POLITECNICO DI TORINO

Master's Degree in Communications and Computer Networks Engineering (Ingegneria Telematica e delle Comunicazioni)

Master's Degree Thesis

Engineering and deployment of the Personal Data Safe (P-DS) for PIMCity

Supervisors

Prof. Marco MELLIA

Prof. Martino TREVISAN

Candidate

Enrico ANNALORO

October 2021

Abstract

With the fast development of a society that bases its economy on the web, almost forcing millions and millions of users to utilize new applications, forms, games, movie platforms and more, directly from the internet, we have noticed how gathering and collecting user personal data has become of fundamental importance. This is sponsored as to be a feature that helps customize the user experience depending on the individual tastes and likes. But personal data should not be given away so easily. Most of the online services do not inform the user on how their data are used, manipulated or even distributed. Personal Information Management Systems (PIMS) try to give users control over their data, by configuring a central, safe location to store all their personal information and manage the distribution in a simple and intuitive manner, while giving feedback back to the user on the status of their stored and distributed data.

PIMCity is a European project that is implementing this technology, based on a set of different tools. One main component of PIMCity is the Personal Data Safe (PDS), a platform to securely store the personal data, with a friendly user interface which allows for easy interaction and gather of information. PDS will then interact with other PIMCity components for the management of authorizations and accessibility to the stored data. In this work I will explain and analyse the implementation of the PDS as a full-stack web application, built from a previously developed version and improved using the latest cutting edge technologies and designs.

The future of technology is deeply infused into the development of internet applications and web services in general. We believe that protecting the users from leading IT companies, which exploit the power of their platform to force the free and uncontrolled delivery of personal information, is of outermost importance, and we think that PIMCity might be a balanced and right solution to a shared pool of controlled information.

Acknowledgements

I would like to thank my supervisors Marco Mellia and Martino Trevisan for the opportunity and for the support given to me during the development and, in general, for the assistance given during these months of work.

I would also like to thank my parents and my close friends for the support and for always believing in me.

At last, to my girlfriend Coco, thank you for accompanying me during these long years and for always being there to help me in the most difficult of times.

"Thank you all."
Enrico

Table of Contents

List of Tables VI

List of Figures VII

Acronyms VIII

1 Introduction 1
- 1.1 Motivation . . . 1

2 Background 3
- 2.1 PIMCity . . . 3
  - 2.1.1 PIMS Development Kit . . . 4
- 2.2 The initial state of the project . . . 6
  - 2.2.1 Data Structure Design . . . 6
  - 2.2.2 Main Functionalities and Frontend . . . 8
- 2.3 Goals of the project . . . 8

3 P-DS architecture: Frontend and Backend 11
- 3.1 From centralized, static server-side web server to fully-flagged, distributed client-side application . . . 11
  - 3.1.1 Server-side vs. Client-side Rendering . . . 12
  - 3.1.2 Django . . . 13
  - 3.1.3 The Frontend Web APP . . . 15
- 3.2 From MongoDB to PostgreSQL . . . 24
- 3.3 Authentication and JWT . . . 28
- 3.4 Using the .yaml file to control all the functionalities of the P-DS . . . 30

4 Databuyers APIs 35
- 4.1 The interaction with the Personal Content Manager . . . 35
- 4.2 The security in the Token . . . 37
- 4.3 The development of databuyers' APIs . . . 38

5 Deployment, testing and next steps 40
5.1 Deployment 40
5.2 Testing 42
5.3 Next Steps 47
5.3.1 Security 47
5.3.2 Integration with other components 47
5.3.3 OAuth 48
5.3.4 Automation for CI/CD implementation 48
5.3.5 Additional features for the REACT APP 48

6 Conclusions 50

Bibliography 52

List of Tables

5.1 Users requesting 1000 location history records to API 43
5.2 Users requesting 1000 browsing history records to API 44
5.3 Databuyers requesting users' records to API 45

List of Figures

2.1 P-DS logical interactions view 5
2.2 P-DS initial logical structure 7
2.3 P-DS initial homepage 8
2.4 P-DS initial PersonalInformation page 9

3.1 Popularity Chart for Js Libraries 18
3.2 P-DS Homepage 19
3.3 Personal Information Page 19
3.4 Comparison of different screen-size View 20
3.5 Logical tree architecture for React App 22
3.6 Example of editable data page 23
3.7 Example of Location History Page 24
3.8 Example of Single Location Map Pop-up 25
3.9 Example of All Location Map Pop-up 25
3.10 Example of Browsing History Page 26
3.11 Schema is checked upon startup 34
3.12 Web APP reflecting changes in Schema.yaml file 34

4.1 Databuyers request workflow 36
4.2 Databuyers APIs usage 39

5.1 PIMCITY project server configuration 42
5.2 Users requesting 1000 location history records to API 44
5.3 Users requesting 1000 browsing history records to API 44
5.4 Databuyers requesting users' records to API 45

Acronyms

API

Application Programming Interface

CRUD

Create, Read, Update and Delete

CSS

Cascading Style Sheets

DOM

Document Object Model

DOS

Denial Of Service

ECDSA

Elliptic Curve Digital Signature Algorithm

EDPS

European Data Protection Supervisor

GDPR

General Data Protection Regulation

HTML

HyperText Markup Language

HTTP

Hypertext Transfer Protocol

JWT

JSON Web Token

PCM

Personal Content Manager

PDA

Personal Data Avatar

PDK

PIMS Development Kit

PDS

Personal Data Safe

PIMS

Personal Information Management Systems

RSA

Public-key cryptosystem: the acronym RSA comes from the surnames of Ron Rivest, Adi Shamir and Leonard Adleman who desribed it officially in 1977

RTT

Round Trip Time

SQL

Structured Query Language

URL

Uniform Resource Locator

VM

Virtual Machine

Chapter 1

Introduction

1.1 Motivation

Today's daily life strongly revolves around the usage of internet services: reading the news, reserving a table at a restaurant, watching movies, listening to music are all activities that are now mainly accessed on their web based counterparts compared to only 10 or 15 years ago. Moreover, due to the recent events in the public health, with the spread of the virus covid-19 throughout the globe, lots of jobs have migrated into a home remote working environment, and this means that most of our lives have now as a central focus point, the access and usage of internet applications.

To sustain a virtual ecosystem of this proportions, the amount of information and data exchanged through the internet is enormous. Just to better understand and put into perspective what just said: it is estimated that in one second are performed around 94'000 Google searches, 10'000 tweets on Twitter, 1'000 Instagram pictures posted and 124'170 GB of data exchanged [1, Internet Live Stats].

We are also well aware that in today's web usage there are few giants of the industry that hold and compete for the monopoly of the internet. We clearly have Google on one side which is constantly challenged by Amazon, Microsoft and many more. We can see the same situation on the leisure category of web applications with Netflix competing with Amazon and Disney, with Spotify competing with Tidal and Amazon. Few companies control the market, and more importantly, control their users. As most of these services require mandatory sharing of personal information, the user has no choice on if or not to share its data, he just has to.

User data is obtained and analyzed by the web companies to enable specific features that make the experience in their service more pleasant. Information

are stored in cookies or imported in the browser to modify each user experience depending on what the system thinks is better for the customer. Although most of the time, the final result seems functional and convenient, the main problem is that there is no choice. Users are forced to present their data if they want to use the service. We have seen a trend in the last years where the most profitable companies are those who manage large amount of information, benefiting from the release or the share of these information to partners and/or into advertising campaigns; yet when it comes to users, the value of the data in their hands is close to or exactly zero.

If there is such a profitable margin for companies in the market of data and information distribution, there should also be an adequate market, for the users, to avoid the free and "forced" release of valuable information, a controlled environment where each and everyone can profit and control where their data is being moved to and used for. There should be a solution that is regulated by protocols and data protection standards which, in fact, are being discussed and analyzed by EU's GDPR [2, GDPR description] following the lack of online privacy that has been developed in these last years over the internet. The proposed project PIMCity aims at solving all these issues in an innovative manner while keeping the user accessibility user-friendly and the process and analysis transparent.

Chapter 2

Background

2.1 PIMCity

Based on all the motivations cited above and much more, the EU founded the project PIMCITY, which aims to improve the methods and tools of the users in managing their personal data. PIMCity is based on the concept of PIMS - Personal Information Management Systems - which are described by the EDPS as systems that help give individuals more control over their personal data, allowing individuals to manage their personal data in secure, local or online storage systems and share them when and with whom they choose [3, EDPS].

PIMCity is creating the tools to make PIMS accessible by creating a PIMS development kit (PDK), which makes the creation of new PIMS easy and standardized. All of this will be combined with the development of transparency tags (TT) used to display to the users information about the services they access and personal data avatar (PDA) to help users control the information shared to third parties [4, PIMCity project description].
The PDK will allow for an easier, faster, controlled and cheaper method of building PIMS, thanks to the open API developed and documented, aiming at distributing the standard accross multiple realities and imprinting the privacy and protection of data as a defacto standard [5, PIMCity project proposal].

The project also has additional goals related, as the development of EASYPims which is described in the project proposal as a fully-fledged PIMS for controlling, visualising, releasing, and monetizing web and mobility data, and demonstrate how easy it is to combine off-the-shelf components from the PDK with a limited amount of ad hoc code to create fast and economically powerful real world PIMS [5, PIMCity project proposal].

PIMCity focuses on personal data platforms and targets the online web environment (including fixed and mobile, making no distinctions). It aims at helping all those startups and small companies which would, most likely, fail in the competition of data security and management, if compared to the tech giants that are established. These small companies will need a system which is easily applicable and connected to many users. This is why PIMCity aims at partnering with TELCO realities to start its system, connecting to its systems the users already established in TELCO and inserting the customers data that has already been collected. Moreover PIMCity really believes that users should be active actors of the whole platform by chosing how their personal data is disposed, receiving monetary benefits for sharing and receiving extra services on security and analysis of their information.

2.1.1 PIMS Development Kit

In this section we are going to analyze more in detail the structure of the PIMS Development Kit (PDK). The PDK is a set of tools which will aid in the creation of standardized and complex PIMS. As described in the PIMCity project proposal, the main components of PDK are divided into different categories: elements to improve data subject privacy, mechanisms for the new data economy and Novel Data Management Tools. [5, PIMCity project proposal (pg. 9)].

- **Elements to improve data subject privacy**: these include functionalities that allow the users to take informed decisions about which information to share and with whom. In this category are defined the following components : **Personal Data Safe (P-DS)**, **Personal Privacy Metrics (P-PM)**, **Personal Consent Manager (P-CM)** and **Personal Privacy Preserving Analytics (P-PPA)**

- **Mechanisms for the new data economy**: Takes care of the fundamental creation of a transparent, open and easily accessible data market. In this category are identified two fundamental components and functionalities: **Data Valuation Tools (D-VT)** and **Data Trading Engine (D-TE)**

- **Novel Data Management Tools**: Moving and manipulating user data among heterogeneous systems need to be standardized with proper metadata which will record data source, data value and the implementation of these data among the different systems. In this category are identified the following components: **Data Aggregation (DA)**, **Data portability and control (DPC)**, **Data provenance (DP)** and **Data knowledge extraction (DKE)**

In this work we will only focus on the elements to improve data subject privacy and more specifically on the **Personal Data Safe (P-DS)**, as this was the target of all the technical work done. When needed we will stop and focus on its interaction with other components.

The Personal Data Safe (P-DS) is the means to store personal data in a controlled form. It implements a secure repository for the user's personal information such as navigation history, contacts, preferences, personal information, etc. It gives the possibility to handle them through REST-based APIs or a web interface. Thanks to the REST APIs, the P-DS can be accessed also by other components of the PDK. The architecture of the P-DS interactions is depicted in the figure below.

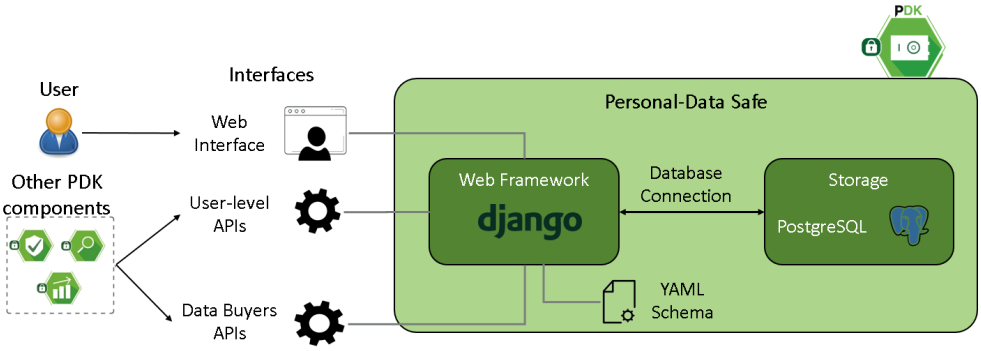


Figure 2.1: P-DS logical interactions view

We will consider the details of the implementation in the following chapters, for now we just want to concentrate on the logical role that the P-DS holds and the input/output connection it holds. As shown in Figure 2.1, our designed P-DS will interact either directly with the user through a web interface or indirectly, through other PDK components, through the defined APIs. The web interface is managed and delivered from one of the project's servers and it uses the same APIs that the P-DS components use. Both customers and data-buyers, which are those entities or companies that wants to access the data of the users, will have specific ad-hoc APIs to permit scalable deployment of the whole application system. We will discuss about this later as well.

