataset created at the end of the data analysis workflow.

In addition to the google cloud dataset, The backend also communicates with a remote PostgreSQL database for storing and retrieving user information. However, it is important to mention that the database is exclusively used for storing visualization parameters associated with the visuals created by the user. The actual data is fetched via the google BigQuery API.

On the other hand, I used React/Redux to handle the user interface and state management throughout the application and components interactions. The client also contains a finalized dashboard created via Looker studio. The dashboard was embedded in the Explore dataset page of the application.

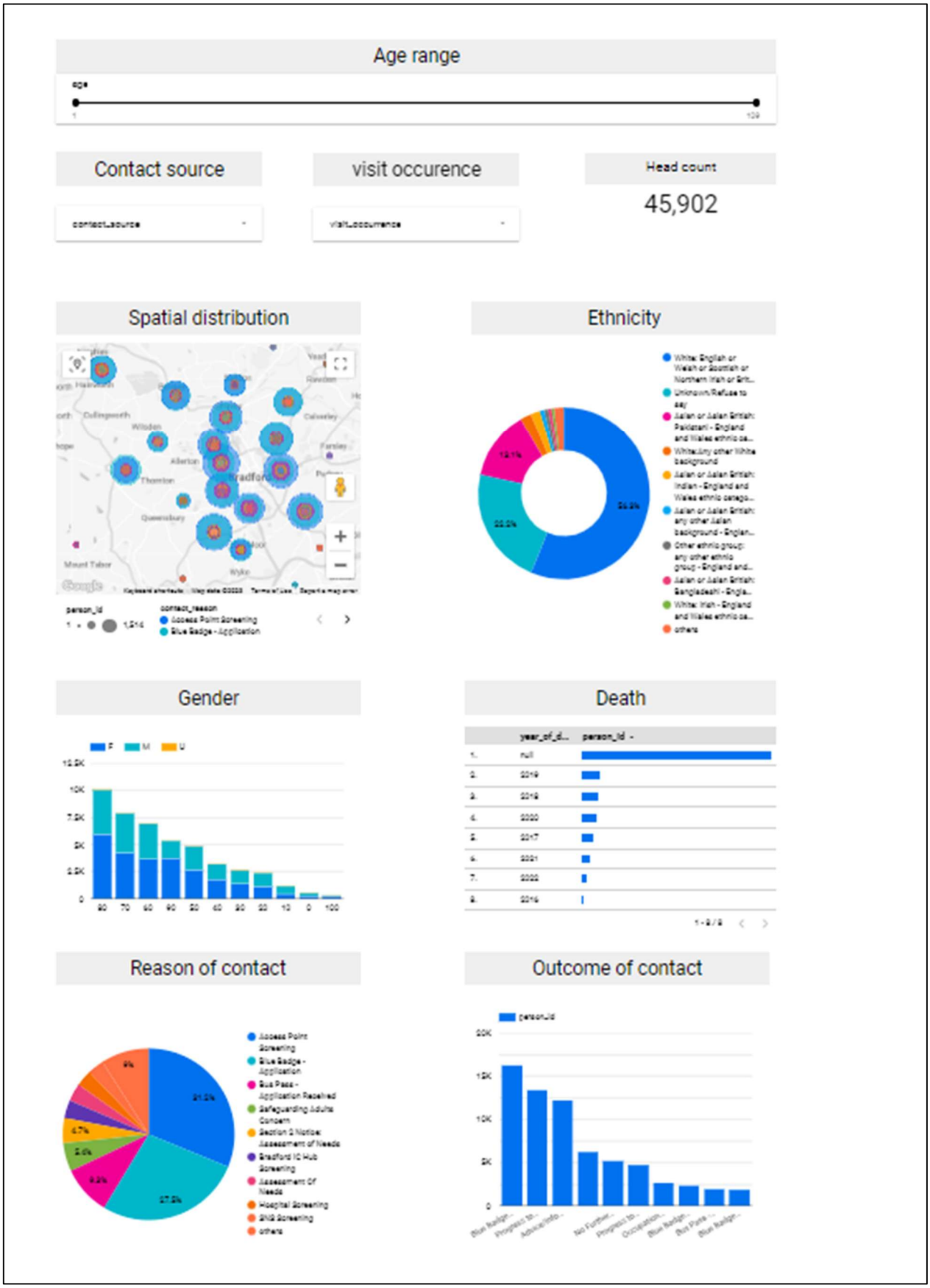


Figure 17 Dashboard embedded in the explore dataset page

Having a Flask framework for the backend enables rapid development and easy integration with other tools, libraries, and packages. Some of the packages I used during the development stage were JWT (to handle the authorization access token and current user information in my application) and SQL Alchemy which I already mentioned in the analysis section.

On the other hand, React is a powerful JavaScript library for building complex user interfaces. It promotes a component-based architecture that encourages code reusability and maintainability. Combined with Redux and Redux toolkit in this case, it effectively manages the entire application state ensuring a responsive user experience and a structured approach to component state management. As a result, the code becomes more maintainable and flexible.

4.2.2 Backend

The backend logic of the solution is found within the Server directory.

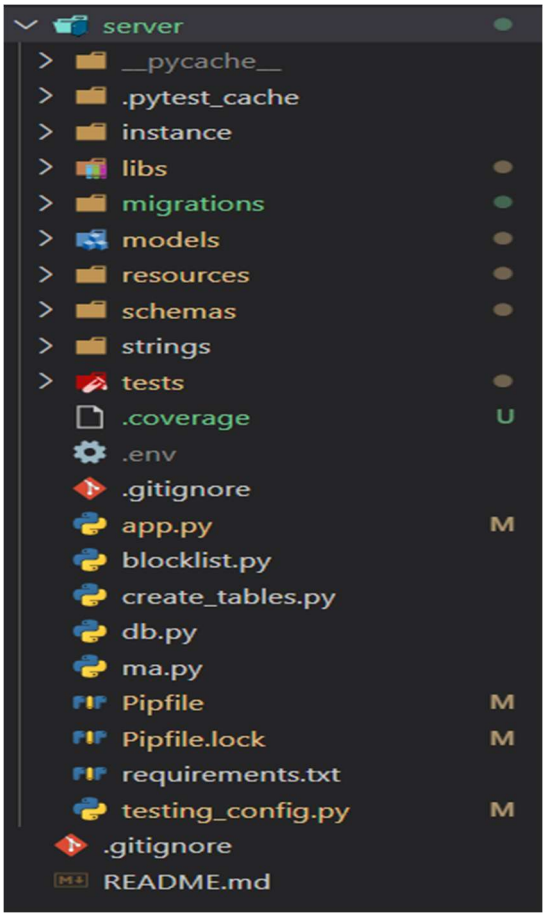


Figure 18 Server directory

It includes the Flask RESTful API, designed following a slightly modified Model-View-Controller (MVC) design pattern. This provided a structured approach for building the application in a way that separates concerns which improved the maintainability and scalability of the backend code.

The traditional MVC architecture has been slightly modified in this case due to the View layer which traditionally deals with the UI. In the context of a Flask RESTful API the View layer corresponds to the HTTP endpoints that handle HTTP requests and return data (JSON format) rather than a visual interface.

Therefore, the implementation Is divided into 3 distinct layers. Each of these layers will be described in detail in the following sections.

1. Models

The models directory contains representations of the PostgresSQL database tables. As mentioned in the requirements' analysis section, SQLAlchemy facilitates object relational mapping. Therefore, instead of using SQL queries, we can interact with the database using Python objects. In this prototype three main classes namely -UserModel, PostModel and VisualizationModel- encapsulate these objects and present the Models that the controllers will interact with.

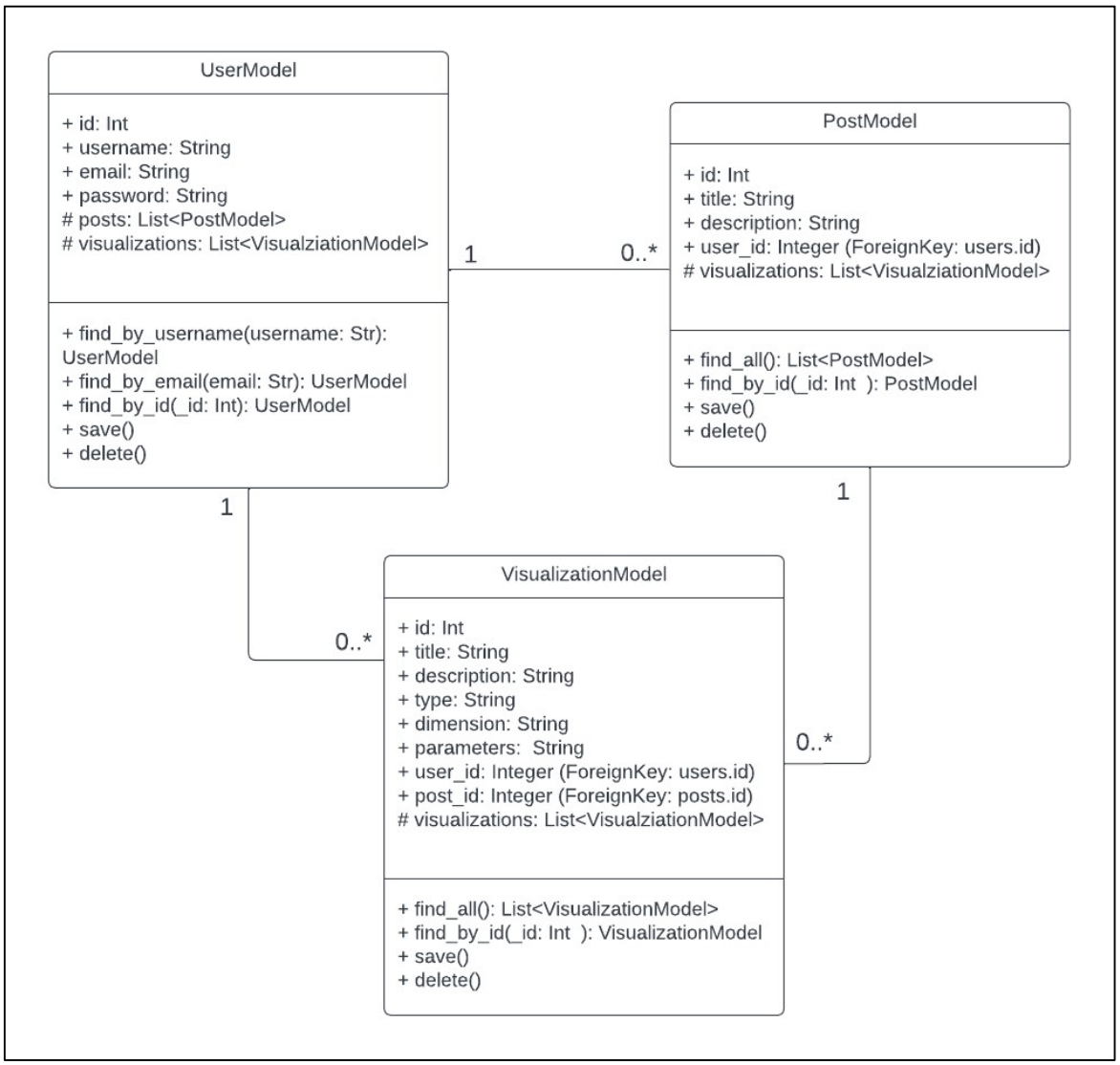


Figure 19 Class diagram for the Backend Models

For more details, I also included a table describing each Model class within the system and its components. It is important to emphasize that the process of identifying and structuring the different models is crucial in implementing the MVC pattern for the backend as they represent the data layer for the entire application architecture.

Model Class	Description of the Class
UserModel	- Represents a single user within the system - Includes fields relevant to the user's interactions within the platform (**username**, **password**, **email**). - 2 one-to-many relationships are established within the model linking the users table to the PostModel named **posts** and VisualizationModel named **visualizations**. - Includes class methods for fetching users by email, username or id.
PostModel	- An abstraction of a dashboard created by the user within the application - Includes fields such as: **title, description** and **user_id** (Foreign key linking back to the UserModel). - A post will have zero or many visualizations represented by the **visualizations** field. - The class contains a method for fetching a post by id (find_by_id) as well as fetching all posts (find_all).
VisualizationModel	-Represents a visualization associated with a particular post. - Captures unique details that represent each visualization created within the system. - Includes fields such as **description**, **type**, **dimension**, **parameters** , **user_id** (Foreign key linking to the UserModel) and **post_id** (foreign key linking to the PostModel).

Table 10 Model classes description

These models encapsulate the main application business logic and effectively promote data integrity by establishing relationships between the different entities as displayed in the class diagram.

2. Controllers

The controllers in this context are represented by different Resource classes included in the resources directory. All these resources are subclasses of the Resource class which is the building block for creating Restful APIs in Flask. These resource classes are responsible for

handling HTTP requests logic and responses following the Flask guidelines for the RESTful API development.

The resources directory in my project contains various classes that serve as controllers in the system each inheriting from the Resource class. These controller subclasses have the responsibility of managing the HTTP requests (POST/GET/PUT/DELETE) logic within the application.

Within the resources directory each filename (user.py, post.py, visualization.py, post.py, dataset.py) contains different Resources responsible for managing the Model associated with it.

It is important to note that each Resource subclass can only contain one of each type of HTTP request which will be associated with a specific route in the API. For that reason, I had to create multiple resources in each file in order to handle different aspect of the system functionality.

The table below describes the Resource classes associated with each Model.

