POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master of Science in Computer Engineering

Master Degree Thesis

Personal Data Safe: a flexible storage system for personal data

Design and implementation of a data repository to empower user privacy in the age of GDPR

Supervisors
Prof. Marco Mellia
Dr. Martino Trevisan

Candidate
Federico Torta

Academic Year 2019-2020

Abstract

With the global diffusion of the Internet, the number of actions and operations performed online has increased exponentially: for example, e-commerce is the preferred means for shopping nowadays, while Google Maps seems to have definitely replaced the traditional paper maps. Behind all these useful services, organizations that own websites are involved in a profitable data market, where users' data is sold by publishers to vendors. In this way, data buyers try to understand users' behaviour in order to present them the most suitable ads according to their recent online activity. The publication of the GDPR in May 2018, however, focused the attention of companies and especially users on the possible security and privacy issues that can emerge while browsing the Internet. These new guidelines increased the concern about how user data is managed and define which users rights each organization must ensure. One of the first consequences has been small and slow changes in the scenario of data management, with websites trying to be as clear as possible on data collection and use. Novel systems for data protection and storage are becoming more popular as well. In this context, the so-called PDS (Personal Data Store) appeared, looking for providing a secure way to store people data.

In this work I present the design and the implementation details of the Personal Data Safe, a data storage system that has been developed as part of the PDS of PIMCity, a European project that aims to provide a standard definition of Personal Data Store and an easy modular way to build new ones out of basic building blocks. The PIMCity PDS groups together the main features of the major PDS on the market and tries to provide a development kit to let organizations define their own system, according to their needs and goals. The Personal Data Safe is one of these building blocks and it's the means for storing data. The system is built on top of the Django framework, using the Python language and manages to

offer a flexible environment where users can potentially store any kind of data; this is possible thanks to the presence of an administrator-created schema, that defines the configuration of the whole system. The final prototype has been designed also to be able to interact and connect with the other PIMCity components, which offer further functionalities to build a complete PDS.

Acknowledgements

I would like to thank my supervisors, Marco Mellia and Martino Trevisan, for the professionalism, availability and enthusiasm they put into the project.

A special thank goes to my parents, my brother, my family, my girlfriend and all my friends for the support in these years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The modern world is almost totally driven by data, thanks to Internet diffusion and accessibility. Every day, there are billions of people that are connected through each other, exchanging data of any nature, from complex structured information to simple text messages. Data is the main actor in the digital society and its importance has grown over the years, with the spread of the Internet and especially with the increase of commercial interests. Every action we perform online, in fact, brings with it a huge amount of information that can be used by companies and organizations: our last position tracked by the smartphone GPS can be used by Google to estimate the traffic in a specific city and in a specific moment of the day, while our browsing history can be useful to understand our interests or what we want to buy. This is why all web pages store information about their visitors, for example in the so-called cookies, which enable websites to understand the behaviour of their users, so that every single person can be presented a different page according to her interests and tastes.

If on the one hand, this kind of profiling seems to be quite common nowadays and useful in most cases, it could be a little disorienting or even scaring to know that each action we perform on the Internet is tracked both by little and big organizations which store information about us even for many years. Because of its ease of use, people tend to use Internet naively, without understanding that also a simple click can have big consequences: we don't know who's behind the pages we visit every day or how our data,

collected with or without our consent, will be handled. From the few past years, the concern about privacy and personal information usage started to grow, especially with the GDPR[1], a collection of data protection rules entered into force in May 2018. The main focus today is the definition of a set of guidelines and methods that can help each user regain control over his data, in order to reduce that sense of spying that someone can feel when browsing the Internet. For example, one of the goals of the GDPR is to enforce anonymization techniques, so that the data stored by web pages can not uniquely identify a specific user, or to increase people awareness on how their data are used so that they know what risks they are facing when performing a certain action on the Internet.

In this context, new systems called Personal Data Stores (PDS) or Personal Information Management Systems (PIMS), emerged, trying to provide a protected and GDPR-compliant environment to users and offer a secure way to store information along with methods to reduce privacy issues. Currently, the market of PDS is extremely heterogeneous, because there are no standard definitions for these types of systems; there exist both open-source and proprietary solutions, each of which offers different functionalities and different security methods. Despite the number of PDS implementations and the potential of such technology, none of them has yet reached business or technological maturity nor managed to attract a sizable user base. These issues are taken into consideration by PIMCity[2], an EU-funded project that aims to increase transparency and provide users control over their data. The PIMCity project tries to overcome the difficulties that characterize the PDS scenario and its main focus is in fact, provide a development kit to build the main components that can be found in all the existing PDSs, in order to define a general and standard structure for these systems. In the next chapters, I will present the prototype of a secure personal information store, called Personal Data Safe (P-DS), that represents one of the building blocks of the PIMCity PDS.

[1] General Data Protection Regulation

[2] https://pimcity.com/

Chapter 2

Background

Web sites, as well as companies and organizations behind them, have a great interest in collecting data about their users and visitors; data, in fact, represent the lifeblood of the digital world and have a strong connection with money and profits: a simple example is shown by Amazon that aims to keep track of the last researches in its web sites in order to present a custom *"You may be also interested in"* section for each user.

The collection of data by web pages is a very common scenario, but just recently people started to be concerned about profile tracking and especially about the means of this operation. The GDPR, for example, pointed out the possible risks and issues that can harm people privacy, so websites have started to become clearer on their data collection and usage policies. Systems for data protection and storage are becoming popular as well.

2.1 Current Scenario

2.1.1 User Tracking

Methods used to gather information about users fall under the definition of (web) tracking, that involves a set of different techniques. Although the possible differences, the common goal is the collection of data, whose types may be very different according to the purpose of each web page. In general, the information that could be worth to collect is:

- IP addresses to determine users' locations.

- information about users' interaction with the site, such as the buttons they click or the average time they spend on the same page.
- information about the browser or the device the users access the page with.
- browsing activity, that can give useful insight into users interests or shopping habits.

This information can be used for multiple reasons; for example, websites can use analytics software to gain data about their customers in order to make business decisions and optimize the website based on how visitors use it. Tracking is typically used also to improve website performance or to provide certain functionalities (e.g. the YouTube recommended videos section).
However, the commonest scenario is the one in which users information are exploited in the marketing and economic field, to provide targeted advertisements: data is collected by web pages and sites, then sold to other organizations; in this way, the so-called data buyers can display adverts for products users have recently viewed while browsing the web or adverts based on users searching history, locations or interests. More information will be presented in the section dedicated to real-time bidding.

Cookies

A possible way to implement user tracking is using cookies, a well-known argument nowadays thanks to the GDPR guidelines and the adoption of the 'Cookie Bar' by most sites. Cookies are small pieces of data that websites store on the user's devices and that can collect different types of information. They can be classified according to:

- **duration**
  - *session cookies*: they are temporary and expires when the session ends or the user closes the browser.
  - *persistent cookies*: they remain on the user hard drive until the user himself or the browser erases it. They are characterized by an expiration date.
- **provenance**

  - *first-party cookies*: they are put on the user device directly by the visited web site.
  - *third-party cookies*: they are placed by a third party, such as advertisers or an analytic system.

- **purpose**

  - *strictly necessary cookies*: they are necessary to exploit the features of the web site, such as secure areas. They are generally first-party session cookies. Cookies that allow webshops to hold user items in his cart while he is shopping online is an example of a strictly necessary cookie.
  - *preferences cookie*: they can be used by the web sites to remember choices and settings selected by the user, such as language or geographic region for weather reports.
  - *statistics cookies*: they collect data about how users use a web site; information is anonymized and aggregated so that they don't identify any user and can be used to improve the functionalities of the web site.
  - *marketing cookies*: they track users' online activity to help advertisers deliver more relevant advertising or to limit how many times users see an ad. Marketing cookies are generally persistent and third-parties.