In the end, as stated above, the P-DS will store, display and distribute the data of the users, it will allow for manipulation, modification, deletion and insertion of data both manually and automatically, which is one of the main targets of the project as giving the users the control and a detailed view of their information.

2.2 The initial state of the project

Before beginning to describe the work that has been done in these months, it is important to define and describe the state of the P-DS. In this section, we will briefly consider the outline, the architecture, the technologies and designs implemented in the P-DS. If interested, it is possible to find all these information, much more in detail, in the thesis elaborated for the description of the initial P-DS implementation [6, Personal Data Safe: a flexible storage system for personal data].

In the beginning, the P-DS was delivered as a Django Web Application that interacted with a MongoDB database. The choice of technologies was mainly based on flexibility and ease of implementation: Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It takes care of much of the hassle of web development, to let developers concentrate on creating websites and APIs without having to build everything from scratch. It's free and open source [7, Django official website]. Its main qualities are to be fast, secure and scalable. Moreover, being based on Python there are lots of free open-source libraries to implement a variety of features.

MongoDB instead is an object-oriented, simple, dynamic, and scalable NoSQL database. It is based on the NoSQL document store model. It gives the developer a flexible choice of the model of data, which for the beginning of the project was a positive aspect. We will discuss later, when discussing the changes made to the project, why it has not been kept in the new design of the P-DS [8, MongoDB official website].

Lastly, the web view was served through back-end generated static HTML files which were styled using a framework known as Bootstrap. Again we considered this a good choice for the ease of implementation it brought as Bootstrap is defined as a free and open-source CSS framework directed at responsive, mobile-first frontend web development which was developed by TWITTER. It contains CSS and JavaScript based design templates for typography, forms, buttons, navigation, and other interface components [9, Bootstrap official website].

2.2.1 Data Structure Design

This section describes how the data have been structured within the P-DS. It is of fundamental importance as, throughout our development of the project we kept this structure as is and try to automate the whole new application from just modifying one single file, as will be explained in later sections.

The central and focal point of the P-DS data structure resides in the **Schema**. The Schema as a YAML file that contains the possible types of information that can be stored in the P-DS, listing the possible fields and the logical group for each type. The schema has the primary goal to control the data that can be inserted on the Personal Data Safe: in this way it prevents the user from inserting unstructured information, that may be too complex to handle and process, or erroneous data. On the other side, the schema can be easily changed or expanded, providing the flexibility required by different project goals. The schema can be consulted by users, but its content is defined and managed by the system administrator.

The whole database is divided into two macro categories, which are identified as **Classes**, defined at source-code-level to represent the data domain: the **User class**, that represents P-DS users, and the **Personal Information class**, which describes the principal characteristics of each P-DS entry.

Lastly, some REST APIs were defined too distribute the data. These APIs were on an early stage, therefore we will reserve a whole section to it later.

We report the initial logical design of the P-DS as described in [9, Bootstrap official website]

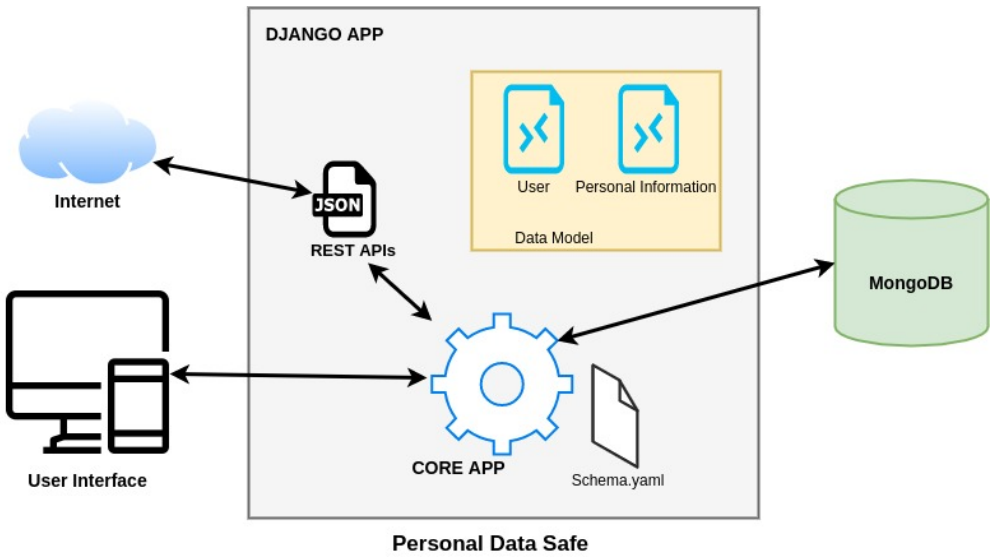


Figure 2.2: P-DS initial logical structure

2.2.2 Main Functionalities and Frontend

The main functionalities of the P-DS have not been touched much during this work, they have been reworked and improved but the main concept has remained the same. With the *Schema.yaml* file set-up, the backend running and the database functioning, it was possible, through the web interface, performing the following actions:

- Log-in: Log-in procedure was performed by inserting username and password as the O-Auth login has not been implemented yet;
- Upload of user data: once authenticated, users can import and view their data from within the web page. It has also been implemented a prototype function that accepts a Google Takeout zip file and imports some specific types of data (location history and browser history).

We would also like to include in this analysis of the starting project, the state of the frontend, UX/UI of the application. Although this was not a focus of the previous project, it has been an important and time-consuming part of this work. Taken directly from [6, Personal Data Safe: a flexible storage system for personal data], we here present the look of the home page for the P-DS.

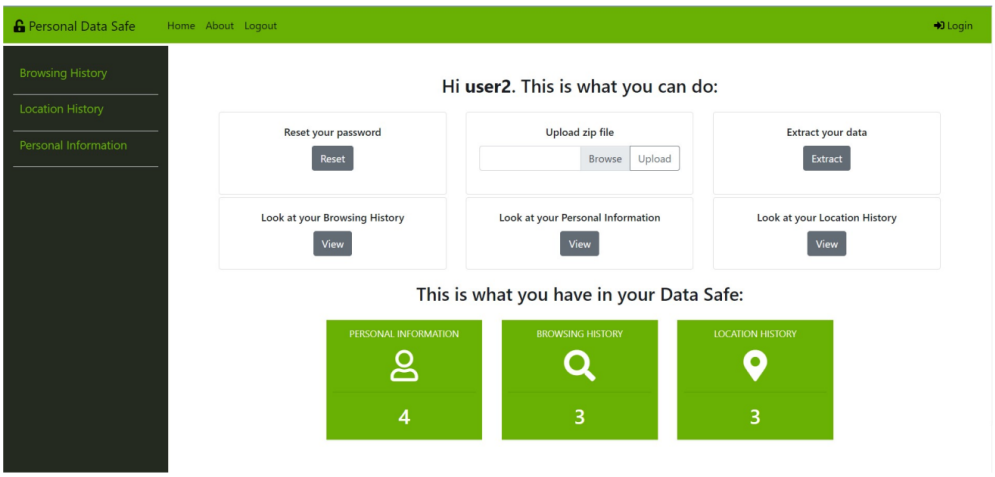


Figure 2.3: P-DS initial homepage

2.3 Goals of the project

Overall we consider the work previously done by Francesco Torta, a solid starting point from where we would like to improve the flow and the concept of the P-DS

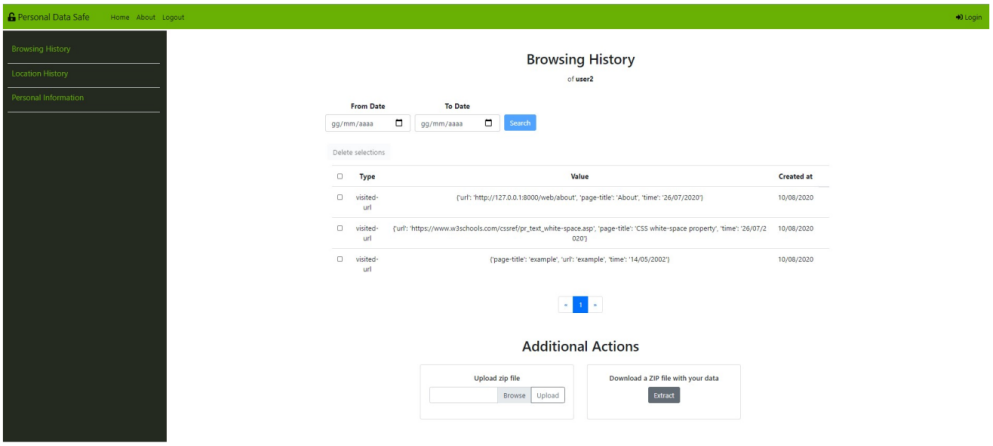


Figure 2.4: P-DS initial PersonalInformation page

both in its backed and frontend. In this section we will shortly introduce what has been touched and reworked before diving deeply into the technical analysis and choices made during the update of the P-DS.

First, we thought that even if a complete Django application was quick to set-up and update, the concept of generating HTML files in the backend and sending back static files is slowly dying in modern web development. Today's modern development techniques revolve around a distributed implementation of fullstack application, separating the frontend from the backend and developing the frontend as a client application which utilizes the backend APIs. For this reason we decided to move the frontend into a ReactJS application and leaving the backend to ONLY transfer information through APIs. In this way we also relieve some of the stress from the backend since it does not need to send HTML files but only JSON responses.

Second, we moved the database from MongoDB to a PostgreSQL. This was mainly for implementation purposes as Django is better suited and developed around SQL databases, making our lives easier with the development.

Third, we modified the authentication and authorization logic for users, from session based to token based, using JWTs (JSON Web Tokens). And we also improved the concept and role of the *Schema.yaml* file to be able to control the whole application state, both backend and frontend, just by changing its content. We felt like this could be a good step forward in the ease of implementation of P-DS as a component for different projects and companies who would like to implement our PDK.

We also did a whole rework of the style and look of the frontend, adding dynamic and responsive views and application feel-like usage of the app. We thought, this was a needed change as the state of frontend was still in the prototypical phase and could not really be considered as a commercially valued product.

Finally, we added some extra functionalities by developing the databuyers APIs, which are based on authorization given in form of a signed JWT (Asymmetric cryptography) that is outsourced by the Personal Consent Manager (P-CM). Moreover, we started the development of OAuth both as functionality in backend and its frontend counterpart view. We uploaded the full application on the server of the project and it is now accessible by navigating to `easypims.pimcity-h2020.eu/pds` .

The final goal of this work was to improve the solid foundations of the project and making it an appealing product that can be considered as finished. With this said, we don't mean that the P-DS is done, as much more work needs to be done, and we will focus on these concepts at the end.

Chapter 3

P-DS architecture: Frontend and Backend

In this chapter we will dive deeply into the changes and updates we made from the previous version of the P-DS. We will discuss why we chose to change, explain our research in the possible alternatives and briefly touch the concept of the technologies used.

3.1 From centralized, static server-side web server to fully-flagged, distributed client-side application

In this section we present the changes made at the core of the Django application. It is of substantial importance because we aim at dividing the application into two main components, releasing stress from the central and unique component which was the Django server and moving some of the responsibility to a **React** application for the frontend part of the P-DS. The Django server is still the focal point of the P-DS as it handles all the REST APIs requests and it interfaces with the database. By doing so, we allow for separate and independent developments of the two main components that constitute the data safe, we will be able to develop specific REST APIs that will not make it into the web views, such as APIs for databuyers and utility APIs for internal use. Moreover, the frontend look and feel could also be updated independently as time passes, without impacting the functionality and development of the core of the P-DS.

3.1.1 Server-side vs. client-side rendering

Since the beginning, with the rise of internet browsing and web pages, the conventional method of accessing information and displaying HTML files into the system was performed using server-side rendering. Server-side rendering works by creating, loading and inserting data into the HTML file directly on the server and, once the file is ready, sending it to the client, as is. The client web browser will then receive the HTML file and display it, for the joy of all us users. Server-side rendering has been working perfectly fine for years since most web pages were mostly just for displaying static images and text, giving little space to interactivity.

In today's usage, this is no longer the case. Today's websites are much closer to mobile applications than to pure websites: used as messages chats, videoconferencing tools, online shopping and much more. With the constant need of updating the information displayed, it is clear how the server-side rendering is not optimal as, for these types of usage, it would need to request a new composed HTML file from the server every time an update is necessary. So when is it ok to use server-side rendering?

As stated above, when data does not change very frequently and when live updates are not necessary, server-side rendering is still more than appropriate to be used. Of course, for very large scale projects, which must handle a lot of traffic, there might still be some problem as, for every user requesting a web page, the server has to process the request, compose the HTML file and send it back, which is a much more computational intense job than just sending requested data back.