Models	Controllers	Methods	Description
UserModel	**UserRegisterResource**	post()	- The client is required to register using a username, email and password. - Both username and email have to be unique and not previously registered - Upon successful registration the password entered is hashed then the user information is saved in the DB - As a response, the user schema is returned alongside the access_token and the refresh token *(useful for handling the logged in user accessibility)*
	UserLoginResource	post()	-The client is required to login using a correct username and password - upon successful login, the user schema is returned alongside the access_token and the refresh_token.
	TokenRefreshResource	post()	-Returns a new access token allowing the user to stay logged in without providing his credentials again
	UserResource	get(user_id)	-If the user with the provided id exists, it returns the user schema
		delete(user_id)	-If the user with the provided id exists, it deletes the user with the provided id from the database
	UserAllPostsResource	get(user_id)	-If the user with the provided id exists, it returns the posts list schema associated with the user

	UserAllVisualizationsResource	get(user_id)	-If the user with the provided id exists, it returns the visualizations list schema associated with the user
	UserListResource	get()	-Returns a user list schema of all users in the database
		delete()	-Deletes all users in the database
PostModel	**PostResource**	post()	-The client provides the title and the description of the post -If the client is logged in, a new post is saved to the database with the title and description provided as well as the user_id of the client
		put(post_id)	-If the post with the provided id exists and the client sending the request is the author of the post, the post in the database is updated with the title and description provided by the client
		delete(post_id)	-If the post with the provided id exists, and the client sending the request is the author of the post, each visualization in the post is deleted from the database then the post itself is deleted.
		get(post_id)	-If the post with the provided id exists, the post schema is returned
	PostAllVisualizationsResource	get(post_id)	- If the post with the provided id exists, the visualizations associated with the post are returned as a visualization_list schema
		delete(post_id)	If the post with the provided id exists, the visualizations associated with that post are deleted from the DB
	PostListResource	get()	-Returns a post list schema of all posts in the database
		delete()	-Deletes all posts in the database
VisualizationModel	**VisualizationResource**	post(post_id)	-The client provides the title, description, type, dimension, parameters -If the client is logged in, a new visualization is saved to the post with the provided post_id and the user_id
		delete(visualizaiton_id)	-If the visualization with the provided id exists, and the client sending the request is the author of the visualization, the visualization is deleed from the DB

		get(visualization_id)	-If the visualization with the provided id exists, the visualization schema is returned
		put(visualization_id)	-If the visualization with the provided id exists and the client sending the request is the author of the visualization, the visualization in the database is updated with the title, description, type, dimension, and parameters provided
	VisualizationDataResource	get(visualization_id)	-If the visualization with the provided id exists, the parameters of the visualization are extracted and used to generate a query which will be sent through the BigQuery API to retrieve the necessary data for the visualization
		post()	The visualualization parameters are retrieved from the input and used to generate a query which will be sent through the bigquery API to get the necessary data for the visualization
	VisualizationListResource	get()	-Returns a visualization list schema of all visualizations in the database
		delete()	-Deletes all visualizations in the database

Table 11 Controllers associated with the Models

Aside from the 3 main Models in the application (User, Post, Visualization), I created resource classes that facilitate the retrieval of labels and column values from the dataset through the google big query api. These resource classes can be found in the dataset.py file.

Controllers	Methods	Description
DatasetLabelsResource	get()	-Returns the dataset labels
DatasetValuesResource	get(column_name)	-Returns the unique values in the column specified in the column_name

Table 12 Controllers associated with the Dataset

3. Views

The views represent the endpoints exposing the data to the client. In this application, the data is structured in the JSON format. In order to manage this representation, I used the Marshmallow library, a powerful Python-based library employed in object serialization and

deserialization. Marshmallow has been used to define the schema for the 3 main entities in the application: User, Post, Visualization.

The UserSchema represents the data structure for User objects including fields like id, username, email, password and nested fields like posts and visualizations.

```
class UserSchema(Schema):
    id = fields.Integer(dump_only=True)
    username = fields.String(required=True)
    email = fields.Email(required=True)
    password = fields.String(required=True, load_only=True)
    posts = fields.Nested('PostSchema', many=True, exclude=('user',))
    visualizations = fields.Nested('VisualizationSchema', many=True, exclude=('user',))
```

Figure 20 User schema

The PostSchema represents the structure for the Post objects including fields like id, title, description, and user_id. The schema also includes nested User and Visualization schemas.

```
class PostSchema(Schema):
    id = fields.Integer(dump_only=True)
    title = fields.String(required=True)
    description = fields.String(required=True)
    user_id = fields.Integer(required=True, load_only=True)
    user = fields.Nested('UserSchema', exclude=('posts', 'visualizations'))
    visualizations = fields.Nested('VisualizationSchema', many=True, exclude=('post',))
```

Figure 21 Post schema

The VisualizationSchema establishes the structure for Visualization objects including fields like id, title, description, type, dimension, parameters, user_id and post_id along with the nested Post and User schemas.

```
class PostSchema(Schema):
    id = fields.Integer(dump_only=True)
    title = fields.String(required=True)
    description = fields.String(required=True)
    user_id = fields.Integer(required=True, load_only=True)
    user = fields.Nested('UserSchema', exclude=('posts', 'visualizations'))
    visualizations = fields.Nested('VisualizationSchema', many=True, exclude=('post',))
```

Figure 22 Visualization schema

The following table provides an overview, mapping each endpoint (view) to its corresponding resources (controllers) within the API.

	Endpoint	Controllers
User related endpoints	/register	UserRegisterResource
	/login	UserLoginResource
	/user/<int:user_id>	UserResource
	/users	UserListResource
	/user/<int:user_id>/posts	UserAllPostResource
	/user/<int:user_id>/visualizations	UserAllVisualizationsResource
	/token/refresh	TokenRefreshResource
Post related endpoints	/post/<int:post_id>	PostResource
	/posts	PostListResource
	/post/<int:post_id>/visualizations	PostAllVisualizationsResource
Visualization related endpoints	/visualization/post/<int:post_id>	VisualizationResource
	/visualization/<int:visualization_id>	VisualizationResource
	/visualizations	VisualizationListResource
	/visualization/<int:visualization_id>/data	VisualizationDataResource
	/visualization/data	VisualizationDataResource
Dataset related endpoints	/dataset/labels	DatasetLabelsResource
	/dataset/<string:column_name>/values	DatasetValuesResource

Table 13 Mapping of the endpoints to the controllers

4.2.3 Frontend

The frontend logic of the solution is found within the Client directory.

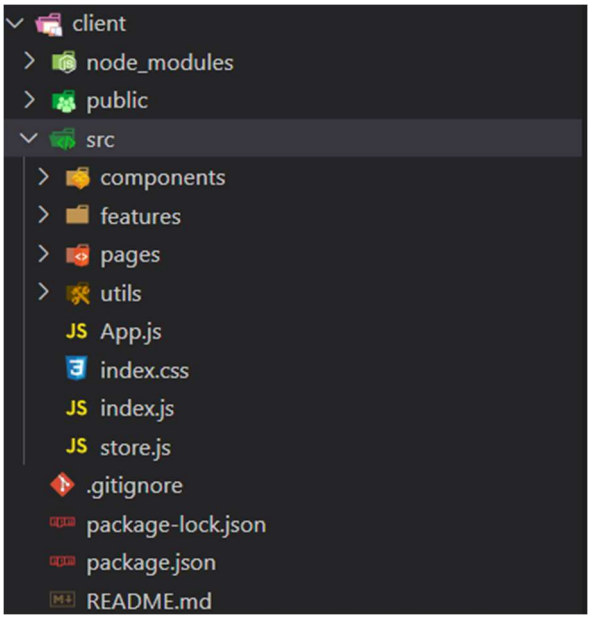


Figure 23 Client directory

The directory is organized into several subdirectories and files each serving a specific purpose in the application.

Directory/file	Description
Components	- This directory contains reusable UI elements - Each component is responsible for rendering a part of the UI - Components can be nested and reused across different parts of the application (Ex: FinalPostCompoenent, FinalPostListComponent)
features	- Contains the state management associated with the different features of the application - Each feature corresponds to a specific functionality of the application - The state of each feature is managed independently using a slice and a thunk file
Pages	- Contains components that correspond to different pages in the application - Each page component consists of subcomponents that together form a page
Index.js	-Entry point of the application - Responsible for rendering the main App component and setting up the Redux store
App.js	-Contains the main layout of the application and the routing of the different pages

Store.js	-Includes the root reducer of the application

Table 14 Client directory structure

4.2.4 User interfaces

In order to create an intuitive and smooth user experience, I decided to incorporate AntD (Ant Design) components and MDB (Material Design for Bootstrap) react components in my project.

These 2 popular UI libraries provide a wide range high quality component that mostly adhere to the design principles and facilitate the creation of visually appealing components which are used in the different UI views.

1. Registration UI:

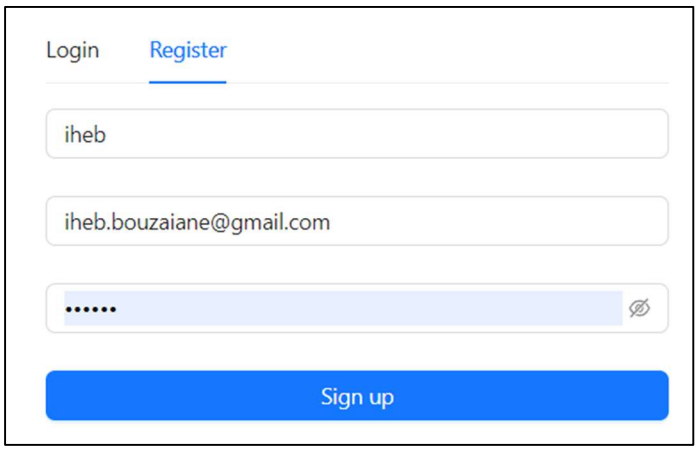


Figure 24 Registration UI

2. Login UI:

Figure 25 Login UI

3. Explore dataset page UI:

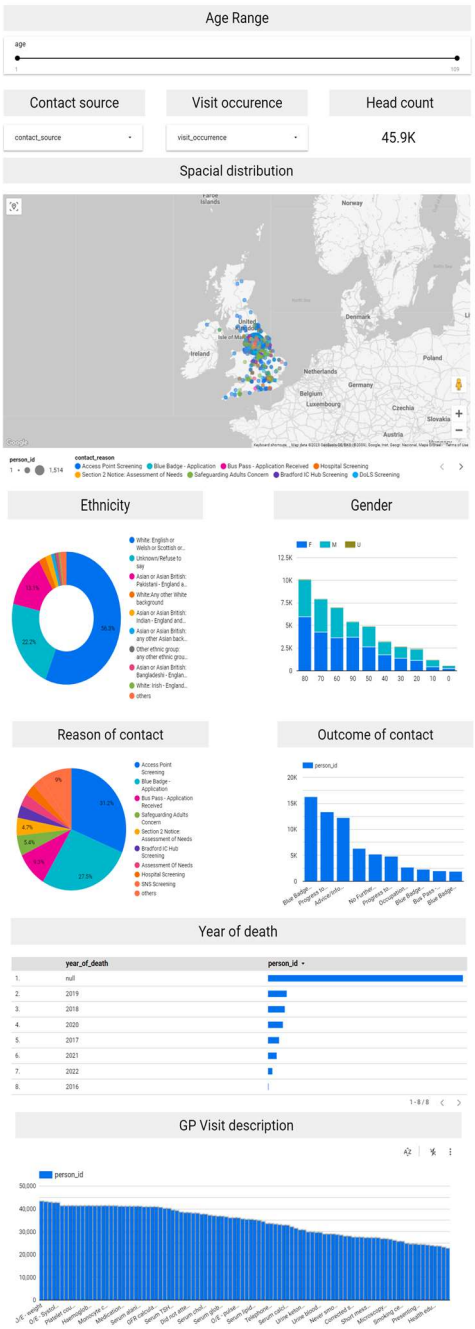


Figure 26 Explore dataset page UI

4. Create Post page UI

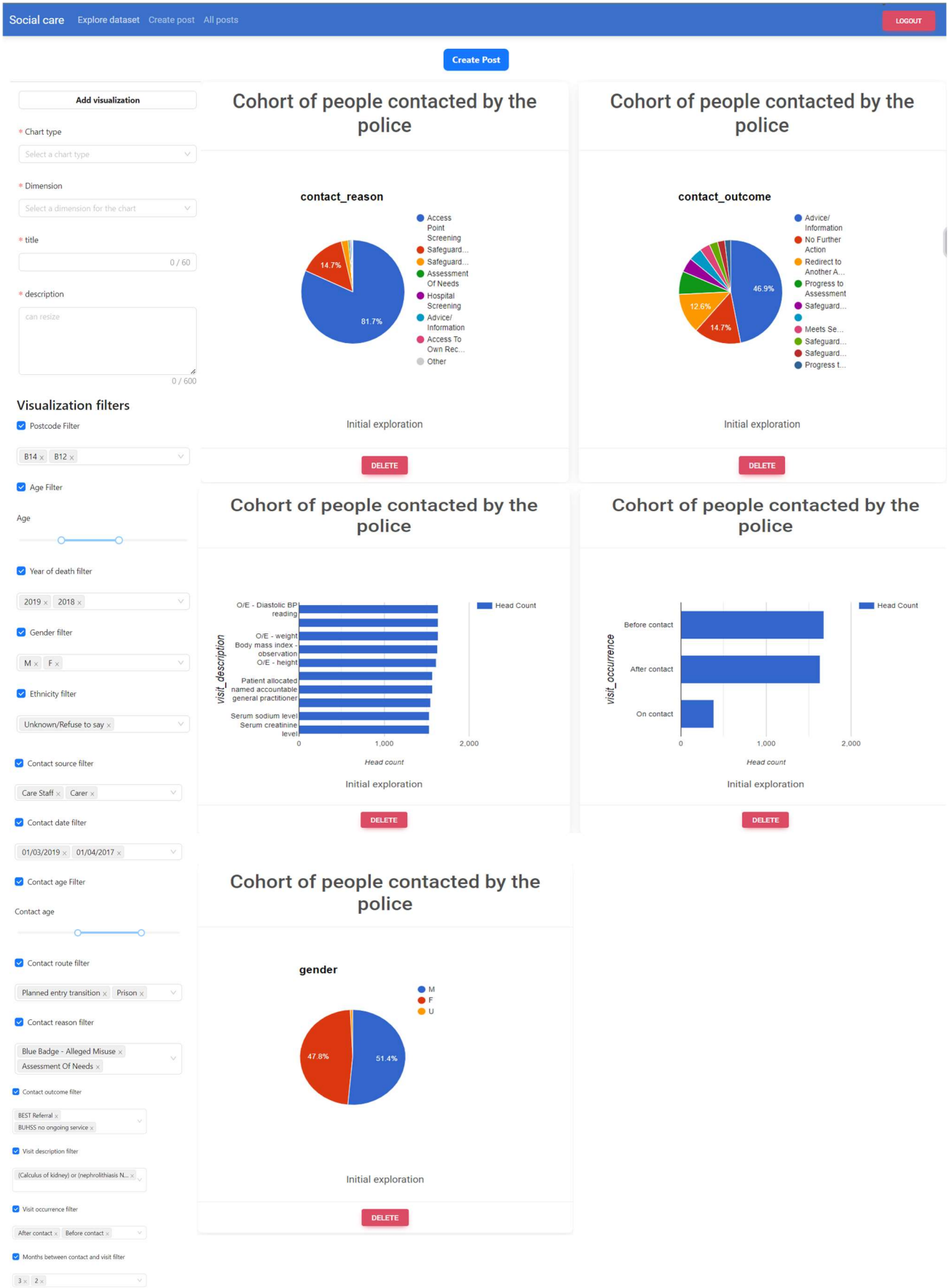


Figure 27 Create post page UI

5. All posts page UI

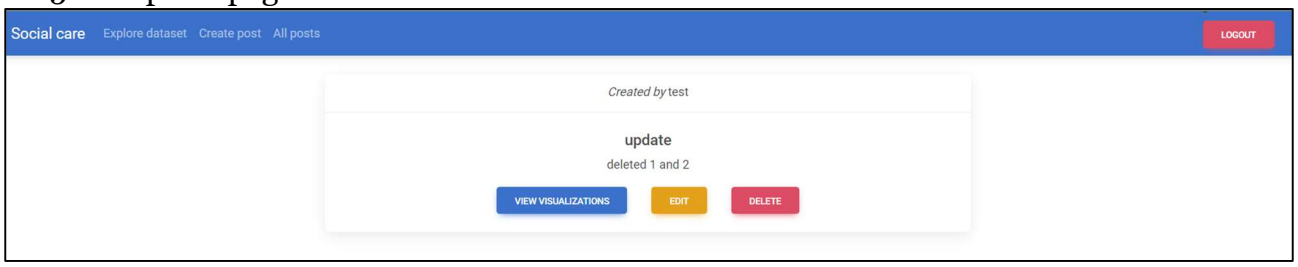


Figure 28 All posts page UI

4.3 Implementation

In this section, I have incorporated flowcharts that illustrate the primary functionalities of the application. Each flowchart is complemented by a comprehensive explanation of each step as well as the relevant code snippets. For the sake of clarity, the flowcharts encapsulate key steps rather than every detail in the code. The flowchart descriptions however provide more insights into each step.