Currently, the main concern and debate are on third-party, persistent and marketing cookies: they can collect a huge amount of information about habits, preferences and tastes of users and this kind of data is generally sold to companies that are interested in providing targeted advertisements. Being third-party, these cookies can collect data potentially for a big number of stakeholders and when information is sent to external companies, the chain of responsibility can become easily complex, leading to possible abuse of individuals information; the user in fact, despite being the rightful data owner, is generally cut out from this chain and he is not able to control who can access his data or how his data are handled. This loss of control influences user privacy since the collected information may uniquely identify a specific individual if anonymization techniques are not used. Furthermore, most of the times these tracking methods act silently

and can collect information without the users being aware, leading to even more privacy issues or personal information leakages.

2.1.2 Real Time Bidding

Data collection is just the first phase of the complex system that involves users' personal information. In fact, data obtained by websites are generally exploited in a buying and selling mechanism in which the vendor monetizes the information he has about his users and the buyer obtains the right to display ads on the vendor sites. All these operations define a structure called *Real Time Bidding (RTB)*.

RTB is a means by which advertising inventory is bought and sold on a per-impression basis, via instantaneous auction; whenever a user visits a website, an ad request and the corresponding bid request is fired through the *Ad Exchange*, the technology-driven platform that handles the RTB mechanism. The bid request incorporates all the user context data, such as demographical data, location information, browser history, etc. and it is sent to companies that are interested in buying an ad space; according to the individual data, each potential buyer makes a bid in the auction and space is sold to the highest bidder: if the real-time bid is won, the buyer's ad is instantly displayed on the publisher's site. The whole thing lasts a few hundred milliseconds and it's performed in a completely user-transparent way.

The RTB mechanism has therefore two main actors:

- **the advertiser**: it is the one that wants to public his ads on the spaces made available by publishers.
- **the publisher**: it is the one who makes available his "ad inventory", so his space for ads (e.g. YouTube).

The main issue is the fact that the user is not actively involved in the auction: he is unaware of how his data are being used and generally he is unaware even of the fact that his personal information is being sold to third-parties and unknown companies. This information trading is very complex to control or standardize and there is no way to avoid the auction participants from combining or aggregating users' data and sell them to other buyers. Users completely lose control over their data, in favour of third-parties' earnings.

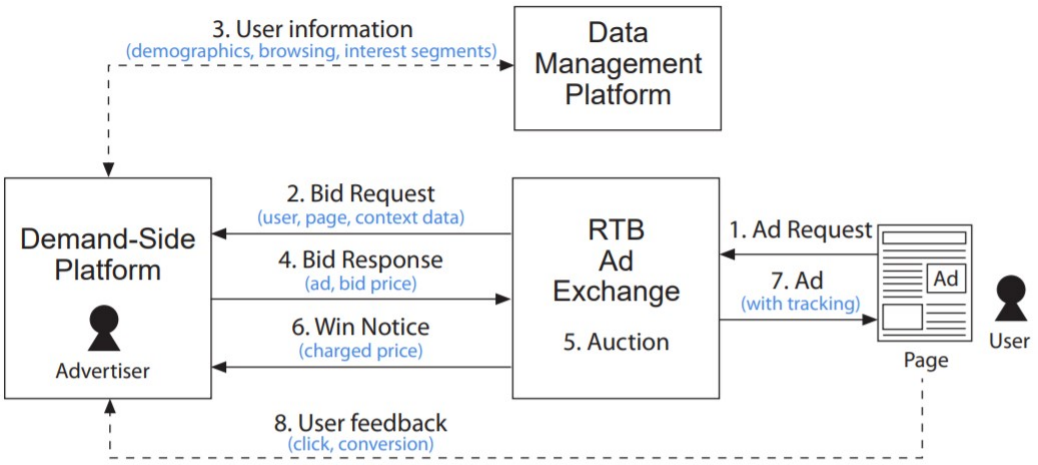


Figure 2.1: Interaction flow in Real Time Bidding

OpenRTB

One of the most used platforms for RTB is OpenRTB [10], created by IAB Technology Laboratory[3]. The OpenRTB protocol aims at spur growth in Real-Time Bidding marketplaces by providing open industry standards for communication between buyers of advertising and sellers of publisher inventory. The main goal is therefore standardizing the communication between parties during online biddings.
In OpenRTB, ad requests originate at publisher sites or applications; for each inbound ad request, bid requests are broadcast to bidders, responses are evaluated under prevailing auction rules, and a winner is selected. The winning bidder is notified of the auction win via a win notice. A loss notification is also available to inform the bidder of the reason their bid did not win.

The main focus of the OpenRTB project is on the interaction between the demand side and the supply side, while the specifications about personal information manipulation and management are not so clear or detailed. Recently, due to the GDPR, some flags have been introduced to be compliant with the new regulations on privacy and user consent. GDPR

[3] https://www.iab.com/

compliance will be a focal point of the OpenRTB 3.0 version.

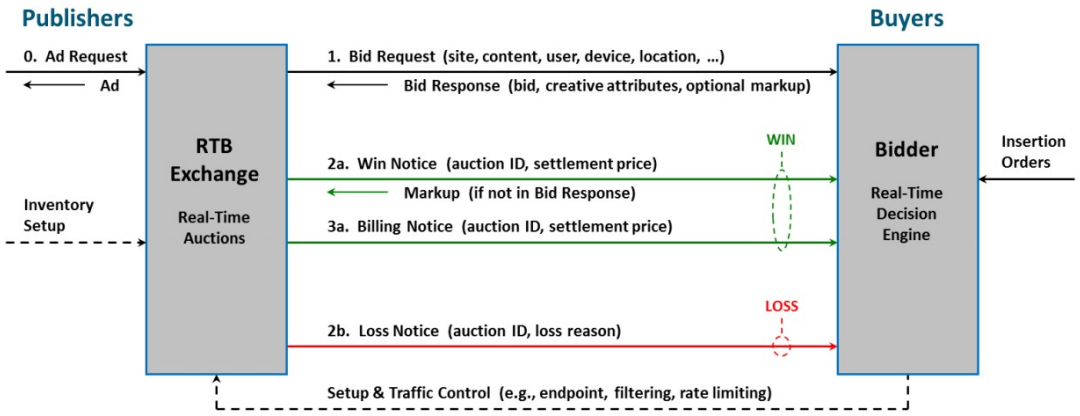


Figure 2.2: Main components involved in OpenRTB

2.1.3 GDPR

In order to have stricter control over usage and processing of users personal information, on May 25 2018, the European Union adopted the *General Data Protection Regulation (GDPR)*; it is a legal framework that requires businesses to protect the personal data and privacy of EU citizens for transactions that occur within EU member states. GDPR has been designed to harmonise data privacy laws across all of its members' countries as well as providing greater protection and rights to individuals. It has also been created to alter how businesses and other organisations can handle the information of those that interact with them.

The GDPR full text contains 99 individual articles, which replace the previous data protection directive of 1995. At the core of the GDPR, there are seven principles, which have been designed to guide how people's data can be handled:

- **lawfulness, fairness and transparency**: *lawfulness* states that all processes that in any way relate personal data of EU citizens must meet the requirements described in the GDPR, including data collection, data storing and data processing; *fairness* means that companies actions must match up with how it was described to data subjects, so

use personal data just for the purposes and during the time that the company indicated. Finally, *transparency* states that the data subject must stay informed regarding the purposes, the mean and the time of data processing.