This is where **client-side rendering** comes into play. When talking about client-side rendering we usually refer to rendering content in the browser using JavaScript, a lightweight, interpreted, object-oriented programming language which is commonly used for scripting in web pages [10, Javascript Official Website]. With client-side rendering, the user gets a backbone HTML document with JavaScript on the background that will fetch and update the content of the HTML document, on the go, keeping it updated.

This has quickly become the defacto-standard of building web applications and websites in general, with some JavaScript libraries becoming very popular and easy to implement. The main ones used today are **React**, **Angular** and **Vue**.

A good example of server-side versus client-side rendering is explained by freeCodeCamp, a platform for learning everything about coding [11, FreeCodeCamp article]:

Listing 3.1: example proposed with static HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Example Website</title>
  </head>
  <body>
    <h1>My Website</h1>
    <p>This is an example of my new website</p>
    <p>This is some more content from the other.html</p>
  </body>
</html>
```

Listing 3.2: equivalent example proposed with client-side HTML

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Example Website</title>
</head>
<body>
  <div id="root">
    <app></app>
  </div>
  <script src="https://vuejs.org"type="text/javascript"></script>
  <script src="location/of/app.js"type="text/javascript"></script>
</body>
</html>
```

As we can see from the second example, the HTML file contains only one div called *root* and a component called *app*. This is the backbone HTML file we mentioned above. The backbone file also has two JavaScript scripts attached to it, these scripts will request, manipulate and return the data as HTML elements; and the good news is that this process can be performed, seamlessly and periodically, giving the feeling of constant updates to the page, without ever refreshing it.

3.1.2 Django

During development, we worked at the same time on backend and frontend to be sure that they were compatible along the whole process, however modifications had to be done to the Django core server, mainly focusing on removing the Django static files generation logic and replacing it with REST APIs. Notice that in this section we will not spend time explaining how Django works, why we decided to use it or which other frameworks we could have used. If you are interested on

this discussion please go read [6, Personal Data Safe: a flexible storage system for personal data] on the highlighted section. We will also assume a basic knowledge of python for the next discussion.

Before continuing on the modifications made to our Django server, we touch very briefly the concept of REST APIs. As stated in [12, Restful API website], REST is acronym for **REpresentational State Transfer**. It is an architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation. The concept of REST is based on some fundamental principles, which are:

- **Client–server**: By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless**: Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
- **Cacheable**: Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- **Uniform interface**: By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- **Layered system**:The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.

And this is why, by migrating to a client-side rendering model, we needed to update and rework the REST APIs of the Django application. We focused on creating new REST APIs by adopting the Django REST framework, which is a toolkit used for building Web REST APIs [13, Django REST framework]. This really shows the power of using Python for programming the backend, as we can

easily find packages and toolkits that help us immensely to speed up the process of developing the app, while remaining standardized and even controlled. To start using it, is as simple as adding a couple of lines of code into our project and we can start taking advantage of its features.
Django's rest framework can be installed using pip with the command:

Listing 3.3: Installation of djangorestframework

```
pip install djangorestframework
```

then we simply install it in the Django APP, that is our server, by adding it in Django project configuration file:

Listing 3.4: Installing package into DJANGO APP

```
INSTALLED_APPS = [
...
'rest_framework',
]
```

Django REST framework, uses the models already defined in normal Django but adds an extra layer of functions on top of it to ease, standardize and speed up the development. It introduces **Serializers**, used to serialize information before inserting it into the database or before sending it as JSON response, **ViewSets** to standardize the CRUD operations (we will discuss about CRUD and all the REST APIs functionality soon) and **Routers** to provide an easy way of automatically determining the URL configuration [13, Django REST framework].

So from now on, all CRUD operations need to be standardized into specific REST APIs, but what is CRUD? **C**reate, **R**ead, **U**pdate and **D**elete are the basic function the REST APIs model should use. There must be a way to perform these actions, using the APIs. With this we are not stating that each user should be able to do it, but that, if it is possible and it is authorized, an API should take care of it. And this was another change we added to our Django server: it now handles CRUD operations on user personal information. Of course not all information can be freely modified or deleted, but this is also determined by the settings of the Schema file because as stated earlier, the Schema file has been treated as the focal point that governs the behaviour of the P-DS.

3.1.3 The Frontend Web APP

Once set up the basic APIs on Django, we started to migrate the static HTML file logic into a client-based application, which will consume our new APIs and will make the user experience more interactive and clear, with up to date modifications

which don't need the refreshing of the page to be displayed.

We had to choose how to implement it and the choice fell on some of the most uptrend and most used JavaScript libraries for developing Web Applications: **React**, **Angular** and **Vue**. Javascript libraries are frameworks which are focused on developing single page applications, that are simply put, applications that resemble those running on smartphones, but that instead run on web browser. They work based on the concept of client-side rendering that we explained in the previous sections: starting from an empty backbone they will fill it accordingly depending on the logic behind it. These libraries became popular for their **component-based** architecture, in that it allows to build encapsulated components that manage their own state, then compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep the app state out of the DOM [14, React official website]. All these libraries will run on on the user browser, from which the user requesting the web views of the P-DS will contact the Django backend for updating and displaying data. In this way, we will have solved one of our tasks: separating and distributing frontend and backend, allowing the frontend to consume the APIs as a client and generalizing the usage of the backend logic. The only additional piece needed, in comparison with the previous design, is another server (the frontend server) which is listening for requests and distributing to the requesting user the backbone of the application with its logic in form of JavaScript files.

Let's now briefly analyse the choices we had in terms of JS libraries and after that we will discuss which one has been chosen, why and how the work has been developed.

Angular vs React vs Vue

Before starting we want to state that the final decision of which library to use has been strongly dependent on the developer's personal experience with the tool. I had previous experience developing web applications in React and therefore we decided to use React as our choice. Nevertheless it is surely good to go over each library, understanding its strength and weaknesses. We will go into more details with React being it our final choice. To help us gather information and differences about these frameworks we will often cite the work done by Sneha Das, Senior Technical writer and consultant in her work [15, AngularJS Vs. ReactJS Vs. VueJS: A Detailed Comparison].

Angular is an open-source framework developed by Google in 2009, it has become popular mainly due to its leverage of using HTML as a template language. It allows for component views development which increase re-usability, involves

the usage of declarative coding which makes the code quite simple to read. It has one of the oldest most established and active communities in the field, which will guarantee assurance to be covered under many development aspects. It also supports Two-way data binding, which is one of the most useful features of AngularJS. Thanks to Two-way data binding, the changes made in the user interface can influence web apps objects immediately and vice versa. In simpler words, any changes that have been made in the user interface will immediately reflect in the app interface too. The main negative point is that it is heavy and since our application was not going to be processing heavy loads we did not consider this as a viable choice [16, Reasons to Choose AngularJs for Web App Development].

Vue is famous for being very reactive, this means that when you set variables in Vue, it will automatically update the user interface. As with AngularJs, Vue is scalable and flexible, this means that it can be used for large applications as well as to construct small interactive parts to be integrated with different technology. In practice, it uses the same concepts of components. the main drawback of Vue is that the user base, although it is growing fast, it is still relatively new and small. This could bring to undocumented procedures, unexpected bugs and hard to find solutions.

Lastly, our choice, **React**. As already said the main decisive reason for using React was my previous experience with the tool, which cut to zero the learning time before being able to work on a project. Moreover React is very developer friendly, with its fast pre-builds that set up an initial project. It also uses components view, which allow for a distributed development and reuse of code, as an example, it is possible to define a special custom table with filtering actions and reuse the table for different parts of our APP, which is exactly what happened, since it has been reused the same table component for all personal information views (more on this later).

Moreover, React was founded in 2013, which means that its lifetime is long enough to be considered stable, it is used by major companies such as Netflix, Airbnb, Storybook, Facebook, Instagram, Whatsapp, Intercom, Atlassian and more. Its userbase is growing strong, being the currently most popular Js framework per GitHub Survey [17, GithHub Survey on most popular Js Frameworks] as it is depicted in Figure 3.1:

For these reasons, we decided to implement the frontend of the P-DS with ReactJs. We will now discuss the changes done to the frontend and how it interacts with the backend, diving deeply in some React and Redux mechanisms which are really promising and useful.

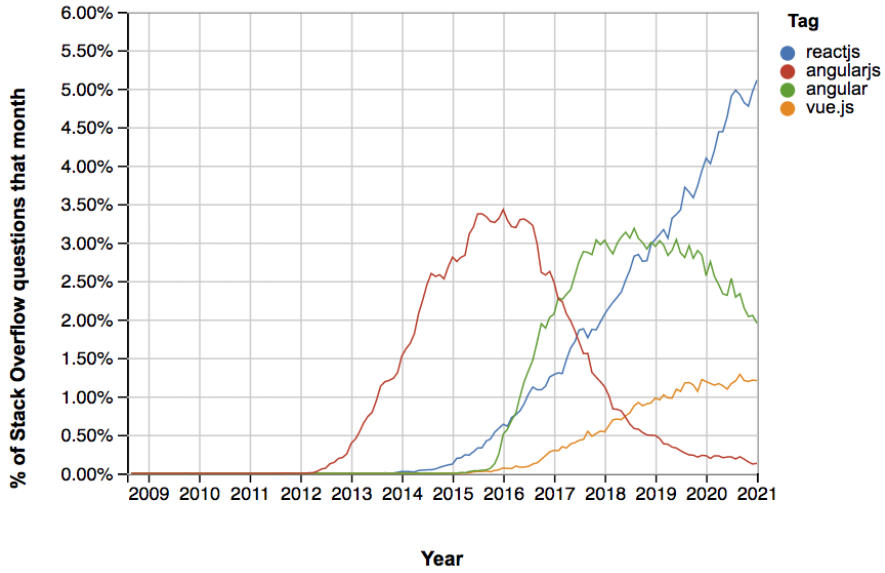


Figure 3.1: Popularity Chart for Js Libraries

React implementation

As previously stated, React basic concept is that of **components**. We can program a component to perform specific tasks, formatting the information received and return a final result (which in most cases it is a HTML code snippet that will be rendered live on the browser). We then started to developed the application, and the main concept of using React is to make the experience similar to that of using a smartphone app. We used the concept of **single page application**, that is there are no url renderings or refreshing of the web page; this said, the application will have different views, feeling like navigating to different pages, but on the background only one is running. Depending on the path the user reach, some components will activate and others will deactivate, giving the feel of switching between tabs, or section of the web app.

As demonstrated in the following Figure 3.2 and Figure 3.3, we present the newly designed **P-DS Homepage** and one of the **personal data pages**.

The homepage, runs on *https://easypims.pimcity-h2020.eu/pds/*. This is the first page an authorized user sees. By navigating through the app, the url (for Figure 3.3 it would be *https://easypims.pimcity-h2020.eu/pds/personal-information*) changes but this is not another request to the react server, comporting a reload of the page. It is, instead, the loading of a new component and the disappearing of the previous one (the homepage component), which triggers a set of REST APIs request to

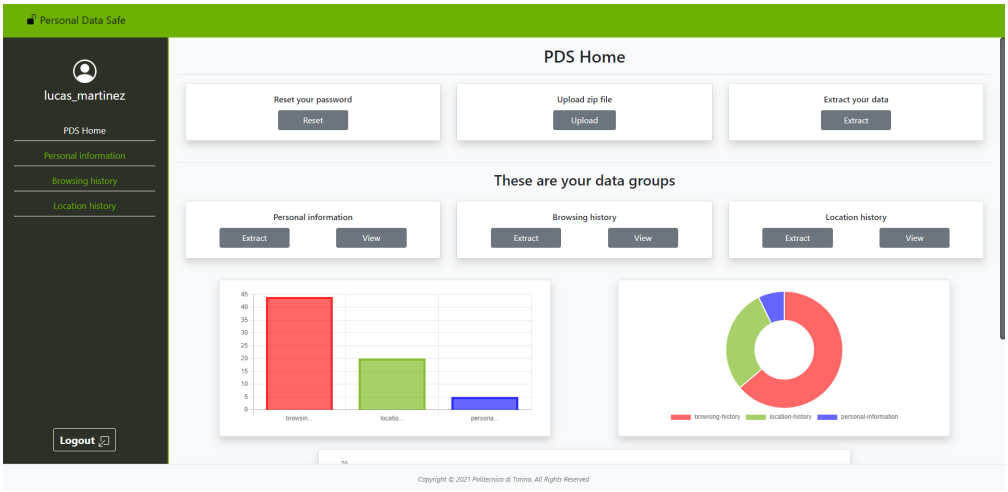


Figure 3.2: P-DS Homepage

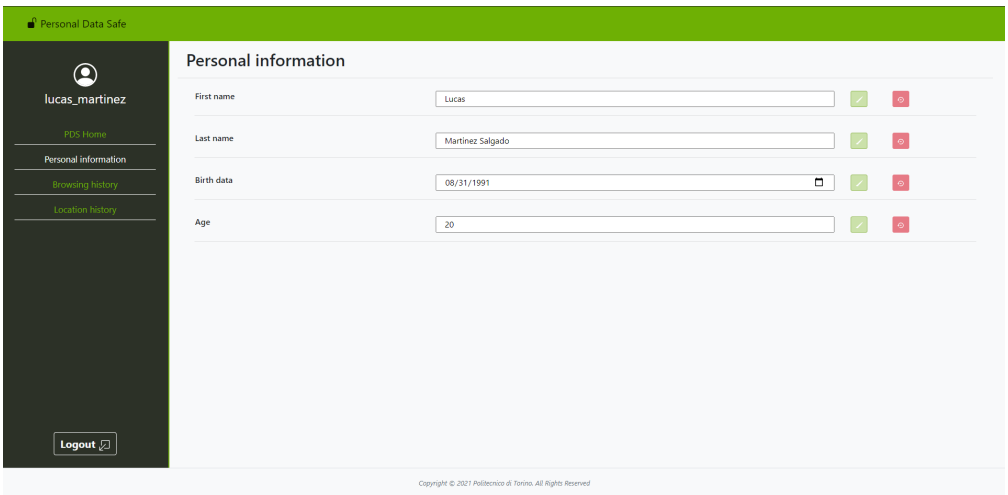


Figure 3.3: Personal Information Page

the Django server. In this way we can still give the user the feeling of "browsing the web" while it is simply a switch of component. This is really important as there is no need to exchange HTML files with the server, only the needed APIs are called, which result in less strain on the server and the network in general. This is the concept that many tech giants are implementing on their side, such as Netflix, Facebook, JustEat, Airbnb, Instagram and much more...