Registration flowchart:

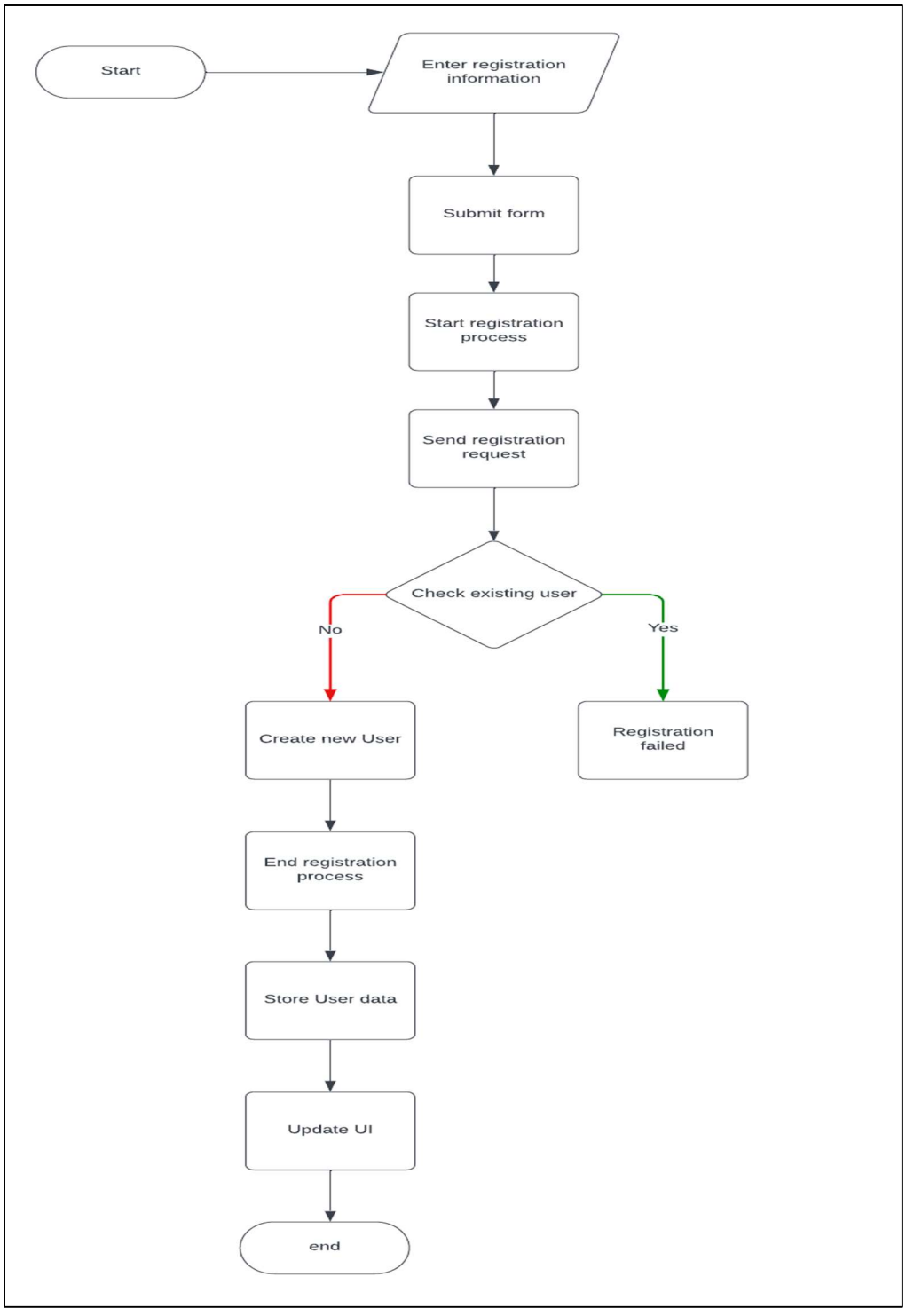


Figure 29 Registration flowchart

Flowchart description:

1. ***Start:*** The process starts when the user decides to register

2. ***Enter registration information:*** The user enters username, email, password on the frontend

3. ***Submit form:*** The user submits the registration form which sends the registration information to the Redux store

4. ***Start registration process:*** The redux store starts the registration process. It changes its state by setting isLoading to True to reflect that a registration process is ongoing

5. ***Send registration request:*** The Redux store sends a POST request to the Flask server with the user's registration information (username, email, password)

6. ***Check Existing user:*** The flask server checks in the database if the username or email already exists
    a. If yes: the process moves to 11 "Registration failed"
    b. If no: the process moves to 7 "Create new user"

7. ***Create new user:*** The Flask server creates a new user in the DB and generates an access and refresh tokens for the user

8. ***End registration process:*** The Redux store ends the registration process. It updates its state with the User data and sets isLoading to false

9. ***Store User data:*** The Redux store saves the user data and tokens to local storage for persistent login sessions

10. ***Update UI:*** The frontend updates the UI to reflect successful registration by displaying a success toast message and navigating to the create-post page

11. ***Registration failed:*** A fail toast message is displayed

12. ***End:*** process ends if the registration is successful

Login flowchart:

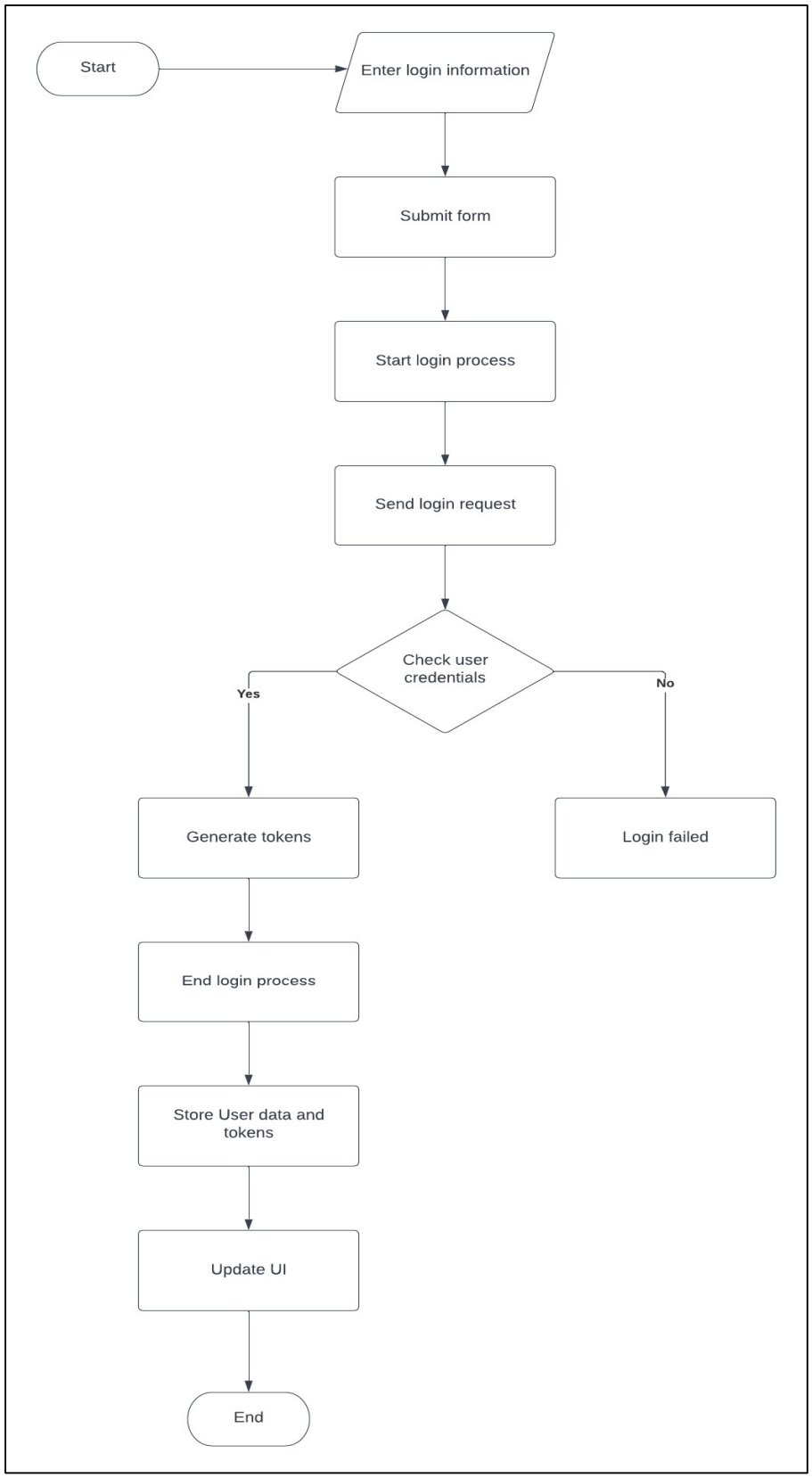


Figure 30 Login flowchart

Flowchart description:

1. ***Start:*** the process starts when the user decides to login

2. ***Enter Login information:*** User enters his login information into the frontend (username and password)

3. ***Submit form:*** User submits the login form which sends the login information to the Redux store

4. ***Start login process:*** The Redux store starts the login process. It changes its state to reflect that a login process is ongoing by setting isLoading to true.

5. ***Send Login request:*** the redux store sends a POST request to the flask server with the user's login information

6. ***Check user credentials:*** The flask server checks the database if the entered username and password match any existing user

    a. If yes: The process moves to 7 "generate tokens" step
    b. If no: The process moves to 11 "Login failed" step

7. ***Generate tokens:*** The flask server generate access and refresh tokens for the user

8. ***End Login process:*** The Redux store ends the login process. It updates its state with the user data and isLoading is set to false

9. ***Store user data and tokens:*** The Redux store saves the user data and tokens to local storage for persistent login sessions.

10. ***Update UI:*** The frontend updates the UI to reflect the successful login by displaying a success toast message and navigating to the create-post page.

11. ***Login failed:*** a fail toast message is displayed

12. ***End:*** process ends if login is successful

Adding a visualization flowchart:

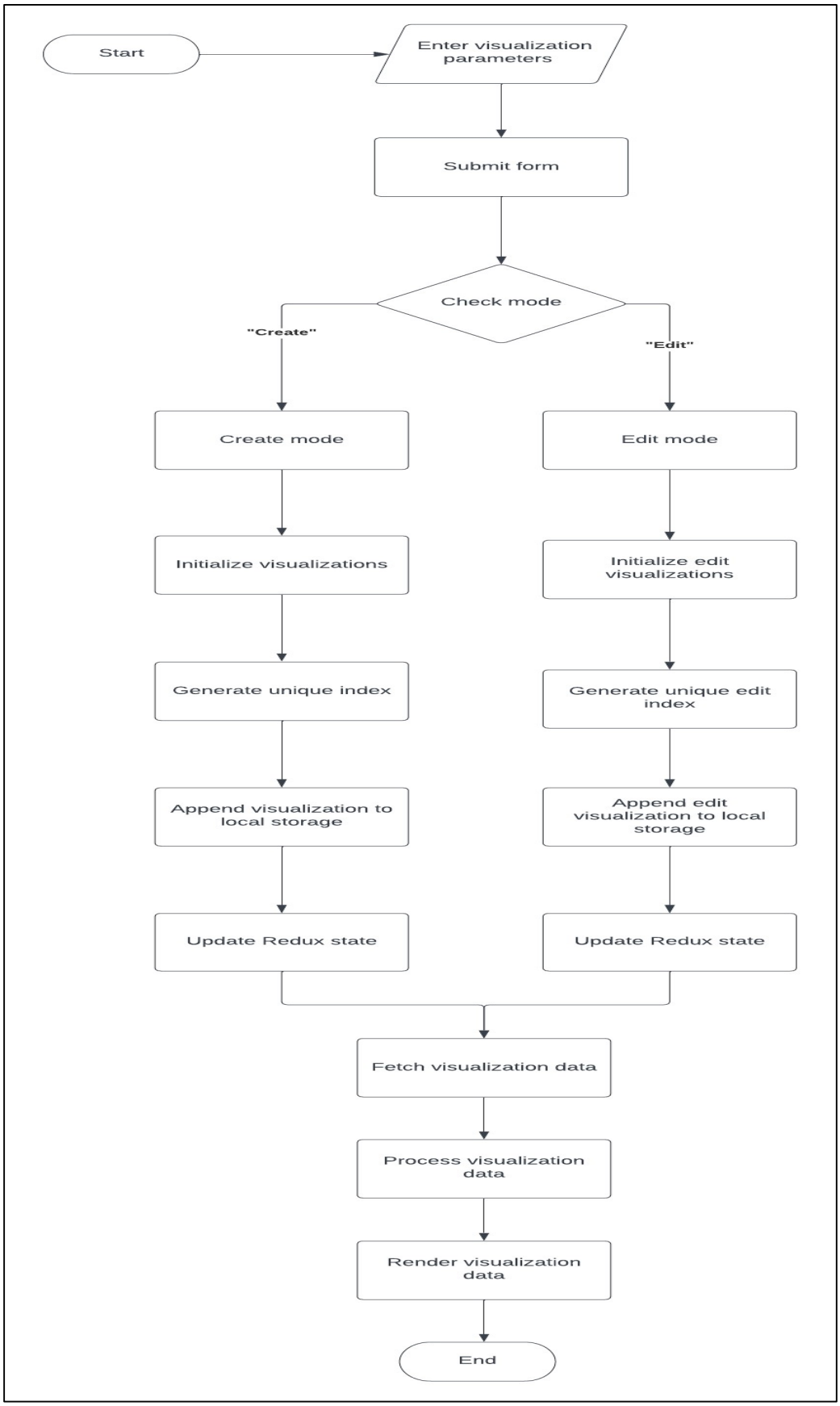


Figure 31 Flowchart for adding a visualization

Flowchart description:

1. ***Start:*** User decides to add a visualization

2. ***Enter visualization parameters:*** User enters the required parameters for the visualization (*title, description, type, dimension*) and filters (*age, postcode, contact source…*)

3. ***Submit form:*** User submits the form triggering the ***addVisualization*** event handler
    a. The type, title, description, dimension, and parameters inputs are converted using the **convert*InputValues()*** utility function found within the *QueryEngineInput.js* file. The function in this context is a transformation function. The goal behind this transformation is to standardize the input data.

        i. Data normalization: The function starts with a declarative mapping of field names to a common schema

```
const fieldMapping = {
    postcode: {
        "field-name": "postcode",
        "condition-type": "multi-select-text",
    },
    age: {
        "field-name": "age",
        "condition-type": "range-slider",
    },
    year_of_death: {
        "field-name": "year_of_death",
        "condition-type": "multi-select-number",
    },
    gender: {
        "field-name": "gender",
        "condition-type": "multi-select-text",
    },
    ethnicity: {
        "field-name": "ethnicity",
        "condition-type": "multi-select-text",
    },
    //...
};
```

Figure 32 Input data normalization code

        ii. Data transformation: The function uses a reducer with the input 'values' object to go through each property in the 'values' object. If a match is found, a new object with the schema ('condition-field-name', 'condition-field-type', 'condition-values') is pushed onto an array 'acc'

```
const parameters = Object.entries(values).reduce((acc, [key, value]) => {
    if (fieldMapping[key]) {
        const { "field-name": fieldName, "condition-type": conditionType } = fieldMapping[key];
        acc.push({
            "condition-field-name": fieldName,
            "condition-field-type": conditionType,
            "condition-values": value,
        });
    }
    return acc;
}, []);
```

Figure 33 Input data transformation code

iii. Data serialization: The function returns a new object containing a serialized parameters array and the title, description, type, dimension from the initial object

```
return {
    title: values.title,
    description: values.description,
    type: values.type,
    dimension: values.dimension,
    parameters: JSON.stringify(parameters),
};
```

Figure 34 Input data serialization code

4. ***Check mode:*** Check if the user is in the "create" or "edit" mode
    a. If 'create' move to **step 5** "Create mode"
    b. If 'edit' move to **step 6** "Edit mode"

5. ***Create mode:*** In create mode the ***appendCreatePostVisualization*** action is dispatched with the visualization data as a payload.
    a. ***Initialize visualizations:*** The ***initializeVisualizations()*** function is called to set up an initial state for the visualizations in the localStorage (If it does not already exist). The function ensures that an item named visualizations exists in the local storage and it is an array stored as a string. If the item does not exist, it will be created and initialized as an empty array.

```
export const initializeVisualizations = () => {
    if (!localStorage.getItem('visualizations')) {
        localStorage.setItem('visualizations', JSON.stringify([]));
    }
};
```

Figure 35 InitializeVisualizations() method code

b. ***Generate unique index:*** A unique index is generated for the visualization in the create mode. The ***generateUniqueIndex()*** function is used to create a unique identifier for each visualization data object stored in the local storage. It guarantees that every new visualization will have a unique index number. This unique index is important for identifying and manipulating specific visualizations.

```
export function generateUniqueIndex() {
    const currentVisualizations = getVisualizationsFromLocalStorage();
    if (currentVisualizations.length === 0) {
        return 1;
    }
    const maxIndex = currentVisualizations.reduce((max, visualization) => Math.max(max, visualization.index), 0);
    return maxIndex + 1;
}
```

Figure 36 GenerateUniqueIndex() method code

c. ***Append visualization to local storage:*** The new visualization is added to the local storage. The ***appendVisualizationToLocalStorage()*** function is called to add a new visualization to the list of existing visualizations in the local storage.

```
export function appendVisualizationToLocalStorage(visualization) {
    const currentVisualizations = getVisualizationsFromLocalStorage();
    const updatedVisualizations = [...currentVisualizations, visualization];
    localStorage.setItem(VISUALIZATIONS_KEY, JSON.stringify(updatedVisualizations));
}
```

Figure 37 AppendVisualizationToLocalStorage() method code

d. ***Update redux state:*** The 'postToCreate' state in Redux is updated with the new visualisation. **Proceed to step 7**

6. ***Edit mode:*** In edit mode the ***appendEditPostVisualization*** action is dispatched with the visualization data as a payload
    a. ***Initialize edit visualizations:*** The ***initializeEditVisualizations()*** function is called to set up an initial state for the edit visualizations in the localStorage (If it does not already exist). The function ensures that an item named edit_visualizations exist in the local storage and it is an array stored as a string. If the item does not exist, it will be created and intialized as an empty array.

```
export const initializeEditVisualizations = () => {
    if (!localStorage.getItem(EDIT_VISUALIZATIONS_KEY)) {
        localStorage.setItem(EDIT_VISUALIZATIONS_KEY, JSON.stringify([]));
    }
};
```

Figure 38 InitializeEditVisualizations() method code

b. ***Generate unique edit index:*** A unique index is generated for the visualization in the create mode. The ***generateUniqueEditIndex()*** function is called to create a unique identifier for each edit visualization data object stored in the local storage. It guarantees that every new visualization will have a unique index number. This unique index is important for identifying and manipulating specific edit visualizations.

```
export function generateUniqueEditIndex() {
    const currentVisualizations = getEditVisualizationsFromLocalStorage();
    if (currentVisualizations.length === 0) {
        return 1;
    }
    const maxIndex = currentVisualizations.reduce((max, visualization) => Math.max(max, visualization.index_edit), 0);
    return maxIndex + 1;
}
```

Figure 39 GenerateUniqueEditIndex() method

c. ***Append edit visualization to local storage:*** The new visualization is added to the local storage. The ***appendEditVisualizationToLocalStorage()*** function is used to add a new visualization to the list of existing visualizations in the local storage.

```
export function appendEditVisualizationToLocalStorage(visualization) {
    const currentVisualizations = getEditVisualizationsFromLocalStorage();
    const updatedVisualizations = [...currentVisualizations, visualization];
    localStorage.setItem(EDIT_VISUALIZATIONS_KEY, JSON.stringify(updatedVisualizations));
}
```

Figure 40 AppendEditVisualizationToLocalStorage() method code

d. ***Update Redux state:*** The 'postToEdit' state in Redux is updated with the new visualisation. **Proceed to step 7**

7. ***Fetch visualization data:*** A POST request is made to the '/visualization/data' endpoint of the Flask server with the visualization data as the payload.

8. ***Process visualization data:*** the post method is part of the API and handles the HTTP POST request sent. It receives a JSON payload containing the visualization data. The method calls several functions within the *query_engine.py* file to parse the request, query the social care dataset and prepare the data to be used in the visualization process. Here is a step-by-step description of the process
    a. The method receives a request with a JSON body and stores it in the *input_data* variable

```
# Get the input data from the request body
input_data = request.get_json()
```

Figure 41 Storing request data code

b. The method checks whether the *'parameters'* key is present. If it is not it returns a 400 error status code with a message about the missing parameters key

c. The parameters string is parsed into a Python dictionary using the ***parse_input_string()*** function. The parsing involves converting the string to a valid JSON string and loading it as a Python object

```python
def parse_input_string(parameters_string):
    # Replace single quotes with double quotes to ensure valid JSON
    parameters_string = parameters_string.replace("'", '"')
    # Parse the JSON string
    parameters_list = json.loads(parameters_string)
    # Initialize an empty dictionary for input_dict
    input_dict = {}
    # Iterate through the list of parameters
    for parameter in parameters_list:
        field_name = parameter['condition-field-name']
        field_type = parameter['condition-field-type']
        field_values = parameter['condition-values']
        # Add the field name to the input_dict if not already present
        if field_name not in input_dict:
            input_dict[field_name] = []
        # Append the parameter to the input_dict
        input_dict[field_name].append({
            'condition-type': field_type,
            'condition-values': field_values
        })
    return input_dict
```

Figure 42 parse_input_string() function code

d. The ***get_bigquery_data()*** function is called with the *dimension* and *parsed_input* as arguments. This function connects to the Google bigquery service through the dataset parameters provided and retrieves the data based on the SQL query generated from the passed parameters through the ***generate_query_string()*** function. The returned data is a list of dictionaries each representing a row within the query result

```python
def get_bigquery_data(dimension, input_dict):
    query_string = generate_query_string(dimension, input_dict)
    print('---------------Query string--------------------')
    print(query_string)
    print('-----------------------------------')
    query_job = client.query(query_string)
    results = query_job.result()
    data = []
    for row in results:
        data.append({dimension: row[dimension], "head_count": row["head_count"]})
    return data
```

Figure 43 Get_bigquery_data() function code

```python
def generate_query_string(dimension, input_dict):
    where_conditions = []
    for field, conditions in input_dict.items():
        for condition in conditions:
            if condition['condition-type'] == 'multi-select-text' or condition['condition-type'] == 'multi-select-number':
                # Check if None is in condition-values and remove it
                null_condition = None in condition['condition-values']
                if null_condition:
                    condition['condition-values'].remove(None)
                # If there are any other conditions left, construct the IN statement
                if condition['condition-values']:
                    if condition['condition-type'] == 'multi-select-text':
                        values = ', '.join([f"'{v}'" for v in condition['condition-values']])
                    else: # 'multi-select-number'
                        values = ', '.join([f"{v}" for v in condition['condition-values']])
                    in_condition = f"{field} IN ({values})"
                    # If null_condition is true, construct the IS NULL OR IN statement
                    if null_condition:
                        in_condition = f"({field} IS NULL OR {in_condition})"
                    where_conditions.append(in_condition)
                # If there are no other conditions left, but null_condition is true, construct the IS NULL statement
                elif null_condition:
                    where_conditions.append(f"{field} IS NULL")
            elif condition['condition-type'] == 'range-slider':
                min_value, max_value = sorted(condition['condition-values'])
                where_conditions.append(f"{field} BETWEEN {min_value} AND {max_value}")
            # Add more condition types as needed
    where_clause = ''
    if where_conditions:
        where_clause = f"WHERE {' AND '.join(where_conditions)}"
    query = f"""
    SELECT {dimension}, COUNT(distinct person_id) as head_count
    FROM `{table_id}`
    {where_clause}
    GROUP BY {dimension}
    ORDER BY head_count DESC
    LIMIT 10
    """
    return query
```

Figure 44 Generate_query_string() function code

e. The function ***extract_labels_and_data()*** is called with the returned data and dimension as arguments. This function processes the data list and seperates the labels (dimensions) and the data values (counts)

```python
def extract_labels_and_data(data, dimension):
    labels = [entry[dimension] for entry in data]
    data = [entry["head_count"] for entry in data]
    return labels, data
```

Figure 45 Extract_labels_an_data() function code

9. ***Render visualization:*** The received data is used to render the appropriate chart (bar chart/ pie chart) on the frontend

```
useEffect(() => {
    const fetchData = async () => {
        const response = await customFetch.post('/visualization/data', data);

        const { data: responseData, labels } = await performDataSampling(response.data);
        const adaptedChartData = labels.map((label, index) => [label, responseData[index]]);
        setChartData([['Dimension', 'Head Count'], ...adaptedChartData]);
    };
    fetchData();
}, [data]);
```

Figure 46 Handling the received data in the frontend

```
else if (type === 'pie chart')
{
    return (
        <Chart
            chartType='PieChart'
            width='100%'
            height='400px'
            data={chartData}
            options={{
                title: dimension,
                titleTextStyle: { fontSize: 18 },
                chartArea: { width: '50%' },
            }}
        />
    );
}
```

Figure 47 Pie chart visualization code

```
if (type === 'bar chart')
{
    return (
        <Chart
            chartType='BarChart'
            width='100%'
            height='400px'
            data={chartData}
            options={{
                chartArea: { width: '50%' },
                hAxis: {
                    title: 'Head count',
                    minValue: 0,
                },
                vAxis: {
                    title: dimension,
                    titleTextStyle: { fontSize: 18 },
                },
            }}
        />
    );
}
```

Figure 48 Bar chart visualization code

10. ***End:*** The process ends here if the visualization is successfully added and rendered

Deleting a visualization flowchart:

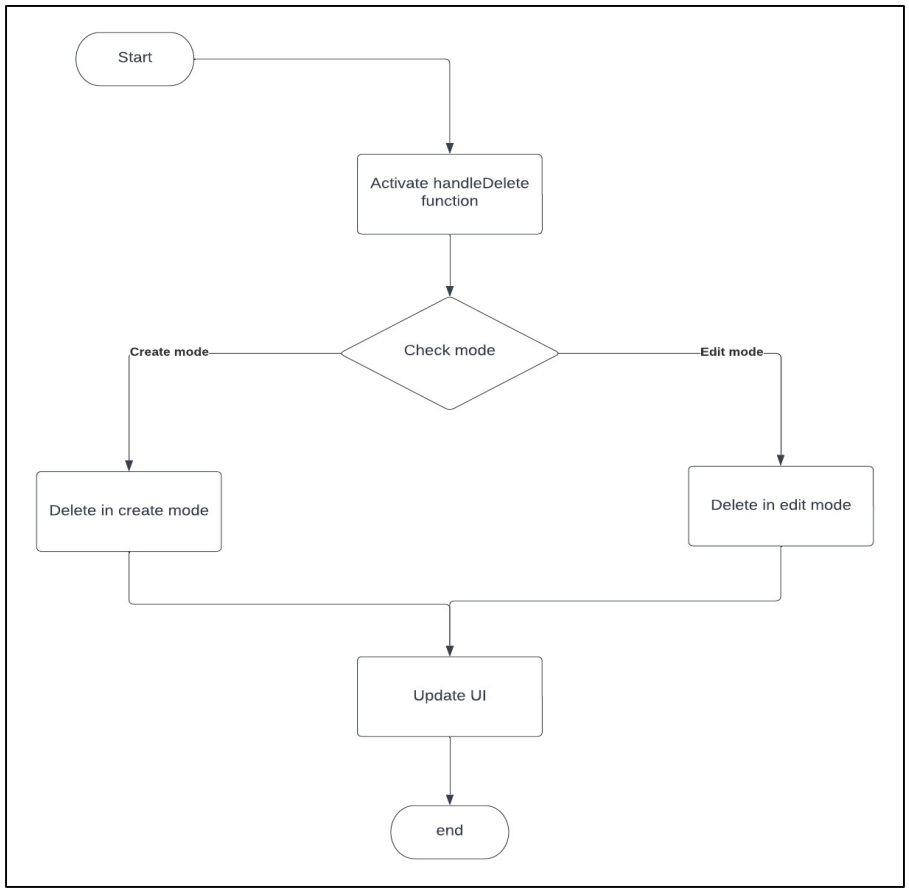


Figure 49 Flowchart for deleting a visualization

Flowchart description:

1. ***Start:*** the process starts when user clicks on delete button
2. ***Activate handleDelete function:*** the user action triggers the ***handleDelete*** function
3. ***Check mode:*** the handleDelete function check the current mode
   a. If mode is 'create' them move to **step 4**
   b. If mode is 'edit' them move to **step 5**

```
const handleDelete = () => {
    if (mode === 'create') {
        dispatch(deleteCreatePostVisualization(data));
    }
    else if (mode === 'edit') {
        dispatch(deleteEditPostVisualization(data));
    }
}
```

Figure 50 HandleDelete function code

4. ***Delete in create mode:*** the function dispatches the ***deleteCreatePostVisualization*** action with the data of the visualization to be deleted

    a. *Find visualization:* the ***deleteCreatePostVisualization*** reducer checks the state for the current visualization in the 'create' mode. It finds the index of the visualization to be deleted in the *postToCreate.visualizations* array

    b. *Check if visualization is found:* if the visualization is found (index != -1) it proceeds to the next step otherwise the process is **terminated**

    c. *Remove visualization from local storage:* the function ***RemoveVisualizationFromLocalStorage()*** is called with the visualization index which removes the visualization data from the localStorage

    d. *Remove visualization from state:* the visualization is also removed from the *postToCreate.visualizations* array

    e. *Error handeling:* If an error occurs during the removal process an error message is displayed via the *toast.error* function

```
deleteCreatePostVisualization: (state, { payload }) => {
    try {
        const index = state.postToCreate.visualizations.findIndex(v => v.index === payload.index);
        if (index !== -1) {
            removeVisualizationFromLocalStorage(payload.index);
            state.postToCreate.visualizations.splice(index, 1);
        }
    } catch (error) {
        toast.error('Error removing visualization from local storage');
    }
},
```

Figure 51 DeleteCreatePostVisaulization reducer code

```
export function removeVisualizationFromLocalStorage(index) {
    const visualizations = getVisualizationsFromLocalStorage();
    const filteredVisualizations = visualizations.filter((viz) => viz.index !== index);
    localStorage.setItem('visualizations', JSON.stringify(filteredVisualizations));
}
```

Figure 52 RemoveVisualizationFromLocalStorage function code

5. ***Delete in edit mode:*** the function dispatches the ***deleteEditPostVisualization*** action with the data of the visualization to be deleted

    a. *Find visualization:* the ***deleteEditPostVisualization*** reducer checks the state for the current visualization in the 'edit' mode. It finds the index of the visualization to be deleted in the *postInEditingt.visualizations* array

    b. *Check if visualization is found:* if the visualization is found (index != -1) it proceeds to the next step otherwise the process is **terminated**

    c. *Remove visualization from state:* the visualization is removed from the postInEditing.visualizations array in the state

d. *Update the local storage:* the edit_visualization variable in the local storage is updated with the current state of visualizations

e. *Error handeling:* If an error occurs dring the removal or update process an error message is displayed via the *toast.error* function

```
deleteEditPostVisualization: (state, { payload }) => {
    try {
        const index = state.postInEditing.visualizations.findIndex(v => v.index_edit === payload.index_edit);
        if (index !== -1) {
            console.log('Deleting visualization at index:', index);
            console.log('Before deletion:', JSON.parse(JSON.stringify(state.postInEditing.visualizations)));

            state.postInEditing.visualizations.splice(index, 1);
            setEditVisualizations(state.postInEditing.visualizations);

            console.log('After deletion:', JSON.parse(JSON.stringify(state.postInEditing.visualizations)));
        } else {
            console.warn('Visualization not found for deletion:', payload);
        }
    } catch (error) {
        toast.error('Error removing visualization from local storage in edit mode');
    }
},
```

Figure 53 DeletePostVisualization reducer code

6. ***Update UI:*** After the deletion process in the edit or create mode, the UI is updated to reflect the changes removing the visualization from the list of visualizations

7. ***End:*** the process ends with the visualization deleted from the state and the localStorage

Creating a post flowchart:

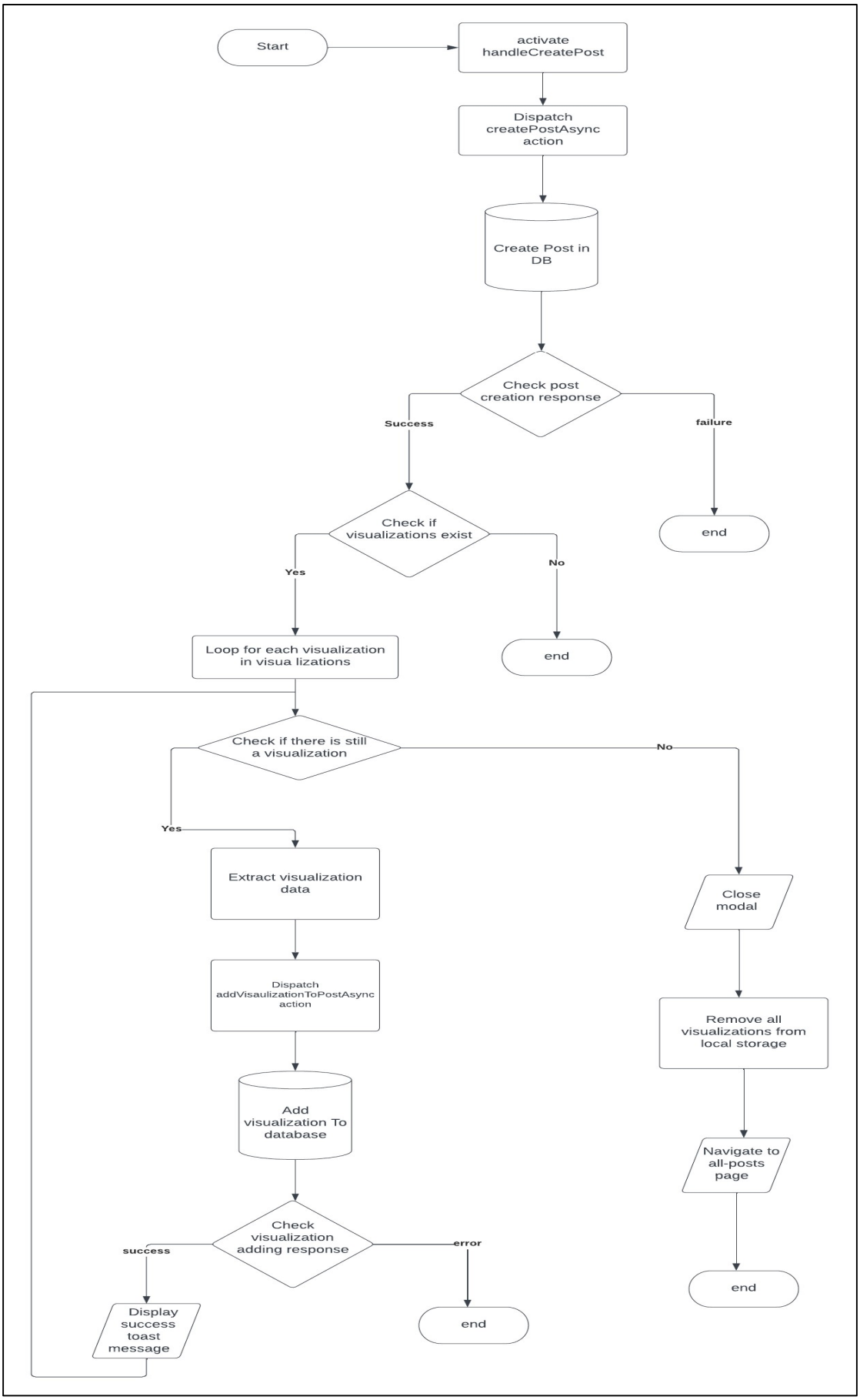


Figure 54 Flowchart for creating a post

Flowchart in depth description:

1. ***Start:*** The process starts when the user submits the create post form

2. ***Activate handleCreatePost:*** the post submission triggers the handleCreatePost function

3. ***Dispatch createPostAsync action:*** The createPostAsync action is dispatched where an HTTP POST request is sent to the '/post' endpoint alongside the post data (title, description)

```
const createPostResponse = await dispatch(createPostAsync(values));
```

```
export const createPost = async (post, thunkAPI) => {
    try {
        const access_token = getAccessTokenFromLocalStorage();
        const resp = await customFetch.post('/post', post, {
            headers: {
                Authorization: `Bearer ${access_token}`,
            },
        });
        return resp.data;
    } catch (error) {
        return thunkAPI.rejectWithValue(error.response.data.msg);
    }
};
```

Figure 55 Actions related to creating a post code

4. ***Create post in DB:*** the request is handled by the server. The post information are extracted alongside the user id and a new post is saved to the DB

```
@jwt_required()
def post(self):
    data = request.get_json()
    title = data.get('title')
    description = data.get('description')
    user_id = get_jwt_identity()
    print("user_id: ", user_id)
    post = PostModel(
        title=title,
        description=description,
        user_id=user_id
    )
    post.save()
    return post_schema.dump(post), 201
```

Figure 56 Create post in DB

5. ***Check post creation response:***
    a. If the post was created successfully move to step 6
    b. If the post creation failed end

6. ***Check if visualizations exist:*** the length of the visualizations array inside the postToCreate state is checked
    a. If the length > 0 move to step 7
    b. If the length <= 0 move to step 13

7. ***Loop over each visualization in visualizations:*** for each visualization in visualizations array

8. ***Check if there is still a visualization in visualizations:***
    a. If yes move to step 9
    b. If no move to step 13

9. ***Extract visualization data:*** the *title, description, type, dimension, parameters* are extracted from the visualization

```
const { title, description, type, dimension, parameters } = visualization;
const visualizationData = {
    title,
    description,
    type,
    dimension,
    parameters
}
```

Figure 57 Visualization parameters extracted code

10. ***Dispatch addVisualizationToPostAsync action:*** the addVisualizationToPostAsync action is dispatched where an HTTP POST request is sent to the '/visualization/post/<int:post_id>' endpoint alongside the newly created post id and the visualization data

```
const addVisualizationToPostResponse = await dispatch(addVisualizationToPostAsync({post_id, visualization: visualizationData}));
```

```
export const addVisualizationToPost = async ({post_id, visualization}, thunkAPI) => {
    try {
        const access_token = getAccessTokenFromLocalStorage();
        const resp = await customFetch.post(`/visualization/post/${post_id}`, visualization, {
            headers: {
                Authorization: `Bearer ${access_token}`,
            },
        });
        return resp.data;
    } catch (error) {
        return thunkAPI.rejectWithValue(error.response.data.msg);
    }
};
```

Figure 58 Actions related to adding a visualization to a post code

11. ***Add visualization to DB:*** the request is handled by the server. The visualization information are extracted alongside the user id and a new visualization is saved to the DB

```python
@jwt_required()
def post(self, post_id):
    data = request.get_json()
    print(data)
    title = data.get('title')
    description = data.get('description')
    type = data.get('type')
    dimension = data.get('dimension')
    parameters = data.get('parameters')
    user_id = get_jwt_identity()
    visualization = VisualizationModel (
        title=title,
        description=description,
        type=type,
        dimension = dimension,
        parameters=parameters,
        user_id=user_id,
        post_id=post_id
    )
    visualization.save()
    print(visualization)
    return visualization_schema.dump(visualization), 201
```

Figure 59 Server-side code for adding a visualization to the DB

12. ***Check visualization creation response:***
    a. If the visualization was created successfully
        i. Display success toast message
        ii. Move back to step 8
    b. If the visualization creation failed end

13. ***Close modal:*** the modal is closed

14. ***Remove all visualizations from local storage:*** the removeAllVisaulizationsFromLocalStorage() function is called to clean the local storage from any visualizations

```javascript
export function removeAllVisualizationsFromLocalStorage() {
    localStorage.removeItem('visualizations')
}
```

Figure 60 Function for removing the visualizations item from local storage

15. ***Navigate to all-posts page:*** The user is taken to the all-posts page

16. ***End:*** the process ends when the post is created and the user is in the all-posts page

Update a post flowchart:

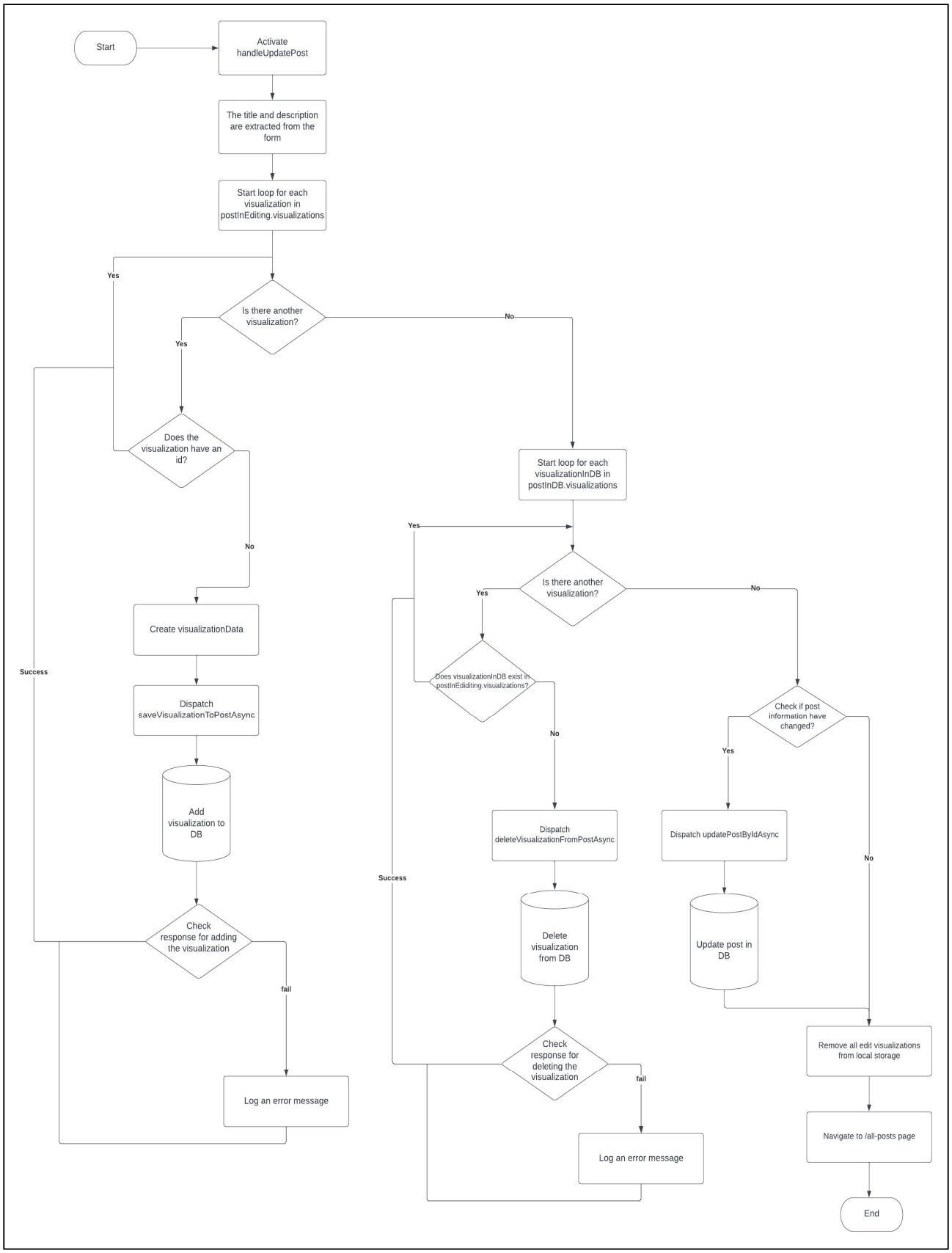


Figure 61 Update a post flowchart

Flowchart description

1. ***Start:*** The process starts when the user click on edit button in the post component

2. ***Activate handleUpdatePost:*** The button click triggers the handleUpdatePost function

3. ***Retrieve update post form input:*** the title and description entered by the user in the update post form are saved In the postData variable

```
const { title, description } = values;
const postData = {
    title,
    description
};
```

Figure 62 The post information retrieved code

4. ***Start loop for each visualization in postInEditing.visualizations***

5. ***check if there is another visualization in the array***
    a. if yes go to step 6
    b. if no go to step 12

6. ***check if the visualization has an id:*** A visualization that does not have an id is a newly added visualization by the user in the editing stage
    a. if yes go to step 5
    b. if no go to step 7

7. ***create visualizationData:*** the visualization attributes (title, description, type, dimension parameters) are retrieved into the visualizationData object

```
const {title, description, dimension, parameters, type} = visualization;
const visualizationData = {
    title,
    description,
    dimension,
    parameters,
    type
}
```

Figure 63 The visualization data retrieved code

8. ***dispatch saveVisualizationToPostAsync:*** The saveVisualizationToPostAsync action is dispatched where an HTTP POST request is sent to the /visualization/post/<int:post_id> route alongside the visualization data and the post_id

```
await dispatch(saveVisualizationToPostAsync({ post_id, visualization: visualizationData }));
```

```
export const saveVisualizationToPost = async ({post_id, visualization}, thunkAPI) => {
    try {
        const access_token = getAccessTokenFromLocalStorage();
        console.log('data to send: ', visualization);
        const response = await customFetch.post(`/visualization/post/${post_id}`, visualization, {
            headers: {
                Authorization: `Bearer ${access_token}`
            },
        });
        return response.data;
    } catch (error) {
        return thunkAPI.rejectWithValue(error.response.data.msg);
    }
}
```

Figure 64 Actions related to saving a visualization to a post code

9. ***Add visualization to DB:*** The request is handled by the server. The visualization information are extracted alongside the user id and a new visualization is saved in the DB

```
@jwt_required()
def post(self, post_id):
    data = request.get_json()
    print(data)
    title = data.get('title')
    description = data.get('description')
    type = data.get('type')
    dimension = data.get('dimension')
    parameters = data.get('parameters')
    user_id = get_jwt_identity()
    visualization = VisualizationModel (
        title=title,
        description=description,
        type=type,
        dimension = dimension,
        parameters=parameters,
        user_id=user_id,
        post_id=post_id
    )
    visualization.save()
    print(visualization)
    return visualization_schema.dump(visualization), 201
```

Figure 65 Server-side code for adding a visualization to a post

10. ***Check response from adding the visualization***
    a. If the response is successful go to step 5
    b. If response has failed go to step 11

11. ***Log an error message and go to step 5:*** an error message is logged indicating that adding the visualization to the DB operation has failed

12. ***Start loop for each visaulizationInDB in postInDB.visualizations***

13. ***Check if there is another visualization in the array***
    a. If yes go to step 14
    b. If no go to step 19

14. ***Check if visualizationInDB exists in postInEditing.visualizations:*** If the visualizationInDB does not exist in the postInEditing.visualizations array it means that it has been deleted by the user
    a. If yes go to step 13
    b. If no go to step 15

```
for (const visualizationInDB of postInDB.visualizations) {
    if (!postInEditing.visualizations.some(v => v.index_edit === visualizationInDB.index_edit)) {
```

Figure 66 checking if the visualization in the DB exists in the postInEditing visualizations array code

15. ***Dispatch deleteVisualizationFromPostAsync:*** the deleteVisualizationFromPostAsync action is dispatched where an HTTP DELETE request is sent to the /visualization/<int:visualization_id> alongside the visualization_id

```
await dispatch(deleteVisualizationFromPostAsync(visualization_id));
```

```
export const deleteVisualizationFromPost = async (visualization_id, thunkAPI) => {
    try {
        const access_token = getAccessTokenFromLocalStorage();
        const resp = await customFetch.delete(`/visualization/${visualization_id}`, {
            headers: {
                Authorization: `Bearer ${access_token}`,
            },
        });
        return resp.data;
    }

    catch (error) {
        return thunkAPI.rejectWithValue(error.response.data.msg);
    }
}
```

Figure 67 Actions related to deleting a visualization from a post code

16. ***Delete visualization from DB:*** The DELETE request is handled by the server. The visualization is deleted if it was found in the DB

```
@jwt_required()
def delete(self, visualization_id):
    visualization = VisualizationModel.find_by_id(visualization_id)
    if visualization is None:
        return {'message': gettext("visualization_not_found")}, 404
    if visualization.user_id != get_jwt_identity():
        return {'message': gettext("visualization_not_author")}, 403
    visualization.delete()
    return {'message': gettext("visualization_deleted")}, 200
```

Figure 68 Server-side code for delete a visualization from the DB

17. ***Check response for deleting the visualization***
    a. If the response is successful go to step 13
    b. If the response has failed got to step 18

18. ***Log an error message:*** an error message is logged indicating that the deletion operation has failed

19. ***Check if post information have changed:*** Check if the new title and description entered by the user are different from the original post title and description
    a. If yes go to step 20
    b. If no go to step 22

20. ***Dispatch updatePostByIdAsync:*** the updatePostByIdAsync action is dispatched where an HTTP PUT request is sent to /post/post_id alongside the post_id

```
await dispatch(updatePostByIdAsync({ post_id, post: postData }));
```

```
export const updatePost = async ({post_id, post}, thunkAPI) => {
    try {
        const access_token = getAccessTokenFromLocalStorage();
        const resp = await customFetch.put(`/post/${post_id}`, post, {
            headers: {
                Authorization: `Bearer ${access_token}`,
            },
        });
        return resp.data;
    } catch (error) {
        return thunkAPI.rejectWithValue(error.response.data.msg);
    }
}
```

Figure 69 Actions related to updating a post code

21. ***Update post in DB:*** The PUT request is hadled by the server. The post is updated in the DB

```
# update a post (LOGGED IN && AUTHOR)
@jwt_required()
def put(self, post_id):
    post = PostModel.find_by_id(post_id)
    if post is None:
        return {'message': gettext("post_not_found")}, 404
    if post.user_id != get_jwt_identity():
        print("post: ", post._id)
        return {'message': gettext("post_not_author")}, 403
    data = request.get_json()
    post.title = data['title']
    post.description = data['description']
    post.save()
    return post_schema.dump(post), 200
```

Figure 70 Server-side code for updating a post in the DB

22. ***Remove all edit visualizations from local storage:*** the removeAllEditPostVisualizations() function is called to remove the edit visualizations from the local storage

```
export function removeAllEditVisualizationsFromLocalStorage() {
    localStorage.removeItem(EDIT_VISUALIZATIONS_KEY)
}
```

Figure 71 Function called to remove the edit_visualizations item from the local storage

23. ***Navigate to /all-posts page:***

24. ***End:*** the process ends when the post and its related visualizations have successfully been updated

Viewing a post flowchart:

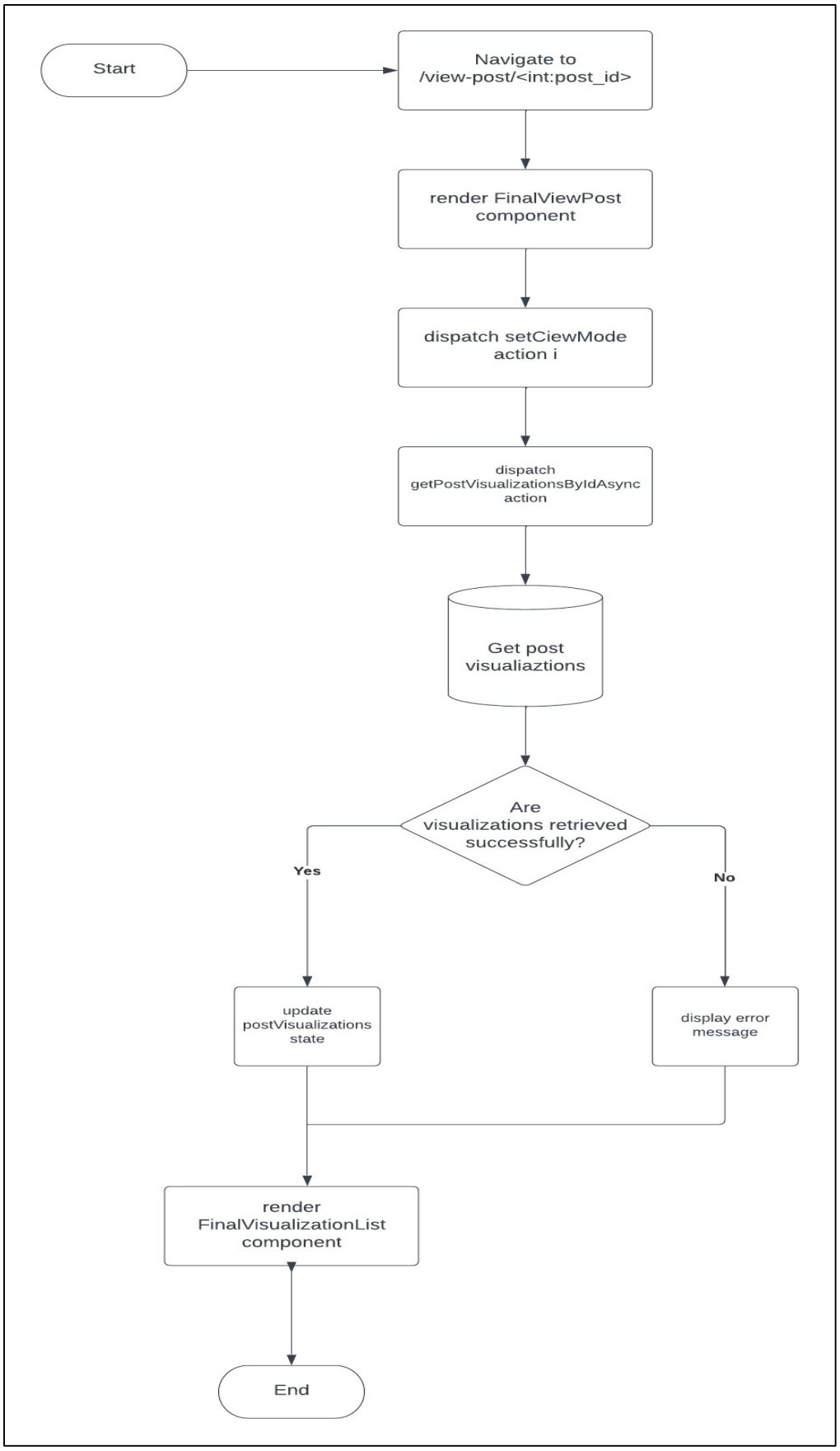


Figure 72 View a post flowchart

Flowchart description:

1. ***Start:*** The process starts when the user clicks on view visualizations
2. ***Navigate to /view-post/<int:post_id>***
3. ***Render FinalViewPostComponent:*** the FinalViewPostComponent is rendered on the page
4. ***Dispatch setViewMode action:*** the user mode is set to 'view'
5. ***Dispatch getPostVisualizationsByIdAsync action:*** an HTTP GET request is sent to the server at the endpoint /post/<int:post_id>/visualizations alongside the post_id

```
dispatch(getPostVisualizationsByIdAsync(post_id))
```

```
export const getPostVisualizationsById = async (post_id, thunkAPI) => {
    try {
        const access_token = getAccessTokenFromLocalStorage();
        const resp = await customFetch.get(`/post/${post_id}/visualizations`, {
            headers: {
                Authorization: `Bearer ${access_token}`,
            },
        });
        return resp.data;
    } catch (error) {
        return thunkAPI.rejectWithValue(error.response.data.msg);
    }
};
```

Figure 73 Actions related to getting a post by id code

6. ***Get post visualizations:*** the request is handled by the server. The post visualizations are retrieved

```
@jwt_required()
def get(self, post_id):
    post = PostModel.find_by_id(post_id)
    if post is None:
        return {'message': gettext("post_not_found")}, 404
    visualizations = post.visualizations
    return visualization_list_schema.dump(visualizations), 200
```

Figure 74 Server-side code for finding a post by id

7. ***Check if the post visualizations were retrieved successfully:***
    a. If no proceed to step 8
    b. If yes proceed to step 7
8. ***Update postVisualizations state:*** the visualizations retrieved are pushed onto the postVisualizations array

```
[getPostVisualizationsByIdAsync.fulfilled]: (state, { payload }) => {
    state.postVisualizationsLoading = false;
    state.postVisualizations = payload;
    toast.success("All post visualizaitons fetched successfully from the DB");
},
```

Figure 75 Code for the successful response for fetching the post by id

9. ***Display error message:*** an error message is displayed in case the fetching operation fails

```
[getPostVisualizationsByIdAsync.rejected]: (state, { payload }) => {
    // Payload is seful only when the rejected request has a response
    state.postVisualizationsLoading = false;
    toast.error("Failed to fetch the post visualizations from DB");
}
```

Figure 76 Error message display code for the rejected response

10. ***Render finalVisualizationsListComponent:*** the finalVisualizationListComponent is renderd with all all the visualizations retrieved from the database

```
mode === 'view' &&
(
    <>
        <VisualizationGrid>
            {viewVisualizationListIsEmpty ? (
                <>
                    <CenteredSpin>
                        <Spin size="large" tip="loading visualizations..." />
                    </CenteredSpin>
                </>
            ) : (
                <>
                {viewVisualizationList.map((visualization) => (
                    <FinalVisualization
                        key={visualization.id}
                        data={visualization}
                    />
                ))}
                </>
            )}
        </VisualizationGrid>
    </>
)
```

Figure 77 Rendering the visualization list component

11. ***End:***

4.4 Testing

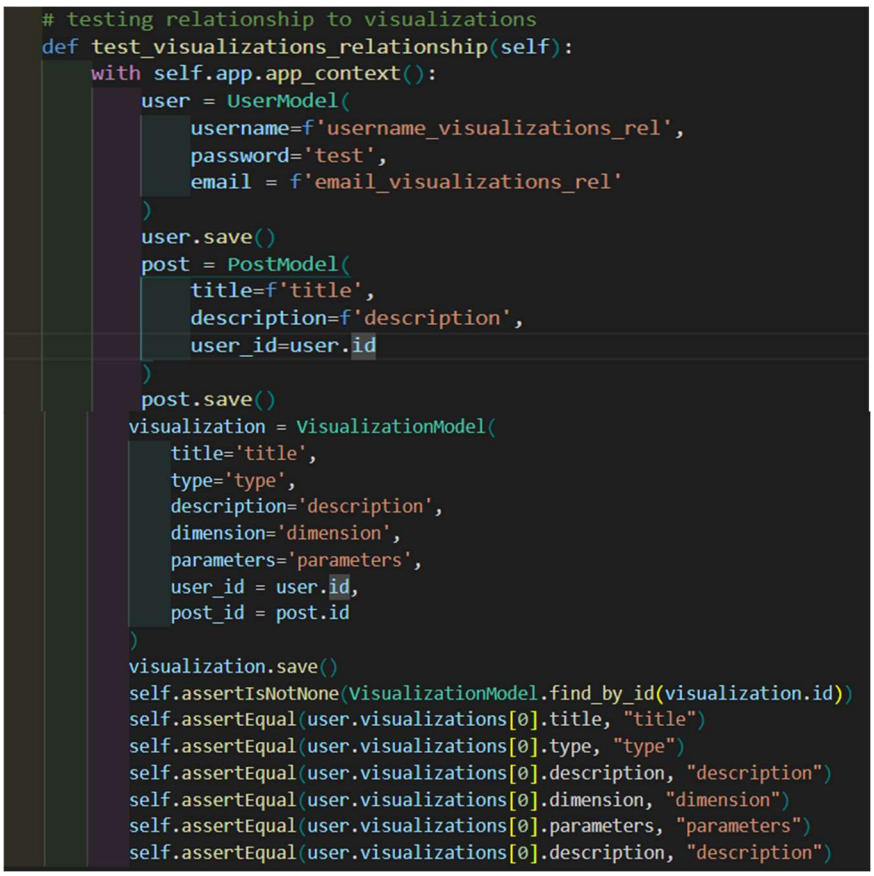
In the next section of the report, I will mention the different types of testing applied in the project. These include integration, system, and unit testing. Each of these testing methodologies contributed to maintaining the code's integrity and reliability in different ways.

4.4.1 Integration testing

Integration testing is particularly important for muti-tiered software architectures like the one employed in the server-side code consisting of the Model, Controller, and View layers. This technique uncovers potential interaction and data transfer issues early on in the development process.

Here, I used integration testing to verify the interaction between the application's main modules: User, Post, and Visualization. This helped ensure that the models work as expected both individually and while interacting with each other. Thus, the application reliability is assured.

Below are code snippets of the tests applied on the User model as an example.



```
# testing relationship to visualizations
def test_visualizations_relationship(self):
    with self.app.app_context():
        user = UserModel(
            username=f'username_visualizations_rel',
            password='test',
            email = f'email_visualizations_rel'
        )
        user.save()
        post = PostModel(
            title=f'title',
            description=f'description',
            user_id=user.id
        )
        post.save()
        visualization = VisualizationModel(
            title='title',
            type='type',
            description='description',
            dimension='dimension',
            parameters='parameters',
            user_id = user.id,
            post_id = post.id
        )
        visualization.save()
        self.assertIsNotNone(VisualizationModel.find_by_id(visualization.id))
        self.assertEqual(user.visualizations[0].title, "title")
        self.assertEqual(user.visualizations[0].type, "type")
        self.assertEqual(user.visualizations[0].description, "description")
        self.assertEqual(user.visualizations[0].dimension, "dimension")
        self.assertEqual(user.visualizations[0].parameters, "parameters")
        self.assertEqual(user.visualizations[0].description, "description")
```

Figure 78 Testing relationship between the User model and the Visualization model

```python
class TestUserIntegrationModel(BaseTest):
    def test_crud(self):
        with self.app.app_context():
            user = UserModel(
                username=f'username_crud',
                password='test',
                email = f'email_crud'
            )
            self.assertIsNone(UserModel.find_by_id(user.id))
            user.save()
            self.assertIsNotNone(UserModel.find_by_id(user.id))
            user.delete()
            self.assertIsNone(UserModel.find_by_id(user.id))
```

Figure 79 Testing CRUD operations on the User Model

```python
# testing relationship to posts
def test_posts_relationship(self):
    with self.app.app_context():
        user = UserModel(
            username=f'username_posts_rel',
            password='test',
            email = f'email_posts_rel'
        )
        user.save()
        post = PostModel(
            title=f'title',
            description=f'description',
            user_id=user.id
        )
        post.save()
        self.assertIsNotNone(PostModel.find_by_id(post.id))
        self.assertEqual(user.posts[0].title, "title")
        self.assertEqual(user.posts[0].description, "description")
```

Figure 80 Testing relationship between User model and Post model

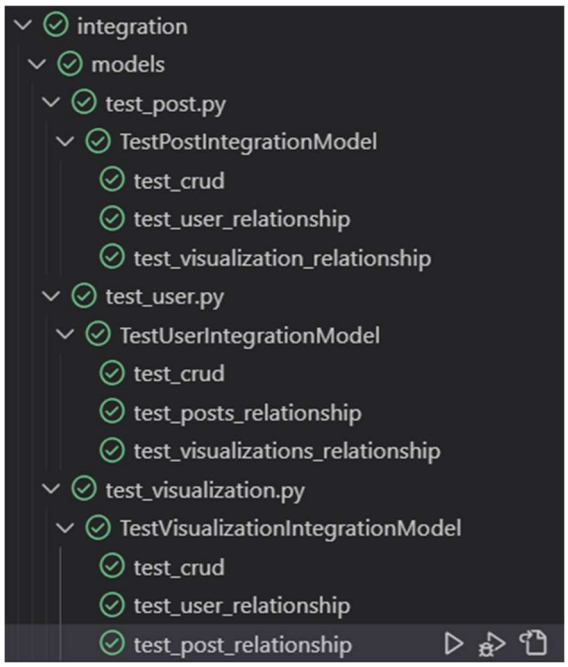


Figure 81 Integration testing output

4.4.2 System testing

In my project, system testing played an integral role in verifying the functionality of the API. Comprehensive system tests were performed on the major components of the API including the three main Models in the application: User, Post, and Visualization Models.

It is important to note that the system tests were designed to target the key endpoints used within the project. For instance, in the User Model, tests were performed to validate the user registration, login and handling duplication registrations. This excludes a few endpoints I created as admin only access (deleting all users, retrieving all users). The code snippets below show a few of the tests applied on the User related functionalities.

```
def test_register_duplicate_user(self):
    with self.client as client:
        with self.app.app_context():
            client.post('/register', json={
                "username": "test_register",
                "email": "email_register",
                "password": "1234"
            })
            response = client.post('/register', json={
                "username": "test_register",
                "email": "email_register",
                "password": "1234"
            })
            self.assertEqual(response.status_code, 400)
            self.assertDictEqual({'message': gettext("username_or_email_exists")}, json.loads(response.data))
```

Figure 82 Testing registering a duplicate user

```
class TestUserSystem(BaseTest):
    def test_register_user(self):
        with self.client as client:
            with self.app.app_context():
                response = client.post('/register', json={
                    "username": "test_new",
                    "email": "email_new",
                    "password": "1234"
                })
                self.assertEqual(response.status_code, 201)
                self.assertIsNotNone(UserModel.find_by_username('test_new'))
```

Figure 83 Testing user registration

```
def test_register_and_login_user(self):
    with self.client as client:
        with self.app.app_context():
            client.post('/register', json={
                "username": "test_login",
                "email": "email",
                "password": "1234"
            })
            response = client.post('/login', json={
                "username": "test_login",
                "password": "1234"
            })
            self.assertIn('access_token', json.loads(response.data).keys())
```

Figure 84 Testing user login

A similar strategy was applied to the Post and Visualization Models to ensure that the main operations were properly tested.

While focusing on the endpoints actively used in the application might seem limited, it allowed me to channel my resources towards the most vital parts of the system ensuring the robustness of the features that the user will interact with.

These system tests facilitated the smooth functioning of the server-side code.

Figure 85 System testing output

4.4.3 Unit testing

Unit testing was fundamental in ensuring the correctness of some of the main utility functions of my server-side code. This type of testing which emphasizes testing separate units of code was important in verifying each code snippet included in the utility folder functioned as expected. In the context of my project, unit testing was applied to the query_engine.py and query_dataset.py files.

For the query engine, I applied a series of tests on the parse input string, generate query string, get bigquery data, and extract labels and data. Mocking was used to simulate the expected behaviour of the BigQuery client's methods since they were external dependencies.

For instance, in the test_get_bigquery_data function, the BigQuery client.query() method was mocked to return predefined results. This allowed me to effectively verify the function

functionality without involving the BigQuery service in the process. The same strategy was consistently applied across other unit tests to ensure the accuracy of the outcomes.

```python
def test_get_bigquery_data(self):
    dimension = "field1"
    input_dict = {
        'field1': [
            {
                'condition-type': 'multi-select-text',
                'condition-values': ['value1', 'value2']
            }
        ]
    }
    expected_output = [
        {'field1': 'value1', 'head_count': 10},
        {'field1': 'value2', 'head_count': 5}
    ]
    # Mock the BigQuery client.query() method and its result() method
    with unittest.mock.patch('resources.utils.query_engine.client.query') as mock_query:
        # Replace the mocked result with your expected BigQuery result
        mock_query.return_value.result.return_value = [
            {'field1': 'value1', 'head_count': 10},
            {'field1': 'value2', 'head_count': 5},
        ]

        result = get_bigquery_data(dimension, input_dict)
        self.assertEqual(result, expected_output)
```

Figure 86 Testing the get biquery data function

The unit tests were beneficial in maintaining the code integrity, especially when modifications were applied. They ensured the new changes do not disrupt the expected behaviour of the utility functions used.

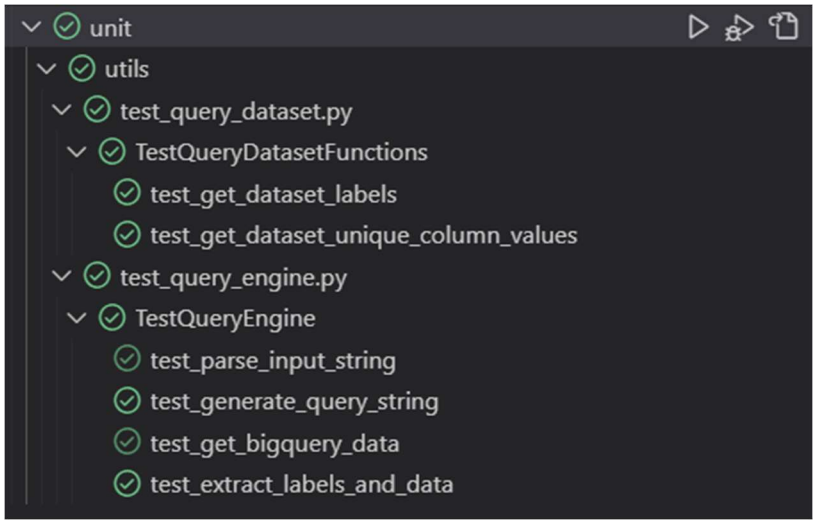


Figure 87 Unit testing output

5. Results and discussions

This section includes the key accomplishments attained during the project as well as the limitations of the solution. In addition, it provides a detailed perspective on the Legal, social, ethical, and economical aspects of the project

5.1 Main achievements

Upon completion of the development phase, two significant accomplishments were realized.