- **purpose limitation**: companies must inform the data subject about the data collection and data can be collected and used only for those purposes.
- **data minimization**: this principle states that the collected data amount should be *adequate, relevant and limited to what is necessary in relation to the purpose for which they are processed*[1].
- **accuracy**: personal data must be *accurate and where necessary kept up to date*[1]. So, the company must avoid retaining old and outdated information.
- **storage limitations**: the company have to set the retention period for collected personal data and justify that this period is necessary for specific objectives.
- **integrity and confidentiality**: the company must enforce anonymization techniques to protect the identity of users, *against unlawful processing or accidental loss, destruction or damage*[1].
- **accountability**: every step of the data management needs to be carefully formulated and justified in the official document form.

While these seven principles define a set of rules and limitations for the ones that use data subjects' information, on the other hand, the GDPR is also designed to help to protect the rights of users. Eight rights for individuals have been defined:

- **right to be informed**: users must be informed about the collection and usage of their personal data and companies have to give specific privacy information about their business, data processing activities, length of time the company will keep the data, rights to lodge a complaint, etc.
- **right of access**: data subjects have the right to access to personal data. Companies must give users confirmation of whether they are

processing users data, supplementary information (including mandatory privacy information) and a copy of the personal data being processed.

- **right to rectification**: data subjects can ask data controllers to erase or rectify inaccurate or incomplete data.

- **the right to erasure**: users can ask companies to delete their personal data if data have been processed unlawfully, companies no longer need the data for the original purpose, the user withdrew the consent for data processing or erasure is necessary for compliance with other EU or national law.

- **right to restrict processing**: individuals can ask companies to restrict processing their personal data; in this case, companies are allowed to store the data but won't be able to carry out any processing.

- **right to data portability**: this right provides the data subject with the ability to ask for a transfer of his or her personal data. As part of such request, the data subject may ask for his or her personal data to be provided back (to him or her) or transferred to another controller.

- **right to object**: if companies rely on lawful bases of public interest or legitimate interests for processing, individuals may have a right to object to such processing. The objection has to be justified and can be made verbally or in writing.

- **rights around automated decision making and profiling**: the data subject shall have the right not to be subject to a decision based solely on automated processing, including profiling, which produces legal effects concerning him or her or similarly significantly affects him or her.

Despite being adopted from 2018, these guidelines and principles need time to widely consolidate. One of the immediate effects has been changes in user tracking consent mechanisms and personal data privacy policies of websites, especially in the context of cookies and purposes of user tracking.

GDPR and Cookies

The GDPR mentions only once the word cookie, but the impact is significant. As already said in the previous chapters, some cookies can collect data that may identify a specific user (e.g. IP addresses, email addresses): these kinds of data fall under the definition of personal information and so they are regulated by the GDPR. To be compliant, organizations will need to either stop collecting cookies or find a lawful ground to process that data. Most of them rely on consent (either implied or opt-out), but the GDPR's strengthened requirements mean it's much harder to obtain legal consent.

Moreover, cookies are regularized by the ePrivacy Directive, formally called the *Cookie Law*, that requires websites to ask *prior informed consent for storage or access to information stored on a user's terminal equipment*[4]. In other words, a website must ask the visitor to authorize the storage and retrieval of data sent through cookies and similar tracking mechanisms before delivering and installing them. This has become particularly evident in the last years with the proliferation of "Cookie Bars" on most websites. However, the results of all these regulations are weak yet and the majority of websites still collects data without users consent, as stated in [12], in which it is demonstrated that on more than 35,000 websites, almost half of them violates the Directive about the use of cookies.

2.2 Personal Data Storage System

While the GDPR defines guidelines and rules about personal data and their management, a great issue remains where to store this kind of information. Nowadays, it's normal the case in which users data are split between different companies: Facebook, for example, could keep the list of the last liked posts of a user, while Google could have a list of the recently visited places and Amazon of the last purchased items. Obviously, users have the right to retrieve their data whenever they want, but this fragmentation leads to a loss of control, because the user is not at the same time owner and keeper of his personal information, and he has to ask companies to get them. A possible solution to this problem is represented by emerging storage systems, called *Personal Information Management*

Systems (PIMS) or *Personal Data Stores (PDS)*.

2.2.1 PIMS and PDS

PIMSs and PDSs are storage systems that help users gather, store, manage, use and share personal information. They provide users with tools to control what information they share with which people or organizations and when, enabling individuals to handle their personal data in a highly secure and structured way. The core idea of PIMSs is to build a secure vault for users data, giving them the possibility to add new information, delete old or inaccurate ones, decide whether and with whom to share their data, for what purposes, for how long, etc.
In this way, PIMSs are trying to transform the current provider-centric scenario into one centred on individuals that are able to manage and control their data. The ongoing situation, in fact, can be considered provider-centric because the service providers are the ones that collect, store and handle users' personal information. They have to define a clear and public policy for data management, but actually, they are the ones that decide how to use the collected information, and more importantly how to monetize them. As explained in the real-time bidding section, the market of user data is very huge and it's completely transparent to the user himself, leading to a data processing mechanism that is unfair to individuals who earn nothing from the selling of their information. PIMSs are trying to reverse the trend, providing the users with a data store that allows them to handle their information as the actual and rightful owners.

2.2.2 PIMS Design

PIMSs are still at an early stage of development and there are no standard definitions of what a PIMS is. This is why existing solutions present differences in features and design:

- **Data Location:** a first main distinction can be made on the decision where to keep data. PIMSs can be defined as local stores, where personal data are kept in users' devices, or can be cloud-based, where information is handled by cloud-based service providers. In the latter case, data can be stored in a single place or may be split among different providers.

- **Data Processing:** data can be processed entirely inside the PIMS itself or can be securely transferred to the service provider, which can apply encryption mechanisms.

- **Data Security and Protection:** in the field of personal information storage, security is a major prerogative and involves encryption algorithms for data confidentiality and authenticity, authorisation mechanisms to control third-parties access to user data, data minimisation and anonymisation services to protect users privacy. PIMSs can adopt different levels of data security and can implement different solutions.

Although all the possible differences, PIMSs present some common features like fine-grained consent management for the release of personal data towards services, the ability to revoke permissions and data, the ability to negotiate and receive payments for the release of data, privacy-preserving release of aggregate analytics or raw data, dashboards for extracting knowledge and quantifications from one's own data. Moreover, PIMSs may support many of the data protection principles, tools and safeguards that are at the core of the GDPR. One of the objectives is to put users in control of their personal information: PIMSs can facilitate the users access to their data and the possibility to keep them up-to-date and accurate (thus enhancing the quality of data), enforcing the rights to access, rectification and data portability. PIMSs can then increase transparency and traceability: for example, by looking at their dashboard in their PIMS citizens could know whether their personal data have been transferred between two different public administrations. Furthermore, even when data are processed for a specific purpose based on another legal basis, PIMS can help individuals to effectively manage their consent for possible further utilisation for other purposes.

It seems clear that PIMSs can implement data protection laws at a practical level and in a GDPR-compliant way. The bigger challenge, however, remains the difficulty of penetrating a market dominated by online services based on business models and technical architectures where individuals are not in control of their data; a market that is well established today.

2.3 Our Goal

Since the GDPR came into force just in May 2018, it is still too early to consider the effects that these regulations may have had. Although, one of the first visible consequences of the GDPR has been an arms race by users who have started to massively use systems to protect their online privacy, e.g. tracking and advertising blockers. In response, services have attempted to bypass blocking using a variety of elaborate tracking techniques, and publishers have developed blocker detection technologies that redirect users to pay-walls.