For the same reasons of giving the feeling of an APP running on the browser, we concentrated a lot on a particular feature of web pages and applications: **responsiveness**. A web page is responsive if it adapts its look, effects and

functionalities to the screen size. Previously, since the focus of the project was not to deploy a polished frontend product, this feature was completely missing, making it impossible to run the P-DS web view on phones or even small browser windows. Now, with the help of React and CSS, the application will run seamlessly on every kind of device. We show in the following Figure 3.4, how the view of the page adapts to different screen sizes.

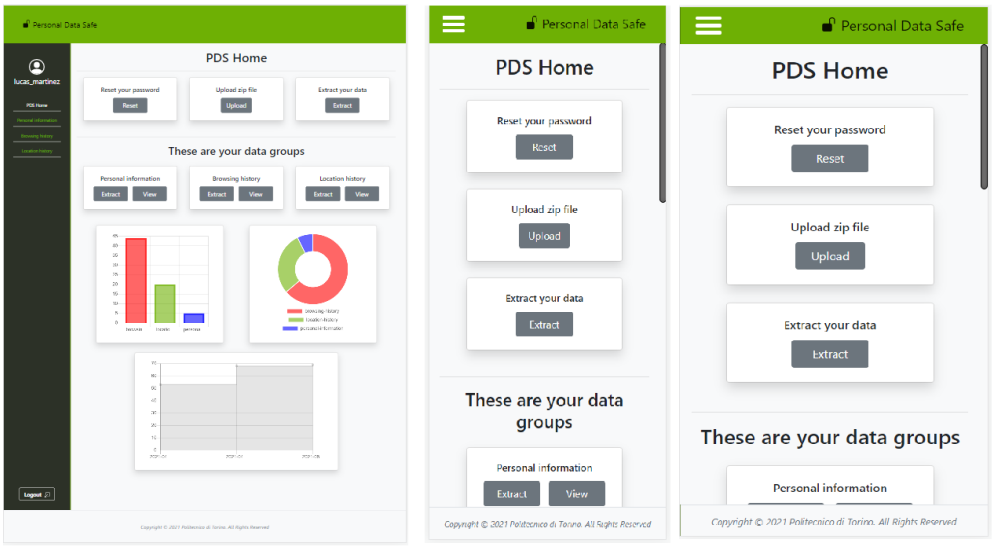


Figure 3.4: Comparison of different screen-size View

On the left, we have an Ipad Pro view, and due to its size, there is not a big difference from the desktop view. Yet being oriented vertically, it will allow for more space and will display all the visual tools at once. Instead, with the two phones, having a smaller screen size, it has been introduced a hidden navigation bar on the side. It is then possible to stretch the components in size, making them usable on the phone. As showed, when accessing the P-DS web app from a phone using the built-in browser, the feeling will be the same as if using a mobile application.

The best part about the concept of components is that we can dispatch particular actions to different components, an example could be requesting data through an API request (maybe for a component that displays a table), another example could be formatting the data (such as a component representing each entry of the table) and another example could be checking the status of a user. We will discuss about user authorizations and authentication in the React application later, for now it is important to know that the status of an authenticated user is kept at application level, which means that each component has the capability, if wanted, to retrieve this information. In our React design we have divided our app into **app-level**

components, **pages** components, **utility** components and **specific** components.

- **app-level** components: we only have two components in this category, **authenticated** and **non-authenticated**. When the user is authenticated, the component containing the whole application will load, giving access to all information and functions. If instead the user is not authenticated, the log-in component will load, displaying the log-in page. By setting this authentication check at this level, we are sure that only authenticated users can have access to the sensitive, personal information.
- **pages** components: are the containers of everything that is displayed, they are the direct child of the app-level components and are visualized between the header/nav and the footer of the whole application. Containing all displaying components, such as tables, titles, buttons and more, by switching among these, we give the feeling of "changing" page. In this category we identify 3 different categories: **Homepage**, **editable-information**, **non-editable information**. The homepage is self-explanatory, it gives an overview of the user current situation in its P-DS. The other 2 categories are different, in which, some information should be editable by the user, e.g. address and phone number, while others should not be possible to modify, e.g. browsing history or location history. In order to choose which information is editable or not by the user, it is as simple as to modify the Schema.yaml file.
- **utility** components: are those components that can be placed inside different levels and have a utility function, such as the Navigation Bar, the Header, pop-ups and more.
- **specific** components: are those components that compose a page. They could be buttons, tables, titles but can also be a set of components grouped into one, like a chart box containing the chart, its title and description.

We propose a logical overview of the hierarchy of the React App design in Figure 3.5.

We have now seen how the different "pages" or parts of the app are rendered and activated. We discussed earlier that, at the app-level we check for authentication and then keep this information available in the whole app. In React, to transfer information from a parent component to a child component, this data must be passed as an argument of a function, this is because every component is actually built in the code by a function (more common) or an object instance (less common). It is clear now, that if we want to distribute information along the hierarchy of components described above when lot's of "jumps" are performed, it will result in a lot of data passed to functions. The most critical case is in fact the one of the user

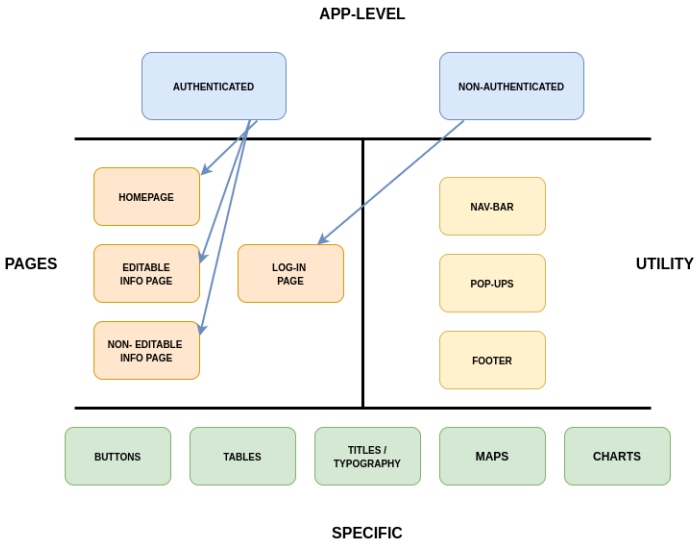


Figure 3.5: Logical tree architecture for React App

authentication data, containing all the information to perform the APIs request and authorizations of the user, that need to be passed from app-level components all the way to specific components.

To avoid this problem we used **Redux**, a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion [18, Redux official website]. When triggered by our log-in component, Redux will perform the necessary authentication steps, recording useful token information and expiration dates on local-browser storage and making it accessible from any component of the application. This is also how the app-level component checks if the user is logged-in or if some actions need to be performed.

Maps and data visualization

At last we will discuss about the focal point of the P-DS web app, displaying the information to the user. Data information pages are divided into two categories: **editable** and **non-editable**, and this is a setting that must be inserted in the *Schema.yaml* file.

Editable information will have a different visualization display as showed in Figure 3.6 and it will allow for all CRUD operations on all data in such category

(at least stated otherwise in the *Schema.yaml*).

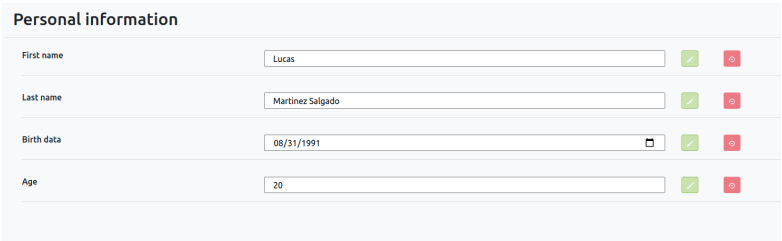


Figure 3.6: Example of editable data page

As an example, we could create a personal information page containing all information of a user's life (e.g. name, last name, birth date, address), then, we could set which fields can be edited and which cannot. For example the address should be editable while the birth date should not. This tuning of the data preset, can be done simply by changing the settings in the *Schema.yaml*. Users will be greeted with a simple and intuitive interface where they can edit the respective fields and then choose to confirm their changes or restore the original format. This is possible because with React, we keep in memory, as a separate variable the original value of the field, while the displayed value is the one that gets modified. If the user confirms the change, an API request for modification will be sent out and, if successful, also the original value will be updated. If instead the change is discarded, the display value will be reverted to the original value.

Non-editable information instead, can only be added, displayed and deleted. This is because this type of information, if edited, lose all their value. As an example we could use the location of a user: if the user could edit it's location history, he could prove to have been in a place it was not, and it is clear how this is would make any location record worthless.

We provide two examples, we mainly worked on: **location history** and **browsing history**, used to record respectively the physical location of a user, recorded in terms of geographic coordinates, and the web pages visited. Both of these category are identified as **groups** in the *Schema.yaml* and we will discuss later how to configure properly each group by defining each field inside of it. An example of location history page is displayed in Figure 3.7.

For this group we set-up the following fields: latitude(float), longitude(float), description(string), time(date). We can see how the non-editable information page has, as main component, a table displaying the data divided in columns representing the fields. Some columns could be ordered in either ascending or descending values (date and name for example) by just clicking the column titles. An arrow will display the current method of sorting.

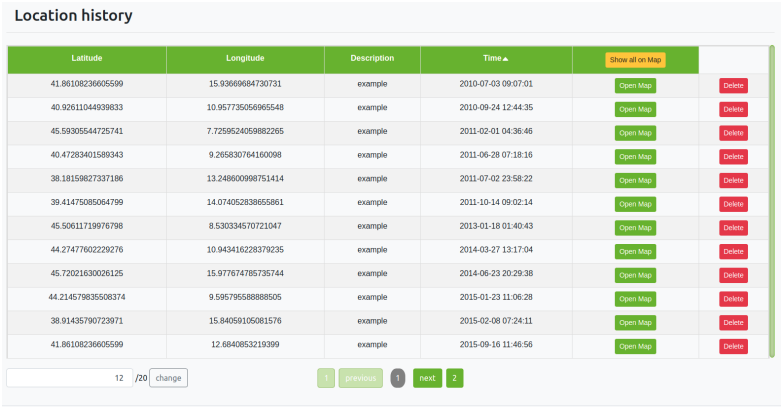


Figure 3.7: Example of Location History Page

It is also present, in the left bottom corner, an editable input field, to control how many entries to display at once. This is done to restrain unnecessary processing from the React application. Each API request executed, even if not specified, will limit the number of items received by the backend. We don't want to receive all entries at once (that could become hundreds of thousands over time) but only those that will be displayed. Therefore every time a change occurs into the table a new API request is performed by the component table itself.

To allow for better exploration of the table, at its bottom we introduced a quick navigation toolbar, which is updated live depending on the number of displayed items per page.

Moreover, for this specific setting, if it contains a latitude and longitude field, we introduced also the possibility to display it in Google maps (also set up in the *Schema.yaml*). By clicking on the map button, that is part of all rows, a pop-up map representation will appear; there is also the possibility to checkout all the currently displayed locations at once by clicking the top most map button, as displayed in Figure 3.8 and Figure 3.9.

For the browsing history, instead, the only feature remains the possibility of deletion, as displayed in Figure 3.10.

We think that this new and separated view between editable and non-editable pages will be more recognizable and intuitive for the users. Moreover it gives and extra field for tuning and customizing for the developers of new P-DS.