The initial achievement involved the creation of a refined dataset, which connects the most relevant characteristics within the social care datasets in line with the stakeholder needs.

The second major accomplishment was the development of a comprehensive data visualization web application. This interactive tool simplifies the process of exploring the generated social care dataset. Most importantly, It satisfies all the functional and non-functional prerequisites outlined in the requirements and analysis chapter.

Other achievements worth mentioning include the alignment with the main learning outcomes of the software engineering programme. In fact, designing and implementing the solution required an extensive practical knowledge in different areas mainly due to the complexity of the prototype.

In the data analysis phase of the project, I used Google BigQuery to filter, extract, and link various datasets in the social care space I had access to. For instance, I used BigQuery's powerful SQL-like language to manipulate and retrieve data across the social care contacts dataset as well as the GP visits dataset. These operations contributed to refining my SQL skills with respect to dealing with large and complex datasets. Alongside BigQuery, I used Looker studio to effectively present the generated dataset. Throughout the data exploration phase, I created interactive dashboards to facilitate a better understanding of the patterns and trends in the social care data. This helped me understand data-driven decision-making processes which will be an invaluable addition to my skillset.

Setting up the development environment required an understanding of the VS code IDE. This knowledge was vital for efficient development. A critical component of the setup was the usage of Flask-Migrate. This utility facilitated the seamless transition between different databases namely PostgeSQL and SQLite for testing purposes. This flexibility was achieved through the command-based operations included within Flask-Migrate.

The backend layer of the solution was built independently, and it incorporates a REST API to handle the data. During the development stage, I used Postman, a tool that facilitated constructing and sending API calls to the server. This allowed me to evaluate the results returned and test the system's endpoints manually.

In addition to the mentioned development tools, an important aspect of my accomplishments include the application of certain programming languages as well as the frameworks associated with them.

For the server-side coding, I employed Python in conjunction with the Flask framework. This combination facilitated rapid prototype creation for my REST API. My expertise in the

Python language significantly increased as I had to build an API that accurately addressed all the system requirements. Since Flask is a relatively new framework meaning community support is more limited, I had to dive into its official documentation which was an essential part of the process. This knowledge and hands-on experience gained from the backend development is valuable in the tech industry. For instance, the adoption of Flask across different sectors and the already significant use of Python proves the relevance of this knowledge.

For the frontend development side of the project, JavaScript was the primary language employed. Prior to undertaking this project, I had a very limited experience with JavaScript. However, as the development process ended, my expertise significantly improved. Also, the React.js framework played an important role in shaping the structure of my frontend code. Amongst its key features are reusable components and router which enabled me to build a maintainable and flexible frontend solution. The reusable components ensured a consistent user interface across pages and reduced the redundancy of the code which is one of the main advantages of React.

Another integral part of the frontend development was using the Redux library, particularly the redux toolkit. This integration created a structured codebase and a predictable state management across the entire application.

These achievements align closely with industry standards which will be invaluable in my future software engineering job.

5.2 Limitations of the solution

While the developed solution is in line with the elicited requirements, it also carries limitations that need to be addressed.

A primary limitation lies within the social care dataset itself which is derived from real world data. The null values within the dataset pose a challenge particularly for future data explorations and especially for the construction of potential prediction models. For example, in the case of 'year_of_death' field, null values do hold a significance. They imply that the individual in the dataset is still alive. However, such values could potentially disrupt the analysis and should be replaced or removed from the dataset in future iterations.

In addition, the final dataset does not encapsulate the entire journey of an individual through the social care system. The data including the GP visits, which could provide deeper insights into the individuals healthcare path was excluded due to the high degree of missing values inside of it.

Furthermore, the solution's filtering capabilities could include more filters allowing for a more detailed breakdown and personalized analysis of the individuals in the social care.

Regarding the visualizations used in the web application, the solution currently contains two chart types: bar chart and pie chart. While these two charts are important for representing broad data trends, they may not be sufficient to generate a detailed visualization of the individuals in the dataset. The current visualization could potentially be expanded in future iterations to provide more nuanced views.

The query engine currently employed generates SQL queries with a constraint of 10 results to be displayed. This limitation was introduced to ensure clear visualizations and maintain an acceptable rendering speed. However, this limitation would potentially create challenges in case the visualization type is data-intensive (Sankey diagram for example). Therefore, enhancing the capacity of the query engine and the API to handle larger output results may be necessary in case more complex visualizations are introduced.

Addressing these limitations and future improvements would significantly improve the effectiveness of the solution and broaden its potential applications.

5.3 Professional, Legal, social, ethical and economic considerations

From a professional standpoint, the project followed industry-standard practices and tools as mentioned in the achievements section of the report. Proper testing such as integration, system and unit testing as well as coding standards were adhered to for maintainability and robustness. For instance, the server-side and client-side development were done separately. This approach allowed for a better separation of concerns as the backend focused on the data processing and database interactions while the frontend handled the user interactions and the UI design. The backend also employs the adapted Flask MVC pattern which is a well-established design principle in web development.

From a legal standpoint, the project complies with the data protection and privacy regulations related to the use of the Connected Yorkshire Research Database. I agreed to these regulations by filling and signing the expression of interest request form to access the dataset. No data has been saved directly from the dataset. The query engine handles the dataset querying operations according to the filter parameters set by the user. Only those parameters are saved in the external database.

This project involves a significant social impact. In fact, the interactive visualization tool would potentially enable stakeholders to better understand the trends in the social care data. This would contribute to a more informed decision-making that ultimately enhances social care services. However, it is important to note that the tool does not promote ethnicity biases or stereotypes.

From an ethical perspective, the privacy of individuals involved in the social care dataset has been carefully considered. The data obtained does not contain information that could easily identify the individuals. Instead, a unique person_id field was used to distinguish between the individuals and ensure their privacy.

Economically, the project is fundamentally a research-oriented project. The goal was to enhance insight generation from the social care dataset. It was not developed with a commercial intent or to generate profit. Therefore, it does not have economic considerations.

6. Conclusion and Further Work

The report outlines the process behind the design, implementation, and development of an interactive visualization tool for social care data. The core features of the tool including dashboard creation, modification, and deletion enable quick hypothesis generation about the data. They also facilitate understanding the major trends within the social care datasets.

Within the report, a comprehensive literature review about the relevant visual analytics field as well as the development tools was included. It also included a detailed requirements analysis section to set both functional and non-functional requirements for the system. In addition, the report outlines the step-by-step data analysis process used to create the final dataset connected to the system. Following that, the design, implementation, and testing conducted throughout the development process were covered. The report concludes with a review of the key achievements, system limitations and LSEP considerations.

The next step in this project would involve refining existing features and introducing new capabilities. The data visualizations chart types can be expanded to include more complex types, such as Sankey diagrams, and mixed line charts which would enhance the system's analytical depth.

On the data exploration and analysis side, the system can be linked to a more detailed and comprehensive dataset. This dataset should consider the inconsistencies often found in real-world data such as handling outliers or missing values. By extending and improving the current dataset, the system would be able to provide more accurate and detailed insights about the social care data.

In terms of system performance, potential improvements would include optimizing the rendering speed of the visualizations and implementing pagination for the posts and the visualizations. These changes would not only improve the user experience but also ensure a better system stability and efficiency when scaling up.

Predictive models could potentially be introduced to offer a view into future trends in the social care space. For instance, dynamic visualization techniques and animations can be utilized to illustrate these predictions enriching the storytelling aspect of the tool. Finally, the UI can be refined for a more intuitive user experience. This could involve redesigning the create-post page and the AddVisualization component in particular.

7. References

Thomas, J.J. (2005). *Illuminating the path: [the research and development agenda for visual analytics]*. Los Alamitos, Calif.: Ieee.

Keim, D. and Al, E. (2010). *Mastering the information age: solving problems with visual analytics*. Goslar: Eurographics Association.

Eirich, J., Bonart, J., Jackle, D., Sedlmair, M., Schmid, U., Fischbach, K., Schreck, T. and Bernard, J. (2022). IRVINE: A Design Study on Analyzing Correlation Patterns of Electrical Engines. *IEEE Transactions on Visualization and Computer Graphics*, 28(1), pp.11–21. doi:10.1109/tvcg.2021.3114797.

Raidou, R.G., van der Heide, U.A., Dinh, C.V., Ghobadi, G., Kallehauge, J.F., Breeuwer, M. and Vilanova, A. (2015). Visual Analytics for the Exploration of Tumor Tissue Characterization. *Computer Graphics Forum*, 34(3), pp.11–20. doi:10.1111/cgf.12613.

Andrienko, G., Andrienko, N., Chen, W., Maciejewski, R. and Zhao, Y. (2017). Visual Analytics of Mobility and Transportation: State of the Art and Further Research Directions. *IEEE Transactions on Intelligent Transportation Systems*, 18(8), pp.2232–2249. doi:10.1109/tits.2017.2683539.

Saraiya, P., North, C., Vy Lam and Duca, K.A. (2006). An Insight-Based Longitudinal Study of Visual Analytics. *IEEE Transactions on Visualization and Computer Graphics*, 12(6), pp.1511–1522. doi:10.1109/tvcg.2006.85.

Guo, Y., Guo, S., Jin, Z., Kaul, S., Gotz, D. and Cao, N. (2022). Survey on Visual Analysis of Event Sequence Data. *IEEE Transactions on Visualization and Computer Graphics*, 28(12), pp.5091–5112. doi:10.1109/tvcg.2021.3100413.

Du, F., Shneiderman, B., Plaisant, C., Malik, S. and Perer, A. (2017). Coping with Volume and Variety in Temporal Event Sequences: Strategies for Sharpening Analytic Focus. *IEEE Transactions on Visualization and Computer Graphics*, 23(6), pp.1636–1649. doi:10.1109/tvcg.2016.2539960.

Reinartz, T. (1999). *Focusing solutions for data mining: analytical studies and experimental results in real-world domains*. Berlin; New York: Springer.

W. Aha, D. and L. Bankert, R. (1995). A Comparative Evaluation of Sequential Feature Selection Algorithms. In: *Pre-proceedings of the Fifth International Workshop on Artificial Intelligence and Statistics*.

Hyunmo Kang, Getoor, L., Shneiderman, B., Bilgic, M. and Licamele, L. (2008). Interactive Entity Resolution in Relational Data: A Visual Analytic Tool and Its Evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 14(5), pp.999–1014. doi:10.1109/tvcg.2008.55.

Bigelow, A., Williams, K. and Isaacs, K.E. (2021). Guidelines For Pursuing and Revealing Data Abstractions. *IEEE Transactions on Visualization and Computer Graphics*, 27(2), pp.1503–1513. doi:10.1109/tvcg.2020.3030355.

Eick, S.G. and Karr, A.F. (2002). Visual Scalability. *Journal of Computational and Graphical Statistics*, 11(1), pp.22–43. doi:10.1198/106186002317375604.

Monroe, M. (2014). *Interactive Event Sequence Query and Transformation*.

Dimara, E., Zhang, H., Tory, M. and Franconeri, S. (2021). The Unmet Data Visualization Needs of Decision Makers within Organizations. *IEEE Transactions on Visualization and Computer Graphics*, pp.1–1. doi:10.1109/tvcg.2021.3074023.

Collins, C., Andrienko, N., Schreck, T., Yang, J., Choo, J., Engelke, U., Jena, A. and Dwyer, T. (2018). Guidance in the human–machine analytics process. *Visual Informatics*, [online] 2(3), pp.166–180. doi:10.1016/j.visinf.2018.09.003.

Sedlmair, M., Meyer, M. and Munzner, T. (2012). Design Study Methodology: Reflections from the Trenches and the Stacks. *IEEE Transactions on Visualization and Computer Graphics*, 18(12), pp.2431–2440. doi:10.1109/tvcg.2012.213.

Google Cloud. (n.d.). *Looker Studio: Business Insights Visualizations*. [online] Available at: https://cloud.google.com/looker-studio.

Association, H. (n.d.). *Government publishes plan to digitalise health and social care*. [online] www.homecareassociation.org.uk. Available at: https://www.homecareassociation.org.uk/resource/government-publishes-plan-to-digitalise-health-and-social-care.html#:~:text=By%20March%202024%2C%20the%20Government [Accessed 15 Jan. 2023].

Google Cloud. (n.d.). *Looker Business Intelligence Platform & Embedded Analytics*. [online] Available at: https://cloud.google.com/looker?_ga=2.38445604.662126753.1684605336-1343692446.1670174034#section-1 [Accessed 20 May 2023].

Tableau. (n.d.). *Our Platform*. [online] Available at: https://www.tableau.com/products/our-platform.

Passlick, J., Grützner, L., Schulz, M. and Breitner, M.H. (2023). Self-service business intelligence and analytics application scenarios: A taxonomy for differentiation. *Information Systems and e-Business Management*. doi:https://doi.org/10.1007/s10257-022-00574-3.

Connolly, T.M. and Begg, C.E. (2020). *Database systems : a practical approach to design, implementation and management*. Uttar Pradesh, India: Pearson India Education Services.

flask.palletsprojects.com. (n.d.). *Welcome to Flask — Flask Documentation (2.3.x)*. [online] Available at: https://flask.palletsprojects.com/en/2.3.x/#api-reference [Accessed 20 May 2023].

ostezer and Drake, M. (2014). *SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems*. [online] Digitalocean.com. Available at: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems.

KOMODO Digital. (2021). *9 Reasons Why React is Still Popular in 2021*. [online] Available at: https://www.komododigital.co.uk/insights/9-reasons-why-react-is-still-popular-in-2021/.

Meta Open Source (2023). *React*. [online] react.dev. Available at: https://react.dev/.

Js.org. (2015). *Redux - A predictable state container for JavaScript apps.* [online] Available at: https://redux.js.org/.

8. Appendices

The source code for my project is accessible through my university's OneDrive account. Here is the link to access and download the compressed zip folder.

Link: https://unibradfordac-my.sharepoint.com/:u:/g/personal/ibouzaia_bradford_ac_uk/Ec_MsEGz9YRDvvIF3fKxnHcBGxNN6bBfqMH98zH-jr5FiQ?e=odcdjK