In this scenario, PIMSs and PDSs started to emerge, trying to provide users with a storage system that could offer security features, giving (or returning) people their data ownership. A great issue, however, is the lack of standards and guidelines in the definition of such systems, which led different organizations to propose their solutions, with different technologies, algorithms and goals. The PIMCity project tries to overcome these problems, aiming to design, build and exploit a set of reusable, flexible components in the form of a *PIMS Development Kit (PDK)*. The PDK intends to make building new PIMS, and extending existing ones, easier, faster, and cheaper thanks to an open API, helping accelerate the developments towards finding the right PIMS for unblocking a fair and safe data economy.

One of the fundamental components of the PDK is the *Personal Data Safe (P-DS)*, that is the means to store personal data in a controlled form. The P-DS implements a safe repository for the user personal information and gives the possibility both to the user to save new data and to automatically import information as they are collected, for example by other services and applications. The goal of this work is defining a prototype of a flexible storage system, that can be easy to reproduce in order to build more complex PIMSs; this prototype has to be as much elastic as possible to be able to store any kind of information, from simple identity data (first name, last name) to more detailed information, such as browsing history, without being tied to a rigid schema or a hard-coded data pattern. The system should be also efficient and quite fast in order to have good performance in stress situations, like an increase in the number of users or the volume of data. More information about design and technical aspects will be presented in the following chapters.

Chapter 3

Related Works

As mentioned in the previous chapters, with GDPR new systems for personal information protection and storage appeared. Companies and organizations proposed over the years their own solutions for PIMSs and PDSs, without following any standard or common guidelines: this is why the world of PIMS is very heterogeneous and each solution presents differences from the others. The following sections aim to provide a brief description of some of the most common products that are available nowadays.

3.1 OpenPDS

OpenPDS[9] is an open-source project, implemented at the MIT. It is defined as a personal metadata management framework that allows individuals to collect, store, and give fine-grained access to their metadata to third-parties. It also introduces the SafeAnswer (SA) algorithm, a practical way of protecting the privacy of metadata at an individual level: for each application (e.g. Facebook, Foursquare, etc...) there is a SA module which is installed in the OpenPDS; it is a piece of code that uses the sensitive raw metadata to compute the relevant piece of information within the safe environment of the PDS. In this way only the processed result is returned to the third-party, that can access just aggregated or transformed information; in this way, the SA algorithm is able to protect the privacy of individuals.

OpenPDS is composed of three main elements:

- **data requester**: any application or web site that want to access user personal information.
- **PDS front-end**: the front-end is composed of several SA modules, one for application. All the SA access to the database must be authorized and each SA module executes inside a sandbox. Just the processed data (the so-called safe answer) leave the SA module, so third-parties get just the minimal amount of information they need and not additional metadata that could harm users' privacy.
- **database**: metadata are stored in a CouchDB database[4], a NoSQL store that provides built-in functionalities to reduce the dimensionality of the metadata.

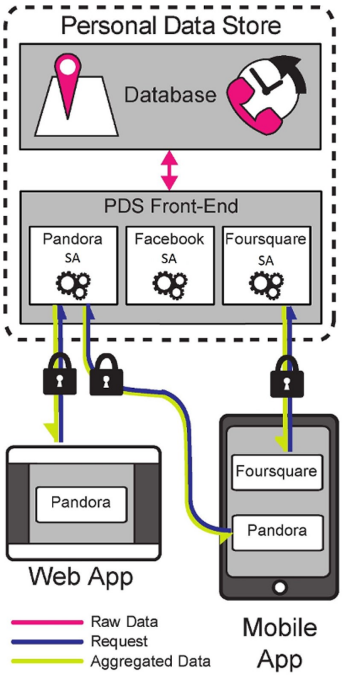


Figure 3.1: Main components of openPDS

The OpenPDS framework offers a set of functionalities to help app developers. Developers do not need to worry about the data collection phase: they can spend most of their time in defining the SA module for

[4] https://couchdb.apache.org/

their application while the PDS front-end takes care of creating the API and of securing the connections for them.

Although OpenPDS may be a potential standard solution for personal metadata management, it still faces several challenges: the automatic or semi-automatic validation of the processing done by a PDS module; the development of SafeAnswers privacy-preserving techniques at an individual level for high-dimensional and ever-evolving data; the development or adaptation of privacy-preserving data-mining algorithms to an ecosystem consisting of distributed PDSs; UIs allowing the user to better understand the risks associated with large-scale metadata and to monitor and visualize the metadata used by applications.

3.2 Mydex

Mydex[5] provides its users with the Mydex Trusted Framework and Platform, which enables to define single Personal Data Stores. Each PDS allows the collection, the management and the distribution of personal information, that are generally stored in the cloud, where the majority of the operations are performed. The Personal Data Store is an independent collection of files, that are uniquely encrypted using a private encryption key that only the individual has. Personal information can be accessed by the owner user and permissioned third-parties.

Client-side, Mydex offers a portable, privacy-friendly MydexID, a PDS and a set of tools completely free of charge. The user has full control over his data and can decide what kind of information insert in the PDS, whether he wants to change platform or which people/organizations/applications can access his data. PDS owners can decide to define *connections*, that enable to send and/or receive data to and/or from other people; in this way, the PDS can be set up to have connections with organizations, so that the user can send data to that organization or receive data from them. With connections, companies can still perform analysis on users' data, but in a way that is fairer for users themselves, since individuals have more control on who can access their data. Users, in fact, can also define *permissions*, which show which separate pieces of data the user is prepared to send to third-parties and what the user is willing to receive

[5] https://mydex.org/

from them. Permissions enable the data owner to have fine-grained access control, both in READ and WRITE mode: READ mode means that the user is sending data, so the connected organization can read a specific part of the PDS, while WRITE mode means that the user is willing to receive information from the connected organization and this will update a specific piece in the PDS.

Developer-side, the Mydex Platform offers a set of APIs that enable to create a secure connection between a user's PDS and a particular application and to exchange data, generally in JSON format. Mydex API offers:

- **Attribute Exchange Services:**
  - invite individuals to have a Mydex PDS and connect it to their records within an organization as a value-adding service.
  - send data to an individual's Mydex PDS with their permission but without their intervention.
  - secure access to verified data from an individual's PDS with their consent.
  - keep records about an individual up to date and get notified when any changes occur.
  - develop an application that runs on a smartphone, tablet, desktop or within a website that can access personal data from a Mydex member.
- **Identity Services:**
  - verify that a Mydex member is who they say they are.
  - use Mydex Identity Services to replace organizations' username and password management services.
  - add support for the MydexID into organizations' sign-in options on organizations' website.

Regarding security, personal data are encrypted using AES 512-bit encryption, while for communications information are always sent using 256-bit SSL. When users exchange data with organizations they have agreed to send or receive data from, it is encrypted using a one-time password which is delivered via an asymmetric key pair, which are unique to the

connection with that organization. Asymmetric keys enable data to be verified in terms of who the sender is; data encrypted with one key can only be decrypted by the other key.

3.3 Hub of All Things

Promoted by HCF, HAT Community Foundation, the *Hub of All Things (HAT)*[6] microserver is a scalable technology that confers legal rights of personal data to users through the ownership of a personal data server. The HAT microserver is hosted in the cloud and individuals can install plugins to bring their data in from the Internet, exchange data with applications and install tools in their microservers to have private analytics for insights into their data.

Hat microserver consists of four main components:

- **HAT Web Server:** each HAT microserver has its own API, which is configurable.
- **HAT Database:** users own a HAT Database and they have complete control over the contents of the database. Each HAT Database contains a data schema, allowing to store individual's data from any source without losing the structure specific to the source. The Database is characterized by *namespaces*, that identifies where an application will have read and write access, just like folders in normal operating systems.
- **File Storage System:** files are held in the S3[7] storage system offered by AWS[8] and managed by Dataswift.
- **HAT Computation:** this component enables the creation of private analytics. HAT Computation provides an environment where third-party code written in different languages can operate on HAT data.