3.2 From MongoDB to PostgreSQL

The next big change done to the P-DS that we need to discuss, has been the migration from a MongoDB database to a PostgreSQL database. The P-DS has been developed previously with MongoDB because it allowed for better flexibility

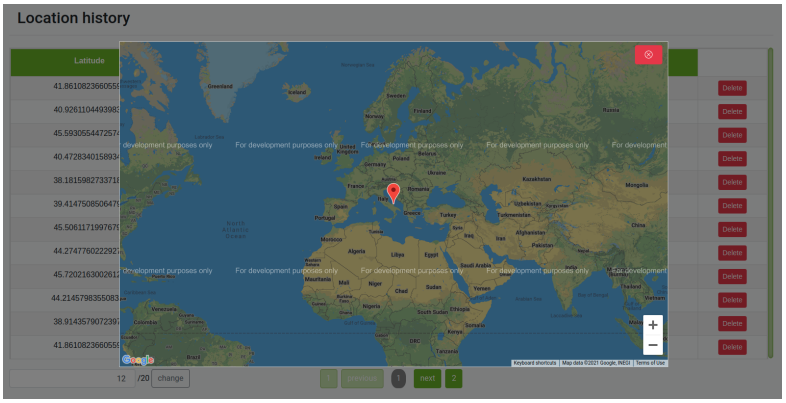


Figure 3.8: Example of Single Location Map Pop-up

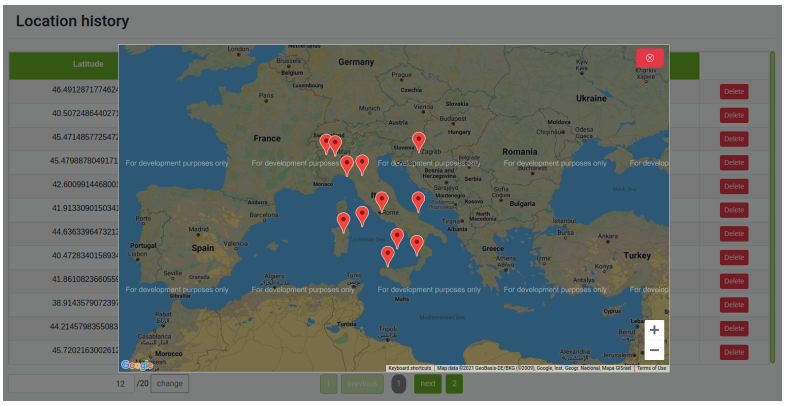


Figure 3.9: Example of All Location Map Pop-up

when defining new personal data groups: it was simpler to structure the wanted fields and to manage a large pool of non-uniform data.

Before discussing why we decided to change, let's have a quick introduction of the different databases types.

SQL vs. NOSQL

The differences between SQL and NOSQL databases can be summarized into 5 focal points, as summarized in [19, SQL vs. NOSQL]:

1. SQL databases are relational, NoSQL databases are non-relational.

2. SQL databases use structured query language and have a predefined schema. NoSQL databases have dynamic schemas for unstructured data.

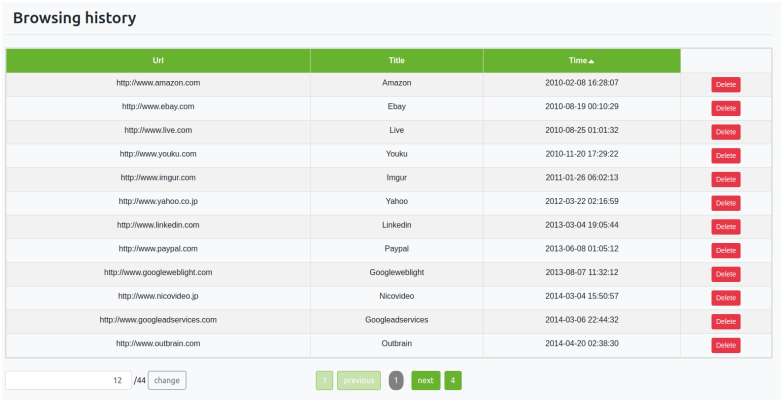


Figure 3.10: Example of Browsing History Page

3. SQL databases are vertically scalable, while NoSQL databases are horizontally scalable.

4. SQL databases are table-based, while NoSQL databases are document, key-value, graph, or wide-column stores.

5. SQL databases are better for multi-row transactions, while NoSQL is better for unstructured data like documents or JSON.

A **relational database** is structured, meaning the data is organized in tables. It may also happen (quite often actually), that the data within these tables are related with one another making therefore a connection among tables, or dependencies. A non relational database is document-oriented, meaning, all information gets stored as a big collection of "documents" which are not related to one another by the above mentioned dependencies.

SQL databases use **tructured query language** that are based on a pre-defined schema for defining and manipulating data. SQL is considered to be one of the most versatile and most used query language, therefore it can be considered a safe choice for the majority of the cases. As a negative point, it can feel restraining as it uses those pre-defined schemas. NOSQL databases, instead, have dynamic schemas for unstructured data. Data can be stored in a variety of manners as documents, graphs, KeyValue Pairs. This makes NOSQL very flexible for different uses.

The **scaling** of a database also differ, as SQL database scales vertically as to add rows into a table, while NOSQL databases scale horizontally which means that they can handle increased traffic simply by adding more servers to the database. NoSQL databases have the ability to become larger and much more powerful, making them

the preferred choice for large or constantly evolving data sets.

The main SQL database systems are MySQL, Oracle, Microsoft SQL Server and PostgreSQL. The main NOSQL database systems are: MongoDB, Cassandra, Amazon DynamoDB.

The main issue with our starting project is that, Django is set-up to work with MySQL (which is an SQL type database), it can also easily work with all other SQL databases (among which PostgreSQL). For interacting with MongoDB, instead, it needs to use a 3rd party interface, that translates all the SQL queries to MongoDB syntax queries. We think that basing the project on a non-supported 3rd party interface would not be a smart decision for long term development. These are the main reasons why we decided to switch to PostgreSQL. We chose PostgreSQL because team members had some previous knowledge of the functionalities and set-up procedures.

To start and allow interaction between the Django application and the PostgreSQL DB it is very simple, and the steps are completely documented within the project. Simply put, after installing PostgreSQL and creating a new database, we can link it to the Django APP by adding it within the settings file, filling up some fields and we will have the link. We then need to create the schemas needed to define the different entities and the connections among them. For our project we defined a personal-data entity and a user entity and connected the two of them with a one-directional link from the personal-data to the user.

We now have to face one big problem: SQL databases have fixed fields and, for each of our data group, we might need different fields, so how do we fix this?

By adding a JSON field in the schema of personal-data, we are able to insert JSON content in that field and that would be the content of our data. This change makes our searches slightly worse, but the increased speed of SQL will compensate for it. Moreover we mostly search for data by ID, owner, or category and not much by content.

The new DB has been set up and the logic of the various APIs has been changed to reflect the changes made in the DB design. We are happy with this change as it smoothed most of the DB operations and interactions with the backend and also allowed us to use simplification features offered by the REST API packages that are available with Django.

3.3 Authentication and JWT

As described previously, we decided to move into a distributed design for our application, where the frontend will operate by "consuming" the backend's REST APIs. Every call to the API is performed over HTTP(S), which is a stateless protocol and ,therefore, all the requests performed will also be stateless as well. But we want to give our users a consistent and convenient experience, as most other online services do, by remembering the state of their connection so they do not need to "log-in" every time they want to make a request. The main two approaches to this problem are the usage of **Session Based Authentication** and **Token Based Authentication** [20, Session vs Token Based Authentication].

Session Based Authentication

A session will be created for a user after the log-in procedure. For each session, its ID will be stored on a cookie in the user browser. For each request, the cookie will be sent together with the request, and the server will compare the received cookie session ID with the session information stored in memory.

Token Based Authentication

Instead of using sessions, it is possible to use JSON Web Token (JWT) for authentication. Upon successful log-in, the server generates a JWT with a secret before sending it to the client. Then the token will be sent in the header of every request. The server will then validate the JWT, its expiration date and any other information before replying to the client's request. The main difference is clearly the fact that the tokens are not kept in memory.

There are advantages and disadvantages on using both these authentication methods. We opted for JWT Token Based Authentication because we believe that it best suits our REST APIs new model and, moreover, it might take less strain on the backend, which must not check and look for session IDs, once lots of users will be using the platform.

For simplicity here we opted to use a python package, which is part of the *rest framework* used in most of the project, to take care of the logic. It is as simple as creating some Django Views (endpoints) and the package will take care of the authentication for the users' requests. We added some more logic and depth to it by introducing the concept of **blacklisting** a JWT. This is very useful for log-out operations or if some token has been compromised. Each Token will have an expiration date, which can be set-up in the backend as an option variable. This also means that, until the expiration date is reached, the token will be operational, but we want to give the user the ability to log-out or change user while working on

the same browser. To implement this, we created in memory a list of blacklisted tokens, these are tokens which are not usable anymore and therefore, a request with such tokens must be rejected. When a user logs-out we will insert its token (which will appear in the log-out request) into the blacklisted list and, from that moment, the user will be officially out of session.

JWT

In this brief section, we describe what are JWTs, their composition and usage.

As defined in [21, What is JSON Web Token?], JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. Being it digitally signed, either via a secret (with the HMAC algorithm) or via a public/private key pair using RSA or ECDSA, a JWT can be trusted. It can provide **integrity** and, if encrypted, also *secrecy* between parties. The main usages for JWT are **Authorization** and **Information Exchange**. For our application, we use both, as we mainly use it for authorizing users after their log-in, but also, inside each authorization JWT, we insert information about its validity and expiration.

JWTs are composed of three parts, separated by dots, which are **Header**, **Payload** and **Signature**. A typical JWT will look as follows:

xxxx.yyyyy.zzzzz

The Header will contain the signing algorithm used (e.g. HMAC SHA256), the Payload will contain the information in JSON format. They will be BASE64Url encoded. The signature, instead, will take the previous sections, a secret and the algorithm specified in the header as input to the sign function. In this way we have an plain-text section which can be easily analyzed by at application level environments and a secret section which can be used for establishing authenticity and integrity.

In Web usage, as well as in our application, JWT are sent in the Authorization Header using the Bearer Schema, creating an header that will look as follows:

Authorization: Bearer <token>

In our application we use JWT both for common user Authorization and also for data-buyers authorization. We will explain the usage and management of information for data-buyers using JWT in the respective section.

3.4 Using the .yaml file to control all the functionalities of the P-DS

As previously stated in multiple occasions, the set-up and settings chosen for the P-DS are contained in the **schema.yaml** file. This file is used to define multiple aspect of the functionalities and information that the database can contain, display, receive and manipulate. In this section we will describe its options and the result functionalities for each option, then we will discuss how we modified the logic of the backend, at start-up, to make the whole P-DS customizable and operative only by editing the schema.yaml file.

The schema.yaml file is nothing more then a set of options and choices we can use to modify the behaviour of the P-DS as we want, it is thought as to allow non-programmers, to be able to modify the P-DS functional behaviour without having to code changes and it might be very useful in the future to develop the PDK.

An example of what the file looks like is presented next in Listing 3.5:

Listing 3.5: Schema.yaml example

```
name: 'PIMCity default P-DS schema'
version: 0.1
author: 'Federico Torta, Martino Trevisan, Annaloro Enrico'
content:
    - group-name: personal-information
      user-insertion: true
      user-update: true
      add-zip-file: false
      extract-json: true
      types:
          - name: first-name
            historical: false
            type: string
          - name: last-name
            historical: false
            display-name: last name
            type: string
          - name: birth-data
            historical: false
            display-name: birth data
            type: date
          - name: age
            historical: false
            type: int
    - group-name: browsing-history
      user-insertion: false
```

```
27        user-update: false
28        add-zip-file: true
29        extract-json: true
30        types:
31            - name: visited-url
32              historical: true
33              type: dict
34              fields:
35                  - url: string
36                  - title: string
37                  - time: date
38    - group-name: location-history
39      user-insertion: false
40      user-update: false
41      visualization-hint: map
42      add-zip-file: true
43      extract-json: true
44      types:
45          - name: visited-location
46            display-name: visited location
47            historical: true
48            type: dict
49            fields:
50                - latitude: float
51                - longitude: float
52                - description: string
53                - time: date
```

The nesting of tags can be deducted from the code snippet. As reported and described by Federico Torta in his work [6, Personal Data Safe: a flexible storage system for personal data], some possible fields are:

- **group-name**: it defines the macro group, which the information belongs to. This field is used to group entries into semantic sets, in order to have a hierarchical and structured repository. Example of group-names are personal-information, browsing-history, location-history.

- **types**: this field is another list and it specifies at a finer-grained level which kind of information each group includes. Basically the types field defines the hierarchy of a single group-name field. For example, data that can be classified as personal-information may be the year of birth, the email address, first name and last name, etc...

- **name**: each entry of the P-DS is associated to a name that describes in a human-understandable way the content of the entry. For example, the user may insert his birth date in the Data Safe and a plausible name for the entry may be "year-of-birth".

- **type**: also the type field is linked directly to a P-DS entry. It defines the type, at code-level, of the information. This field is required to perform consistency control functions and avoid the user inserting erroneous data, such as a string object for a field that requires an integer.

Other than these basics fields, other fields have been added to the list, they define features and authorizations that the user might want to have to enhance and protect its experience with the platform.