[6] https://www.hubofallthings.com/

[7] Simple Storage Service by Amazon Web Service

[8] Amazon Web Service

As the rightful owner of the HAT Database, the user has complete control over the stored information and so can decide which organization can access a particular subset of the data. The core of the data exchange mechanism is represented by the *Data Debit* system, defined as the cornerstone of the permission-based, data contracts platform. Any data acquisition from the database require permission: when permission is granted, a legitimate contract is agreed upon. Once the data-sharing permissions are given, the contract between the HAT PDA (Personal Data Account) owner and the application provider is logged and Data Debits becomes the only way data can be retrieved from a HAT PDA by anyone other than the owner.

Chapter 4

P-DS Definition and Implementation Options

As part of the PIMCity group dedicated to the PDK[9] development, my objective has been the definition of a possible and plausible structure of the component dedicated to data storage, called Personal Data Safe (P-DS). Basically, the P-DS is the secure repository of a PIMCity PDS and the prototype has been designed in order to be quite simple to integrate with the other PDK parts, which are:

- **Personal Privacy Metrics**: this component collects, computes and shares easy to understand novel privacy metrics, providing fundamental information that can be used by users to make informed decisions.
- **Personal Consent Manager**: it is the means to define the users' privacy policies for consent management and the policies users desire to apply when sharing personal data with services.
- **Personal Privacy Preserving Analytics**: this element offers standard and open implementation of the fundamental methodologies that can be exploited to enforce data control and user privacy.

All the components must be able to communicate together in an efficient and simple way, so I chose to design the P-DS to support the REST[10]

[9] PIMS Development Kit

[10] REpresentational State Transfer

paradigm. REST is a common architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. The main functionalities of the P-DS can be exploited with a set of RESTful APIs that provide a simple interaction mechanism, especially between non-human systems.

During the P-DS definition, the requirement phase involved two main areas, the framework for the prototype and the database for the data storage. This chapter is dedicated to explaining the main technical and implementation choices that have been evaluated in the first months of the project.

4.1 Framework

As the starting point for the prototype implementation, I decided to leverage a web framework, in order to make easier the P-DS backend development. Today, web frameworks are very common and popular and they are used also for big and complex applications with thousands of users: they offer useful built-in functionalities (security, authentication, logging, auditing, etc.), can speed up the implementation process and they generally come with a middleware that connects the high-level programming language with the underlying storage layer. In the following sections the three principal frameworks that have been considered will be presented.

4.1.1 Django

Django[11] is a high-level, open-source, full-stack web framework written in Python and it is used by popular sites and applications such as Mozilla, Instagram, The Washington Post, and many others. Django was created by two web developers, Adrian Holovaty and Simon Willison working at the Lawrence Journal-World Newspaper in 2003. It was released publicly as a BSD license in July 2005.

[11] https://www.djangoproject.com/

Architecture

Django follows a Model-View-Controller (MVC) architecture that can be actually defined as MTV since the three main components are effectively named Model (same as in MVC), Template (that has the role of the MVC View) and View (the MVC Controller):

- data collected from the database is displayed by the **Template**; it includes a set of files that come from the combination of static HTML layout and Django syntax, which is essentially Python code, evaluated and processed server-side.
- the Django **View** instead is composed by a set of Python files that contain the business logic of the application and become the mediator between the model and the template. In the Django MTV architecture, the View is actually a little different from the Controller of the classical MVC design, since Django Views are only corresponding to a particular template and they are technically not selecting a model or view by themselves. The controller function is indeed held by the Django Framework itself, which is able to communicate with all the components. Views can be distinguished in class-based and function-based views.
- lastly, a **Model** in Django is a Python class that acts as the bridge between the database and the server. This class is a representation of the data structure used by the website and it can directly relate this data structure with the database.

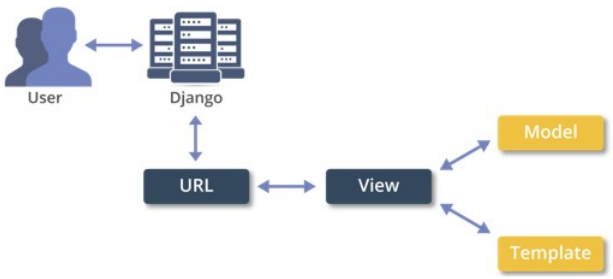


Figure 4.1: Django MVC design

This architecture in Django presents several advantages: first, it makes easier for multiple developers to work on different aspects of the same application simultaneously because the project is separated in different components; second, this architecture has different elements which require each other at certain parts of the application, at every instant, that increases the security of the overall website; finally, if there is a change in different components, the developer doesn't have to change it in other components.

Features

Being a big and complex framework, Django offers a huge set of functionalities. The framework, in fact, follows the *batteries included* philosophy and provides almost everything developers might want to do "out of the box": in this way common cross-cutting problems such as authentication, logging, etc., present an already-working implementation, helping developers in saving time. Some examples are:

- **ORM:** Django offers a robust ORM[12] system, which provides support for MySQL, Oracle, SQLite and PostgreSQL and makes easier the dialogue and the interaction with the database. ORMs define a middleware layer that closes the gap between high-level models and lower-level entities, which represent how data are physically stored in the database. In this way, all the operations that concern querying the database are simpler and the developer can focus on the core business of the application. The Django ORM automatically transforms all the Python instructions in SQL instructions.
- **admin panel:** it is a customizable administrative interface, that combines authentication, permissions levels and form data validation.
- **authentication:** Django offers an authentication system with permission levels and groups, completely customizable.

Behind this set of built-in functionalities, Django defines a highly secure environment, that helps developers by providing a framework that has been engineered to protect the website automatically. For example,

[12]Object to Relational Mapping

Django provides a secure way to manage user accounts and passwords, avoiding common mistakes like putting session information in cookies where it is vulnerable or directly storing passwords rather than a password hash. It also enables protection against many vulnerabilities by default, including SQL injection, cross-site scripting, cross-site request forgery and clickjacking.

Furthermore, Django is considered as a scalable, maintainable, portable framework. It is scalable because it uses a component-based *shared-nothing* architecture, where each part of the architecture is independent of the others and can hence be replaced or changed if needed. Having a clear separation between the different parts means that it can scale for increased traffic by adding hardware at any level: caching servers, database servers, or application servers. Maintainability is achieved with principles and patterns that encourage the creation of maintainable and reusable code. In particular, Django makes use of the Don't Repeat Yourself (DRY) principle so there is no unnecessary duplication, reducing the amount of code. Django also promotes the grouping of related functionality into reusable *applications* and, at a lower level, groups related code into modules. Finally, Django is written in Python, which runs on many platforms. That means that developers are not tied to any particular server platform, and can run applications on many flavours of Linux, Windows, and Mac OS X.

Project Structure

When creating a new project, Django automatically defines a root directory that contains some predefined files and folders. The root directory is the default app provided by Django and contains files that are used in maintaining the whole project. The main files created by the framework are:

- **manage.py:** this file is the command-line utility of the project and it is used only to deploy, debug and test with the project. The file contains the code for starting the server, migrating and controlling the project through command-line and provides all the functionality as with the django-admin. Some of the most common commands include:
  - *runserver:* it starts the test server provided by Django framework.

  - *makemigrations:* it is the command for integrating the project with files or apps developers have added in it. This command checks for any new additions in the project and then add that to the same.
  - *migrate:* this command actually add migrations developers made with the *makemigrations* command.

- **settings.py:** the settings.py is the main file where developers will be adding all their applications and middleware applications.

- **urls.py:** it contains the project level URL information and it connects the web-apps with the project.

- **wsgi.py:** this file refers to the fact that Django is based on python which uses WSGI server for web development.

4.1.2 Flask

Flask[13] is a lightweight and extensible Python web framework, which offers basic features for web application development. It keeps its core functionalities small, but it can be easily enhanced with extensions that can add application features as if they were implemented in Flask itself. Due to the minimalist architecture, Flask is generally considered a micro-framework, since it lacks most of the functionalities that are common in full-fledged web application frameworks, like Django. Among the big companies that use Flask, there are LinkedIn, Pinterest, Reddit, and more.

Flask is based on two main projects:

- **Werkzeug:** it is a utility library meant for usage with the Python language. Mostly, it is a Web Server Gateway Interface or WSGI app that can create software items for request, response, or utility functions.

- **Jinja:** it is a template engine for Python programming purpose and it could be compared to Django web frameworks templates.

[13] https://flask.palletsprojects.com/en/1.1.x

Features

The main goal of Flask is to keep the architecture as simple as possible, leaving all the additional features as external modules; Flask, in fact, is designed to be easy to use and extend and the main idea behind the framework is to build a solid foundation for web applications of different complexity. Developers are then free to plug in any extensions they need and are also free to build their own modules. Flask is great for all kinds of projects, especially for small ones or for prototyping tasks. However, if this simplicity requires more effort from the developer to define complete web applications, on the other side the overhead of executing a Flask project is much more reduced, leading it to be mostly faster than other frameworks. The main features of Flask include:

- built-in development server and fast debugger.
- integrated support for unit testing.
- RESTful request dispatching.
- Jinja2 templating.
- support for secure cookies (client-side sessions).

Flask allows to define MVC applications, but it requires more low-level operations, since the Model part of the MVC pattern, and so all the operations related to physical data and query translation, is not covered by the framework. Flask, in fact, is defined as ORM-agnostic and ORM support is not automatically provided but can be easily integrated with any project, as well as tools for handling migrations. In the context of security, Flask offers basic services, such as protection against XSS[14] in Jinja2 templates.

4.1.3 Spring

Spring[15] is an MVC web framework for Java-based enterprise applications, that enables to build applications from *plain old Java objects* (POJOs) and to apply enterprise services non-invasively to POJOs.

[14] Croos-Site Scripting

[15] https://spring.io/

Key Principles

The Spring framework is based on three key principles that try to solve and simplify one of the greatest issues of Java projects, that is dependencies. Traditional Java-based programs, in fact, are generally composed of two separate parts: a configuration one in which the developer defines and resolves the relationships between classes and an operational one, that contains the actual business logic of the program. This approach leads to classes that are strongly coupled and that are not able to work without the others since all the relations must be resolved before the real business logic is executed. Therefore, applying any change becomes really complex because relations cause a chain of variations in all the coupled classes; for the same reason the testing phase is complicated as well. Spring tries to overcome this problem with an automatic system that handles relations and couples Java classes; this system is based on the main principles of Spring:

- **Inversion of Control (IoC):** while in the normal case, a class A would have a direct dependency to class B, with IoC the dependency is removed with the introduction of an interface that abstracts the behaviour of class B. In this way the interface splits two types of responsibility: class A knows just *when* it needs the services, offered by the interface itself, while the real implementation of *what* is declared in the interface is provided by class B or any other class, even prototypes or test-only classes. IoC enables an easier way to perform tests because Java objects are loosely coupled and just need to know the interface's methods to work, while the class implementing the functions can be changed, modified or updated without any effect on other elements.

- **Dependency injection (DI):** DI is a programming pattern that realizes the coupling between objects at run time and not at compile time. All the dependencies are created and stored inside a container, that is handled directly by the framework itself; the developer just has to define in a declarative way (using annotations for example) the relationships between classes and then, at run time, the framework takes care of resolving them, injecting the dependencies as object properties. Spring offers a system for DI, called *autowiring* that provides a type-based injections mechanism.

- **Aspect Oriented Programming (AoP):** it is a programming model that enables to describe and define cross-cutting concerns, separating them from the application domain. Cross-cutting concerns, in fact, include those functionalities that are not specific to any particular application but are present in any project, such as logging, authentication, security. AoP allows the developer to define these functionalities in just a single place, using special classes called *aspects*. Tasks of each aspect can be applied in any part of the programs using annotations that modify the base class at loading time.

Architecture

The Spring framework architecture contains four main modules, which are the Core Container, the Data Access and Integration module, the Web Container and a Miscellaneous module. The **Core container** provides the fundamental parts and features of the framework, such as the IoC, the dependency injection mechanism, a sophisticated implementation of factory pattern called BeanFactory and the Application Context that has access rights for any objects that defines and configures. In Spring, Beans are the base classes that the developer defines for every single component.

The **Data Access/Integration** module includes a JDBC-abstraction layer, that removes the need for JDBC related code, and the Spring Object XML Mappers, a module that eases the mappings between Java object and XML documents. The Spring Framework doesn't have its own ORM implementation, but it offers the integration layers with other popular Object Relational Mapping tools such as iBATIS and Hibernate.

The last important Spring component is the **Web Container**, which contains several frameworks for developing web-related applications. For example, it offers the Web-MVC module, to develop MVC-compliant web application and the Web-Socket module, with WebSocket support.

Finally, Spring comes with more than 50 starter-packs in the form of POM[16] files, offering a collection of libraries that help developers in solving almost automatically common sub-problems and sub-tasks, like web programming, access to relational and non-relational databases, real-time messaging, security, caching.

[16]Project Object Model

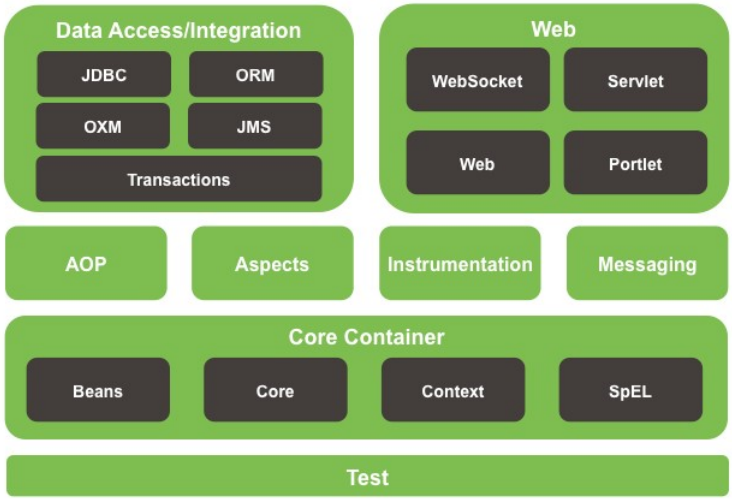


Figure 4.2: Spring Framework modules

4.2 Database

The second important aspect that has been taken into consideration during the design phase is the database. The Personal Data Safe, in fact, aims to be a secure repository for user personal information, so a weighted decision about what kind of data storage to use is fundamental. The debate has involved both relational and non-relational solutions: for the SQL world, I considered just MySQL, since almost all relational databases offer the same functionalities and features, while for the NoSQL field, two different products have been compared, because of the higher number of types of NoSQL databases.