- **user-insertion**: a boolean option that defines if the user is allowed to insert manually an entry for that specific group. For example we would like the user to add manually an address (or maybe more if needed) but we do not want him to add a location history entry as that could lead to the generation of relevant false data.
- **user-update**: a boolean option that defines if the user is allowed to modify manually an entry for that specific group. The previous example is valid for this field as well.
- **add-zip-file**: a boolean option which is used to determine whether the user can use the option of uploading information from a zip file (for now this is in development phase and only accepts certain fields of a Google Takeout zip file). This is very useful for groups like location-history or browsing-history as those information are already recorded and logged by browsers and/or Google.
- **extract-json**: is a boolean option that defines if the user can export the information contained in that group as a zip file.
- **visualization-hint**: a new field, added to customize how some visualization of data is exposed in the web app. As of today the only option is **map** and this allow to display coordinates as point in a google maps. This feature has been developed for the location-history group and more visualization ideas to be implemented in the future are tables (for extra data), carusel (for images) and more.

Some of these fields depends on other fields or information contained in the relevant data of the group. Since we want to be sure that, after customizing the file, the status and operability of the backend will remain stable, some test and initialization check scripts have been developed. These scripts will check that all the needed fields are present and if some dependencies are needed (e.g. with the visualization-hint option), that those dependencies are defined in within the group. If, for any reason, these checks fail, the server will not start. This is because, once the checks are passed, the schema.yaml file settings will be used to generate, delete

or modify the tables where the data is contained. Admin rights are needed to make any modification and more controls should be used in production phase.

With our latest changes, we know have a distributed application, therefore the changes made to the Schema.yaml file must also reflect to the frontend. For example, we explained earlier how the frontend will display different types of "pages" depending on how many groups are defined. Being the groups defined in the backend we will need to distribute the information to the frontend. For this reason, and to deliver other types of information to the frontend (another example is the map display option), a set of APIs has been created. These APIs take care of this information requests and delivery from backend to frontend.

As an example, we will now add a new group by using the Schema.yaml file configuration. This new group will be used to record the purchase history of the users. We will add a purchase location type, to record the location in which a certain purchase has been performed, therefore we might want to add the **visualization-hint** option; moreover the we will want to record latitude, longitude, the amount payed, the time of purchase and maybe give a brief description. Here in Listing 3.6 are shown the changes made:

Listing 3.6: Adding a macro group to the schema

```
- group-name: purchase-history
  user-insertion: false
  user-update: false
  visualization-hint: map
  add-zip-file: true
  extract-json: true
  types:
   - name: purchase-location
     display-name: purchase location
     historical: true
     type: dict
     fields:
         - latitude: float
         - longitude: float
         - description: string
         - time: date
```

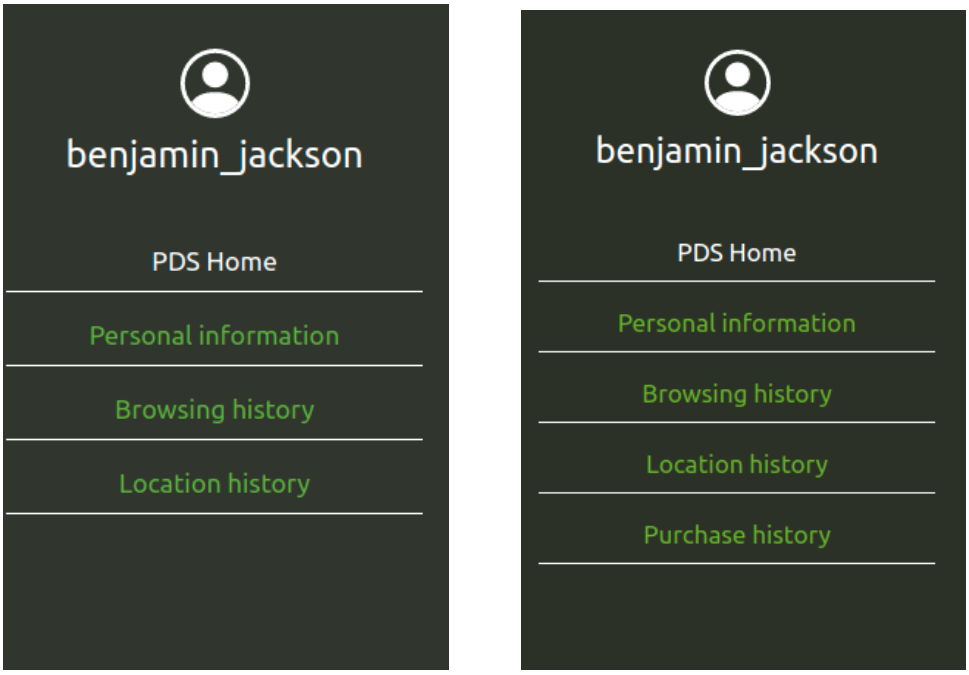
These changes will be considered only upon restart of the server, because we want to check if such modifications are acceptable and could work. If the tests are passed, then the changes will be automatically added. In Figure 3.11 we can see how the checks are being performed on start-up.

```
(venv) enrico@enrico-B450M-DS3H:~/Desktop/university/thesis/personal-data-safe/backend$ python manage.py runserver 0.0.0.0:8000
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 07, 2021 - 17:37:14
Django version 3.1.7, using settings 'config.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
```

Figure 3.11: Schema is checked upon startup

Finally, when opening the web app, we can see how all users can now access a new "page" containing the new category introduced, as shown in Figure 3.12.



(a) before **(b)** after

Figure 3.12: Web APP reflecting changes in Schema.yaml file

We have successfully created a distributed APP which is modifiable, both in frontend and backend, by only editing a single file. We believe this to be a big step towards the creation of the PDK, as we will need to create commands and options that will control the whole application generating changes automatically, after performing the relevant tests and checks.

Chapter 4

Databuyers APIs

The next main step in the development is the backend logic and the APIs for databuyers. This, being one of the fundamental concepts of PIMCITY to allow for controlled distribution of the users' information, represented an important step towards a future definitive version of the platform. In the following sections, we will try to explain the main concept of how databuyers can access the users' data contained in the P-DS, how they can get authorization of access and how we secure the interaction between different components of the whole platform.

4.1 The interaction with the Personal Content Manager

As described in [4, PIMCity project description], the Personal Content Manager (P-CM) is the means to control personal data. It is the means to define, once and for all, the user's privacy policies for consent management. It defines the policies the users desire to apply when sharing personal data with services. In details, it defines which data a service is allowed to collect by managing explicit consent.

For this reason, in order to allow users' information to be distributed to other entities or individuals, we have to coordinate our actions with the P-CM. The general workflow, as described in Figure 4.1, follows some simple steps:

1. Databuyers contact the P-CM asking for permission to retrieve users' data. This part is completely managed by the P-CM and by the users' choices of whom can access their data.

2. If the request is successful, a signed token will be returned to the databuyer. This token will contain a list of data that the databuyer can "withdraw" from the P-DS. The security in this procedure is given by the sign of the P-CM in

the token. In this way, the P-DS will be able to trust the token given by the databuyer.

3. With the signed token, the databuyer will perform its request by means of the Databuyers APIs developed and retrieve the results.

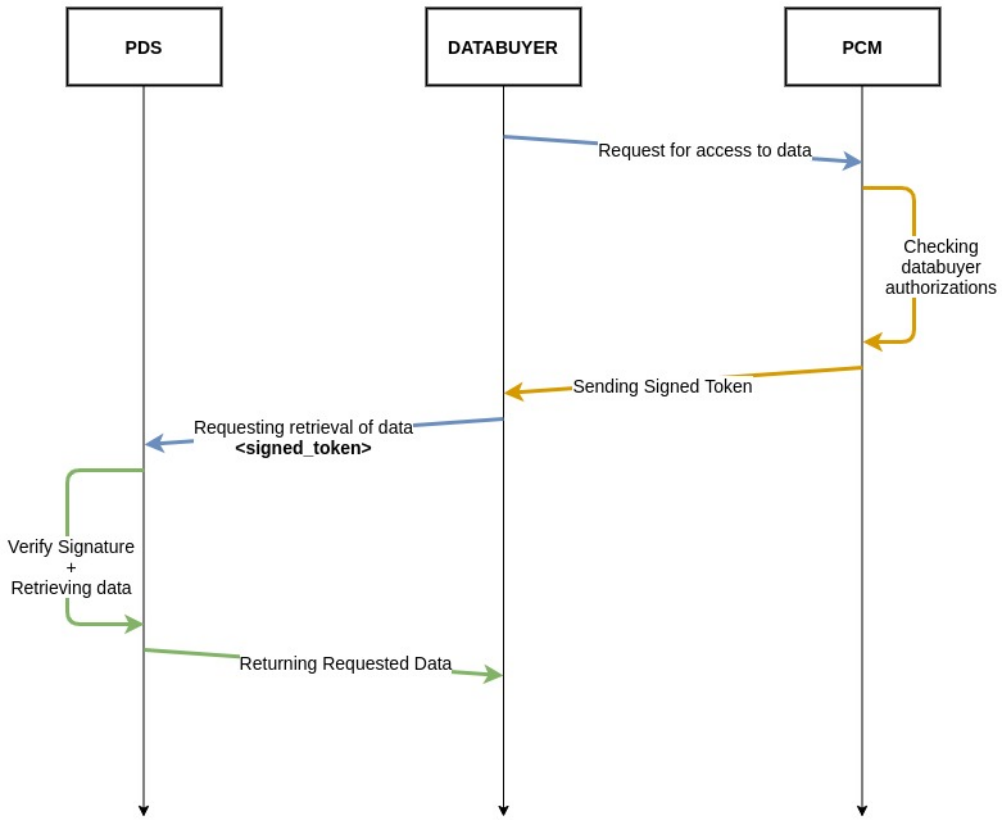


Figure 4.1: Databuyers request workflow

The databuyer will then act as an intermediate step between P-CM and P-DS. This reduces the stress from our platform and gives better independence to the clients on how and when to perform API calls.

Once the databuyer has received the token, it won't need to request a new one unless new types of data are needed or the original token has expired. In this way we release stress from the P-CM and we also create a better user experience for the databuyers which only need to request the token once and then can make multiple requests; for example if the databuyer wants to implement his website to display the users' data on their profile page: it can either take the data once and

store them in its DB, but if the data gets modified by the user in the P-DS, these changes won't reflect in the profile page because the DB does not get updated. It can instead use the same token as for the first request (therefore storing the token in DB and not the data) and then perform request to the P-DS using the developed APIs. In this way the content will be always up to date and the information of the users will be protected by our P-DS.

4.2 The security in the Token

We stated that the we can trust the validity of a token because it is signed. but how does the procedure of detecting trusted tokens work?

Again we resort to JWT, and this is because of three main reasons:

1. They provide the asymmetric key signatures needed for trusting the information inside the token.
2. They can pass information as a compact and encoded message.
3. We already used them for authorization in our P-DS web APP and therefore we are more familiar with them.

Asymmetric cryptography is exactly what we need, because we will use the databuyers' users as intermediate step, as previously presented. Therefore we must have some procedure to understand if the token passed is actually been modified or if its integrity is intact.

But what is asymmetric cryptography? As best explained in [22, Search Security - TechTarget], Asymmetric cryptography, also known as public-key cryptography, is a process that uses a pair of related keys (one public key and one private key) to encrypt and decrypt a message and protect it from unauthorized access or use. A public key is a cryptographic key that can be used to encrypt a message so that it can only be deciphered by the intended recipient with their private key, viceversa a message encrypted with the private key can only be deciphered with the respective private key. A private key (also known as a secret key) is shared only with key's initiator. By doing this, a message can be encrypted with the private key only by the author of the message and, therefore, implies the integrity of the message if it is possible to decypher it using the respective public key.

In the PIMCity project, the private key will be kept secret inside the P-CM component and the public key will be distributed to the P-DS to be able to check the integrity and non-repudiation of the tokens.

This is very convenient to us, since both components are part of the same infrastructure it will be easy to distribute public keys and, in case of any problem, generate and redistribute a new set of public keys (making all previous corrupted tokens invalid).

JWT still have a major disadvantage: they do not encrypt the content of the message. Any JWT content will only be base-64 encoded and therefore easily read by any man-in-the-middle. As of today we did not take this flaw into consideration as the connection between different components of PIMCity as not been introduced yet. If confidentiality of databuyers' request is necessary we might have to rethink how to behave with JWTs and how to distribute efficiently their token requests.

4.3 The development of databuyers' APIs

Not having an active connection with the P-CM component, we had to create an internal system to create sample JWTs as if the P-CM was generating it. Purely for development and testing purposes, we had to keep private and public keys in the same space, generating and signing sample tokens ourselves and then sending them to our APIs. We used python scripts and its JWT package for the generation of the tokens, then checked their validity in our APIs logic.

Databuyers' requests are sent as a POST request to the databuyers reserved url. The JWT token has to be passed as a POST request parameter. From here, the backend will check validity, extract information from the token, retrieve the data from the user safe and return it in JSON format to the databuyer. The procedure is displayed graphically in Figure 4.2.