4.2.1 Relational VS Non-Relational

Relational Systems

Relational databases are a well-consolidated technology, used in the majority of the applications since the 70s. Relational systems organize data in a structured way, using tables that are composed by rows (also called tuples) and columns. Information can be processed and manipulated using

a querying language called SQL[17], which provides the possibility to enforce business rules and constraints, ensuring data integrity during operations. A key concept in the relational model is the transaction, that is defined as a single logical unit of work, containing one or more SQL instructions; each transaction is an atomic unit, so the effects of all SQL instructions will be *commited* (made permanent in the DB) if everything succeeded or will be *rolled back* (deleted from the DB) if an error occurred. Transactions can be used to guarantee the so-called ACID properties which are fundamental in the relational world:

- **Atomicity:** all changes to data are performed as if they were a single operation.
- **Consistency:** data is in a consistent state when a transaction starts and when it ends, so the execution of a transaction preserves integrity constraints on data.
- **Isolation:** the execution of a transaction is independent of other transactions execution.
- **Durability:** after a transaction successfully completes, changes to data persist and are not undone, even in case of a system failure.

Relational databases require the definition of a strict schema before data insertion: each entry must fit the schema and this means that all the records will have the same set of fields and the same data type for each field. This aspect can be useful when the information that will be stored present a fixed structure that does not change much over the years, but it may be a problem for systems that need to evolve very quickly: the schema of relational databases, in fact, may introduce significant downtime if it is necessary to make changes that affect the whole database. Another possible drawback of relational databases is the fact that data need to be normalized before insertion, in order to avoid redundancies and anomalies during updates. Normalization requires the use of JOIN operations to connect tables and retrieve data: if tables contain a huge amount of entries, these operations could become expensive. A last possible disadvantage is the difficulty to scale horizontally: relational databases are easier to

[17]Structured Query Language

scale vertically, which is better for enforcing ACID properties, while in a horizontally-scaled environment it is way more complex.

Non-Relational Systems

The other half of the storage systems is represented by the NoSQL model. As the name suggests, NoSQL and non-relational databases abandon the use of the SQL language and prefer simpler APIs: in particular, JOIN operations are avoided since they may be too expensive and time-wasting, while custom APIs can lead to faster and more efficient systems. The other important innovation introduced by non-relational databases is the schemaless design: these products, in fact, enable the insertion and the processing of data that do not follow a fixed and rigid schema; every single record can have a different set of attributes and changes to the general schema can be applied without any service downtime. This aspect is very useful in the case of applications that need to evolve rapidly or that deal with mutable data. As opposed to relational databases, NoSQL systems offer native support to horizontal scalability (sharding), which make them particularly suitable for distributed environments, where adding new servers/machine is preferable than enhancing the hardware of a single machine.

On the other side, one of the main drawbacks of non-relational system is the fact that transactions are not supported: in a distributed environment it's more complex the enforcement of the ACID properties, due to the high number of nodes that have to be coordinated; instead NoSQL databases are subjected to the CAP theorem by Eric Brewer[7], which states that in distributed systems it's impossible to guarantee simultaneously the consistency, availability and partition tolerance properties. If the system is designed to be consistent and available, it will be not partition tolerant, while a not consistent system will be available and partition tolerant and so on. This leads to the possibility to ensure a different set of properties for NoSQL operations, the BASE properties, in contrast to the ACID ones:

- **Basic Available:** the system is available most of the times.
- **Soft State:** the system state can change over time even without any input.

- **Eventually Consistent:** the system will become consistent in a certain time interval during which it doesn't receive external inputs.

Non-relational databases can be divided into four different types, which share the mentioned-above characteristics but differ in the way data is stored and managed:

- **Document Store:** they enable to manage semi-structured data where each key is associated with a document, containing key-value pairs or nested objects.
- **Key-Value Store:** each element is stored as a key-value pair.
- **Column-Oriented:** data are stored in columns, to optimize the calculation of aggregates.
- **Graph Store:** relations between entity are represented with graphs.

4.2.2 MySQL

MySQL[18] is an open-source SQL database management system, developed, distributed, and supported by Oracle Corporation and it is one of the most popular and used RDBMS[19] on the market. Being a relational system, data is stored in tables, composed by rows and columns, which must follow a fixed schema that has to be defined before any insertion: the database administrator defines and runs the table initialization code, that creates the structure and the fields of each table.

Main features

As a relational database, MySQL supports transactions than enable to enforce strong consistency of data and provides several mechanisms for replication; replication allows the contents of a database to be copied (replicated) onto several computers: this can increase protection against system failure and improves the speed of database queries.

In terms of efficiency, MySQL is generally considered among the fasted RDBM, thanks to a large number of benchmark tests. Performance can

[18] https://dev.mysql.com/

[19] Relational DataBase Management System

either be improved in different ways, using for example memory caches or *indexes*, which are a data structure that improves the speed of operations in a table, avoiding scanning the whole table to find the relevant rows. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records. Indexes are completely transparent to users and are stored as a special type of table, which keeps primary key or index field and a pointer to each record into the actual table.

Finally, to ensure ACID properties, MySQL offers a locking system for tables and rows: it enables to define a tool for concurrency control for when multiple and concurrent queries need to access the same subset of data. Locks can be divided into two groups, that are related to separate operations, READ and WRITE. For WRITE operations, MySQL puts a write lock on the table, if there are no locks on the table, otherwise, it puts the lock request in the write lock queue. For READ operations, the working flow is very similar, but requests are placed in the read lock queue. When a lock is released, the lock is made available to the threads in the write lock queue, then to the threads in the read lock queue. In this way, MySQL can enforce the data consistency property.

Architecture

MySQL is defined as a client-server system, where the client establishes a connection with the server: first, client and server exchange their capabilities, then an SSL communication channel is set up if requested and lastly the client authenticates against the server. MySQL servers are composed of three major components:

- **Application Layer:** this layer is the topmost layer in MySQL architecture and it includes some of the services which are common to most of client-server applications; when the client connects to the server, the *Connection Handling* module creates a new thread for the connection and all the queries from that client are executed within that specified thread, which is cached so that it does not need to be created and destroyed at each new connection. At connections creation, the server performs authentication operations, based on the username, host of the client and password of the client user. After the client gets connected successfully to the MySQL server, the server will

check whether that particular client has the privileges to issue certain queries.

- **Server Layer:** this layer takes care of all the logical functionalities of the MySQL RDBMS. The server layer is divided into various subcomponents:
  - *MySQL Services and Utilities:* it provides the services and utilities for administration and maintenance of MySQL system, such as backup and recovery, security, replication, partitioning.
  - *SQL Interface:* it is a tool to interact between MySQL client user and server, offering Data Manipulating Language (DML), Data Definition Language (DDL), stored procedures, views and trigger.
  - *Parser:* MySQL parses queries to create an internal structure, called the parse tree. This module behaves as a single-pass compiler, providing support for lexical analysis and syntactic analysis.
  - *Optimizer:* after the creation of the internal parse tree, the optimizer applies a variety of optimization techniques that may include rewriting the query, order of scanning tables, choosing the right indexes to use.
  - *Cache and Buffers:* MySQL caches store complete results for SELECT statements. Before parsing the query, MySQL server consults the query cache and if any client issues a query that is identical to one already in the cache, the server simply skips the parsing, optimization and even execution, displaying the output from the cache.
- **Storage Engine Layer:** the pluggable storage engine feature makes the MySQL as a unique and preferred choice for most of the developers, allowing to choose among a variety of storage engines for different situations and requirements.

4.2.3 MongoDB

MongoDB[20] is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional

[20] https://www.mongodb.com/it

relational databases, MongoDB makes use of documents: a document is a single record in MongoDB and represents the basic data unit. They are analogous to JSON objects but exist in the database in a more type-rich format known as BSON. This provides MongoDB with a great level of flexibility: schema, in fact, is no longer required and developers can store any kind of data within the same collection, without worrying that documents may have a different structure. In this way, dynamic changes can be produced at any time and with no system downtimes. A set of related documents defines a collection, that is the equivalent of an RDBMS table. Generally, documents within a collection have different fields but share a similar or related purpose.