There are some considerations to make regarding this procedure:

1. We are sure that we will find the user data because we are sure the token has been generated by the P-CM.

2. In case a user first grants access to data to a databuyer, those data should then be locked in the P-DS because a "contract" has been established between user and databuyer. We have not yet implemented this step as there is no active connection between components.

3. User data is returned, for now, as plaintext response to the POST request. This is not safe. As a future step, we should focus on securing these transactions, making it safe for information to travel across the internet.

We consider the overall structure of the databuyers' process to request data deployed. Lots of improvements are still necessary to make it operative and secure.

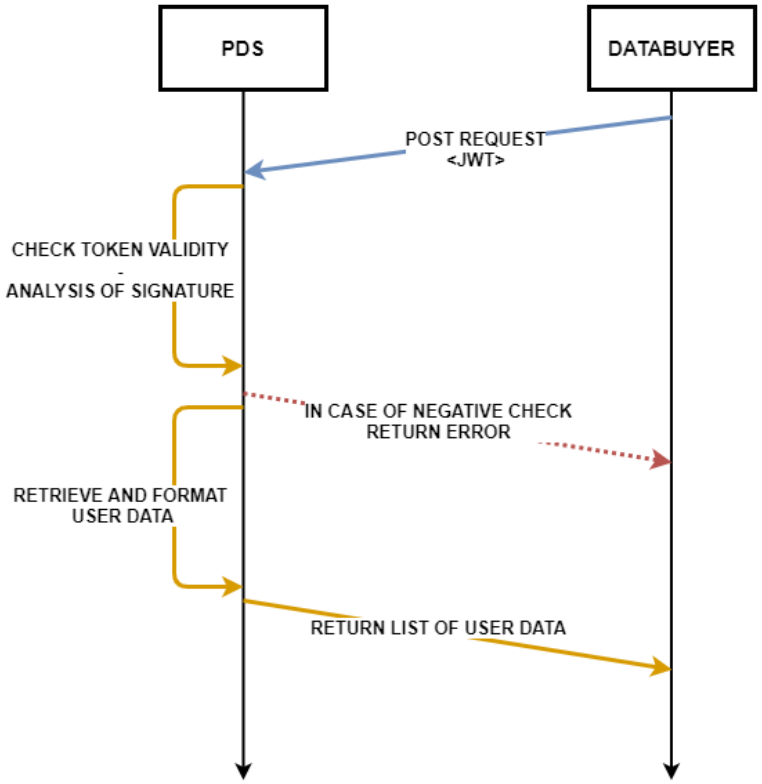


Figure 4.2: Databuyers APIs usage

The main aspects to concentrate from now on are those three mentioned above, and, in general, focus on security.

Chapter 5

Deployment, testing and next steps

In this section we will discuss the deployment of the application into the Polito servers, the usage of different VMs to deploy the components, the naming convention of the APIs and web APP paths applied by our internal proxy and more. We will also dedicate a section to performance and testing of the servers to see how many request can it sustain and other relevant metrics.

5.1 Deployment

Our P-DS is composed of 3 main components: a DJANGO application to distribute APIs, a POSTGRESQL database to store the data and a REACT server that distributes the JS application. With these 3 components in place, the P-DS is able to be fully functional, as described throughout this work. It is also possible to avoid using the REACT server and only have APIs working, this is the beauty of distributed production.

Nevertheless, we have put in place these three components in our Polito servers. We are handling the whole PIMCity system as a set of Virtual Machines, each of them hosting a specific component. This design will later allow for easier communication among components, making it simpler to set up a stable internal infrastructure. Once the components will all be connected, we will also be able to start working on some features that, at least for now, are only a prototype (such as the signed JWT tokens for databuyers' authorization).

In the P-DS virtual machine, we first set up the POSTGRESQL database as we will need to connect it to the settings of the DJANGO project to allow for connectivity and to perform the start-up checks. Then, next is the DJANGO

server. We can start it easily on whatever port we like as (it will be explained later) we have set in place a Reverse Proxy Server at the entry point of the PIMCity POLITO server. We opted for port 8000, but any other port will do (apart for well-known-ports).

Once both the DJANGO server and the POSTGRESQL database were up and active, we had to set-up the forwarding settings of the Reverse Proxy: we wanted to divide the paths for requesting the WEB APP "pages" of the REACT application from those paths requesting APIs. We also wanted the paths to be separated from those of other components, therefore the first step was to separate the P-DS paths from the other components. We simply decided to append "*/pds*" to the easypims url and redirect all */pds* requests to the selected VM.

https://easypims.pimcity-h2020.eu/pds

Now that our VM was set to receive all request with "*/pds*" appended, we could simply continue with the same concept to separate requests once more. All requests with "*/pds/api*" will be directed to the DJANGO server, onto the VM's port 8000, while all other requests will fall into the REACT APP.

It was now time to set-up the REACT server. As we are still in the development cycle, we thought it would not make sense to build the REACT project since it is still subject to frequent editing, we therefore kept using the npm command to run a development server on a specific port:

npm start

This command runs our application in development mode. If we were on a local environment, we could just navigate to *http:localhost:3000* inside a web browser to preview our app live, using the same concept, we redirect the requests arriving to the Proxy Server to the VM's port 3000 and redirect the response back to the user that made the request.

The REACT dev server is very useful as the page will automatically reload whenever it detects any code change in the source files, making it easier to make changes and perform bug fixes. Beware that warnings and errors can also be seen in the console, therefore once the project is ready for the public, we must build the project and have a server provide the built JS files upon request (which is what the dev server is doing on the background)

Finally, and not without many issues and troubleshooting, we have set up the whole environment, which is working correctly and delivering the correct WEB APP and APIs. For simplicity we report a graph to better display the whole configuration on Figure 5.1.

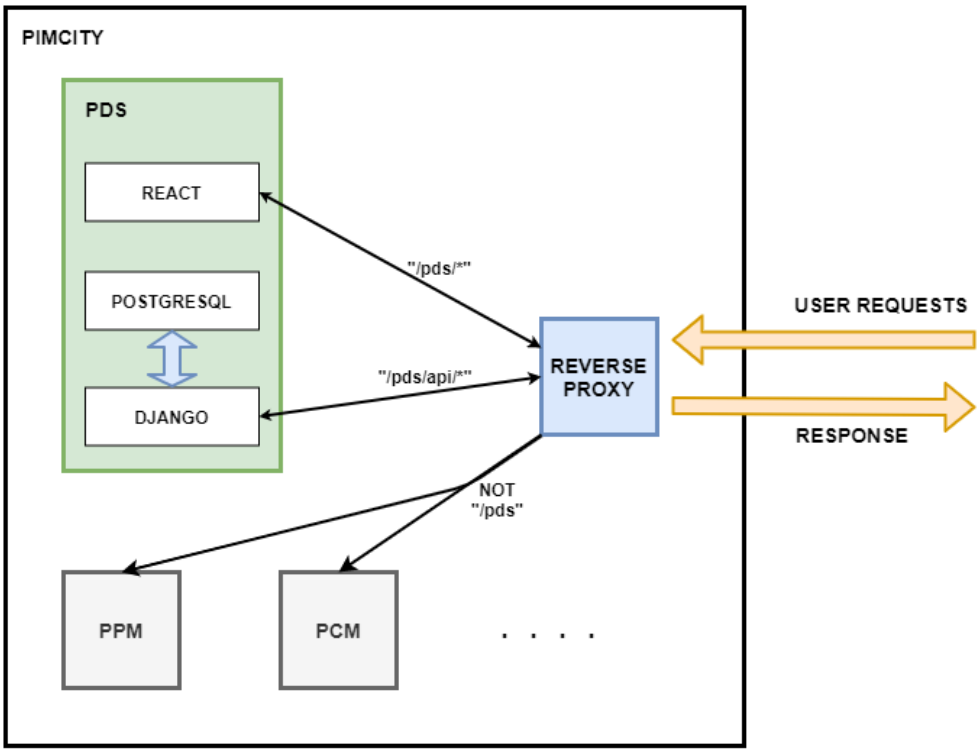


Figure 5.1: PIMCITY project server configuration

5.2 Testing

For the testing portion of the process we focused mainly on the stability of the servers. For the general code, some test have been developed in DJANGO, using its test system, leaving to future developers the ability to check if the code they will develop or modify is consistent with our build. Tests mostly focus on checking if the expected outputs of different views is still the same after the changes done to the code.

For the tests performed on the servers, we divided it into two categories: functionality tests and performance strength tests. The functionality tests are similar to the code tests, which is just checking if a specific call returns the expected output. Therefore a bunch of scripts will just request some content (mainly the DJANGO APIs content) and check if the returned values are the expected ones.

For performance test we mainly focused on stress testing the system. We wanted to be sure that the system can withstand a fair amount of consequent requests. The DJANGO server will instantiate a new DJANGO process, every time the server

is busy with a request and receives another one. Therefore we tried to send as many request as possible using a python script and monitor their response time. We mainly focused on normal requests and not on the heavy processing requests which are rare and very long (for example the upload data from google Takeouts files is fairly heavy).

We tested both the users' and the databuyers' APIs performance.

Users and databuyers testing

For the users' APIs, we tested the retrieval of information, because we believe that is going to be the most used request, since the REACT APP uses this API calls frequently. Therefore we created a script that tries to retrieve data continuously from a set of logged-in users. As a first analysis, we opted to perform requests asynchronously, and check, for each request the round-trip-time. We understand that it is not going to be a realistic scenario, but we wanted to avoid any queuing delays from the proxy server, which is the bottleneck of the whole system.
We analyzed the average, maximum and minimum response time for a set of various amount of stress tests requests, ranging from 100 requests to 10000 requests, while retrieving 1000 records of different categories (location history, browsing history, ...). We think that 1000 records is already a big amount of records to be transmitted in one request as, at least the React app, is built to retrieve a smaller amount "per page".

For the location history group, requests results are reported in Table 5.1 and displayed in Figure 5.2, all times are recorded in seconds:

Requests	Minimum	Average	Maximum	Time to complete test
100	0.149 s	0.087 s	0.072 s	8.683 s
1000	0.171 s	0.088 s	0.0720 s	88.129 s
10000	0.179 s	0.088 s	0.065 s	876.688 s

Table 5.1: Users requesting 1000 location history records to API

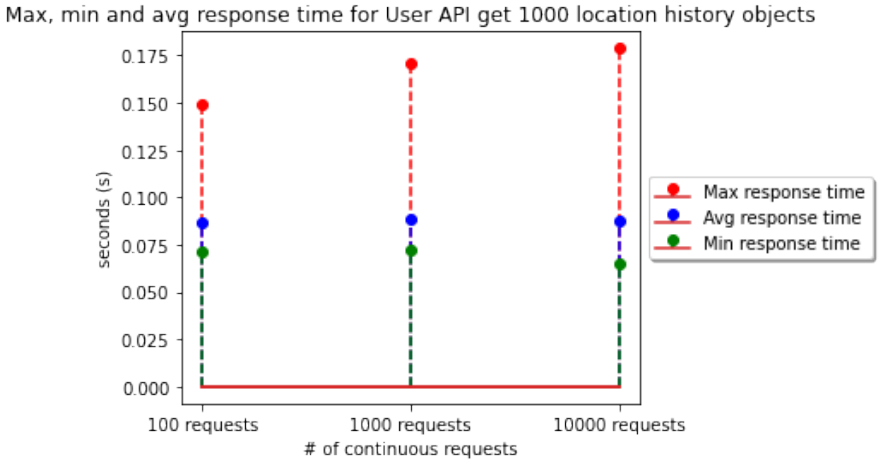


Figure 5.2: Users requesting 1000 location history records to API

For the browsing history group, requests results are reported in Table 5.2 and displayed in Figure 5.3, all times are recorded in seconds

Requests	Minimum	Average	Maximum	Time to complete test
100	0.0735 s	0.0861 s	0.156 s	8.615 s
1000	0.071 s	0.085 s	0.172 s	85.442 s
10000	0.067 s	0.085 s	0.168 s	850.318 s

Table 5.2: Users requesting 1000 browsing history records to API

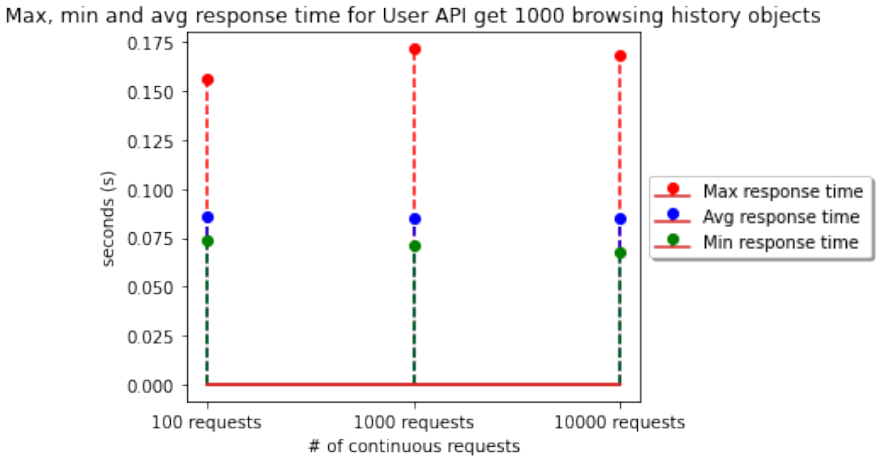


Figure 5.3: Users requesting 1000 browsing history records to API

And, in the end, we also wanted to test the databuyers APIs, as they are of central importance in the project. For this we reduced the number of requested records because each record has to be specifically referenced in the JWT token request, and therefore thousands of records at once are not the optimal way of using the databuyers APIs.

Requests results are reported in Table 5.3 and displayed in Figure 5.4, all times are recorded in seconds

Requests	Minimum	Average	Maximum	Time to complete test
100	0.022 s	0.031 s	0.063 s	3.052 s
1000	0.0196 s	0.031 s	0.065 s	30.618 s
10000	0.020 s	0.031 s	0.067 s	305.709 s

Table 5.3: Databuyers requesting users' records to API

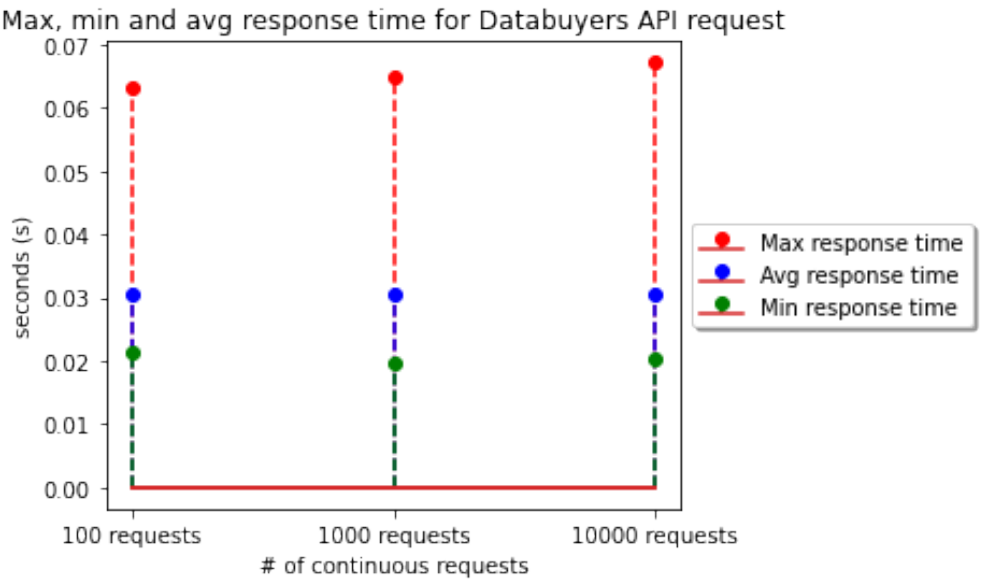


Figure 5.4: Databuyers requesting users' records to API

Analysis of results

As shown in the tables and graphs above, the system is resistant to bursts of requests performed by both users and databuyers. We consider the average RTT time to be solid and stable, and the maximum and minimum RTT to be very good as well. Of course these value represent a local testing environment, which is closed

with respect to the rest of the internet and therefore cannot be influenced by other factors such as processing, queuing, transmission and propagation delays. That being the case, anyway, there is not much we can do about it as we are not in control of the internet and all its users.

For a quick comparison we tested a utility API, which is also used by the REACT app, directly from the online server to see the average response time of it. With an average recorded value of 80 ms, we think that our results with the other requests previously analyzed could be averaging slightly more than this, and therefore being optimal for online usage. We can also observe that the size of the response is the main factor of RTT timings, as the databuyers responses, carrying much less data within them, are on average 50 to 100 ms faster than the user ones.

Considering the differences between minimums, averages and maximums of all the tests, we believe that the only factor affecting the recorded values is the amount of test performed, as the more tests are performed the higher is the chance to have a request that has slightly longer or shorter RTT. This is shown by the fact that the averages are very similar among them and that, with more requests performed in a single test, the maximum RTT increases but the minimum RTT decreases. This is what makes us think that the backend is resistant to long and persistent stream of requests.

We must consider that these are controlled and repetitive tests. Still we tried, as much as possible, to vary the type of requests (by requesting different sets of records in order to avoid that the database cache would influence the performance tests. Nevertheless, these test are not completely realistic and have to be analyzed for the information that they bring: the backend logic is resistant to a continuous stream of requests, coming from different users and requiring different sets of records. Moreover, it must be noticed that requesting 1000 records at the same time will not be the case for the average user of both the APIs and the REACT web platform, especially the second as 10 or 15 records per "page" are already hard to analyze.

During all tests, no failed request occurred which is a good sign that the logic is correctly working for a different variety of requests and that the backend is able to sustain the continuous requests' stream. It must be noticed that, for more extensive tests, which are tests that lasted hours, the access token expired and, without the logic for requesting a new one with the refresh token, these requests failed. We did not include these extensive tests in the graphs as they did not bring any more useful information to our study.

Before the full application will be operational and the development cycle will

be over, these tests should be performed onto the official server. Testing if the bottleneck of the system is the ability of processing request of the DJANGO backend or redirecting the request of the NGINX proxy server, this remains to be studied once the final server and systems are put in place. In my opinion, the bottleneck will be the proxy server as it will have to analyze and redirect the requests of all the PIMCity components at the same time.

5.3 Next Steps

With this new implementation of the P-DS, we set up solid foundations to a well developed distributed application. There are still many more features to add and functionalities to review and update. In this section we will go over all next steps that we think will be beneficial for the P-DS, and for PIMCity in general.

5.3.1 Security

On top of all, because of its importance in the project and the current status of the development implementation, we need to discuss security. Security should be implemented step by step, together with all new features and functionalities of the software development cycle. For most of the time, we tried to make our code and design as secure as possible, but the limited time and personal resources did not allow for a thorough security analysis. The implementation of JWT, for both user sessions and databuyers requests, surely has improved considerably the security of the data exchanges with the platform. Moreover, the addition of the proxy server covers and hides our real servers which can then be protected more with a firewall or other designs.

The APP is still vulnerable to DoS attacks as it has been shown from the stress tests, continuous requests from the same user are accepted (which should not be the case since a single user would not need to continuously request records). And this is only one example of the lack of cyber-security of the system.

5.3.2 Integration with other components

The P-DS is one of the central components of the system, and therefore it will need to communicate with other components to unlock some of its functionalities. One example might be the coordination with the P-CM, which will release JWT tokens to the databuyers so that they will be able to request users' information records.

As already discussed, from a development point of view we took care of this lack of interaction with other components, by simulating the behaviors of those

components from within the P-DS, an example would be the generation of private a public key for the release of "valid" databuyers token as a utility function of the P-DS. Obviously this will not be the case with the final build and therefore we believe that another important next step could be the implementation of the communication system from the inside of the PIMCity environment and among its components.

5.3.3 OAuth

One of the main changes we expect in the near future is the ability to log in via OAuth (Open Authorization) which is an open standard for access delegation. As previously discussed, we would like to delegate telco providers the ability to offer authentication to our users. This section of the next steps has already started and we are already trying to implement it with the REACT APP and the delivery of JWT session tokens for the backend. We are presenting it here because we think it is a very important step on both deployment and security of the final product.

5.3.4 Automation for CI/CD implementation

Now that the code is deployed into a running VM, once some changes need to be applied, we must be sure that these changes will work and will not freeze the operability of the machine. Moreover, we would like to not have to update the code manually each time a small change is developed. For these reasons, we think that an important step for the long-term deployment of the code would be to implement some sort of automation for CI/CD delivery. This can be done by using tools such as Jenkins or Ansible.

We leave the design of this for future developers.

5.3.5 Additional features for the REACT APP

At last, we have a bunch of suggestions from our testers on how to improve the usage of the Web APP to make it more user-friendly and accessible.

- **Filtering the records by title or dates**. This feature has been highly requested, and it is currently being developed. It will allow users to search and find records faster, and together with the ordering of records that is already in place will make the research even easier.
- **Filtering places directly on map**. The final goal of this feature is for the user to be able to click on the google map, then set a radius and display markers for places only within the radius limit from the point selected. The main problem with this work is the difficult interaction with the REACT component

that handles the Google Maps API. Since we used a third party component, it lacks specificity which would mean that, in order to complete and deploy this feature, we might have to write our own Google Maps component.

- **User Statistics**. We could also improve the visualization of the user data. For now, information is displayed in the homepage of the P-DS, but we could also add an extra view which is only dedicated to statistics.

Chapter 6

Conclusions

This work presented a development cycle of the P-DS component of the PIMCity project. It discussed design options, additional features, the implementation of a distributed architecture and its benefits and the development of APIs for both users and databuyers. It showed all the logical and implementation steps of the development of the component, analyzing strengths and weaknesses of the new choices adopted.

As a central ideology of the process, we discussed the importance of the distributed architecture, which we have seen been fully functional, bringing all the benefits of having the frontend completely decoupled from the backend. We are sure that the future of development will be much easier thanks to this division. Moreover, now, the backend system is completely autonomous and, being it the main building block of the whole application, easily improved.

We also discussed how the *schema.yaml* file is now controlling the functionalities and types of each personal information contained in the P-DS, both backend and frontend. We consider this a big step forward towards a fully functional SDK. Moreover, it provides easy modification of the application content and functionalities by both programmers and non-programmers.

A first addition of a security layer was needed, and achieved with the introduction of JWTs. Being them very customizable in content while keeping the benefits of signed asymmetric cryptography, were used in different sections of the APP, both for users and databuyers.

Some APIs were revisited and other new ones were added to the list. Some were strictly for the interaction with the frontend and others were purely thought for

backend. The introduction of databuyers APIs started the consideration of inter-component communication (e.g. with the P-CM for signed JWTs for databuyers).

Overall, we are happy with the final results obtained. The full application was uploaded on the PIMCity server hosted in Politecnico and it is currently working correctly. Local tests proved the stability of the final product and intensive user usage proved that the online version of the platform is providing the wanted results.

This project has been very useful for the development of new skills, the refinement of old ones and management of both time and resources. All troubleshooting, design thinking and communication with other peers helped develop a product that is getting closer to a final, business ready, well written piece of software. A platform that, we believe, will help hundreds of thousands of users to manage their sensitive data and decide where and to whom give their precious information.

Bibliography

[1] «Internet Live Stats». In: (2021). URL: https://www.internetlivestats.com/ (cit. on p. 1).

[2] «General Data Protection Regulation (website)». URL: https://gdpr-info.eu/ (cit. on p. 2).

[3] «EDPS PIMS description». In: (2020). URL: https://edps.europa.eu/ (cit. on p. 3).

[4] «PIMCity project description». In: (2019). URL: https://cordis.europa.eu/ (cit. on pp. 3, 35).

[5] Marco Mellia. «Pimcity: Building The Next Generation Personal Data Platforms». In: (Mar. 2020) (cit. on pp. 3, 4).

[6] Federico Torta. «Personal Data Safe: a flexible storage system for personal data». In: (2019/2020) (cit. on pp. 6, 8, 14, 31).

[7] «Django official website». URL: https://www.djangoproject.com/ (cit. on p. 6).

[8] «MongoDB official website». URL: https://www.mongodb.com/ (cit. on p. 6).

[9] «Bootstrap official website». URL: https://getbootstrap.com/ (cit. on pp. 6, 7).

[10] «Javascript Official Website». URL: https://javascript.info/intro (cit. on p. 12).

[11] «FreeCodeCamp server-side vs client-side rendering». URL: https://www.freecodecamp.org/news/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d/ (cit. on p. 12).

[12] «RestfulAPI.net». URL: https://restfulapi.net/ (cit. on p. 14).

[13] «Django REST framework». URL: https://www.django-rest-framework.org/ (cit. on pp. 14, 15).

[14] «React official website». URL: https://reactjs.org/ (cit. on p. 16).

[15] «AngularJS Vs. ReactJS Vs. VueJS: A Detailed Comparison». URL: https://dzone.com/articles/angularjs-vs-react-js-vs-vue-js-a-detailed-compari (cit. on p. 16).

[16] «Reasons to Choose AngularJs for Web App Development». URL: https://customerthink.com/reasons-to-choose-angularjs-for-web-app-development/ (cit. on p. 17).

[17] «GithHub Survey on most popular Js Frameworks». URL: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190 (cit. on p. 17).

[18] «Redux official website». URL: https://redux.js.org/ (cit. on p. 22).

[19] «SQL vs. NOSQL». URL: https://www.xplenty.com/blog/the-sql-vs-nosql-difference/ (cit. on p. 25).

[20] «Session vs Token Based Authentication». URL: https://sherryhsu.medium.com/session-vs-token-based-authentication-11a6c5ac45e4 (cit. on p. 28).

[21] «What is JSON Web Token?» URL: https://jwt.io/introduction (cit. on p. 29).

[22] «Search Security - TechTarget?» URL: https://searchsecurity.techtarget.com/ (cit. on p. 37).