Main Features

Being a non-relational database, MongoDB refuses JOIN operations; relationships between documents can be defined in two different ways:

- **references:** in this case, document A contains a field whose content is the id of document B, which it is related with. This technique emulates a JOIN, but it's more inefficient because while JOIN operations require one single table access, references require at least a double access to the database.
- **nested documents:** the more used solution to represent relations is with nested documents; each MongoDB document key, in fact, can contain a nested object that is a complete document by its own. As a result, it requires a single database access to retrieve the information about the relation. The main disadvantage is the fact that this method can lead to data replication or denormalized databases.

MongoDB is designed to be able to scale horizontally in systems where the dataset is divided over multiple servers and system capacity is increased with the addition of new servers. In particular, MongoDB uses a specific method for distributing data across multiple machines, called *sharding*. A MongoDB sharded cluster consists of three main components:

- **Shard:** each shard contains a subset of the sharded data and can be deployed as a replica set.
- **Mongos:** mongos act as a query router, providing an interface between client applications and the sharded cluster.

- **Config Server:** config servers store metadata and configuration settings for the cluster.

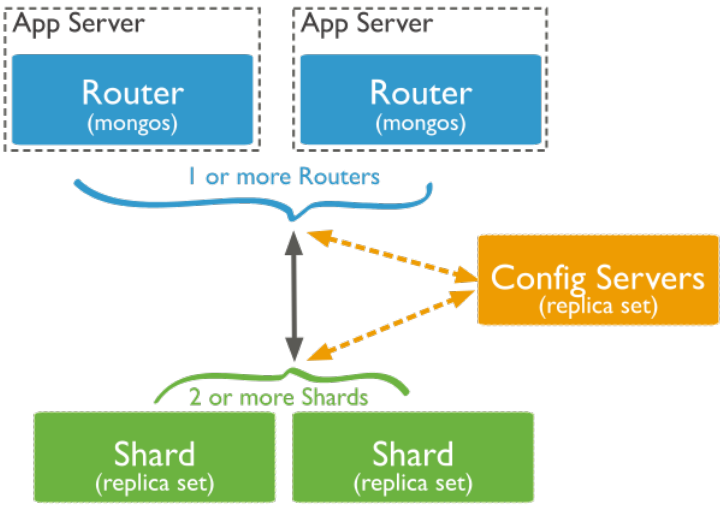


Figure 4.3: MongoDB architecture

Performance-wise, MongoDB is one of the highest performing databases, with the implementation of replication, indexing and load-balancing. Furthermore, it can handle large unstructured data, that allows users to query in different manners that are more sensitive to workload.

4.2.4 HBase

HBase[21] is a column-oriented non-relational database management system that runs on top of Hadoop Distributed File System (HDFS)[22]. The main difference between HBase and (non) relational databases is the fact that it's a columnar data store; with respect to row-oriented databases, HBase uses a completely different mechanism for storing data in disk, since information is stored by columns and not by rows. Physical organization of data has an impact on features such as partitioning, indexing, caching, views, etc. In this case, the column-oriented system optimizes operations

[21] https://hbase.apache.org/

[22] https://www.ibm.com/analytics/hadoop/hdfs

on data aggregation. HBase stores data in tables, which consist of rows identified by a RowKey and each row has a fixed number of column families. Each column family can contain a sparse number of columns.

As the other non-relational DBMS, also HBase exploits a schema-less design, where just column families must be defined and not a fixed column schema. It provides data replication across clusters for higher availability and it's linearly scalable. With respect to MongoDB, HBase offers a higher level of consistency, thanks to atomic read and write, on a row-level: during one read or write process, all other processes are prevented from performing any read or write operations.

Listing 4.1: row-oriented

```
1,Smith,Joe,40000;
2,Jones,Mary,50000;
3,Johnson,Cathy,44000;
```

Listing 4.2: column-oriented

```
1,2,3;
Smith,Jones,Johnson;
Joe,Mary,Cathy;
40000,50000,44000;
```

Architecture

HBase is composed of four main components which are:

- **HBase Regions:** they are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of Column families. It contains multiple stores, one for each column family.
- **HBase Regions Servers:** when region servers receive writes and read requests from the client, they assign the request to a specific region, where the actual column family resides.
- **HMaster:** it is the implementation of a Master server in HBase architecture. It acts as a monitoring agent to monitor all Region Server instances present in the cluster and acts as an interface for all the metadata changes. HMaster plays a vital role in terms of performance and maintaining nodes in the cluster, it provides admin performance, distributes services to different region servers and has

features like controlling load balancing and failover to handle the load over nodes.

- **ZooKeeper:** it is a centralized monitoring server which maintains configuration information and provides distributed synchronization. Distributed synchronization is to access the distributed applications running across the cluster with the responsibility of providing coordination services between nodes. If the client wants to communicate with regions, the server's client has to approach ZooKeeper first.

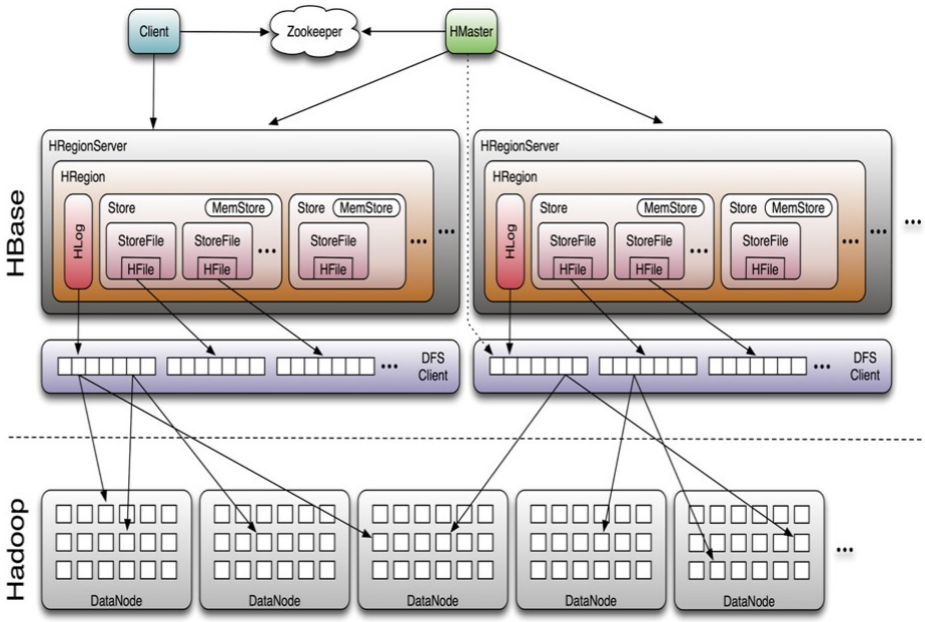


Figure 4.4: HBase architecture

4.3 Implementation Choices

After the evaluation phase, I opted for a Python web framework, since Python is easier to learn and to use, especially for beginners and has a learning curve that is less complex than the Java one, potentially leading to a higher level of productivity. Among the selected frameworks, I

decided to use Django, because it provides a larger set of built-in and already-tested functionalities, along with a wider online community that can supply ready-to-use solutions for common problems. Furthermore, Django offers a developer-friendly way to implement a set of RESTful APIs, using a few lines of code. In order to achieve the flexibility and the extensibility goals stated in Our Goal section, I decided to use a MongoDB database, which can offer an elastic environment that is able to evolve without being constrained to a rigid schema.

Technical information about the implementation of the P-DS prototype will be presented in the following chapter.

Table 4.1: Comparison table between Django, Flask and Spring

	Django	**Flask**	**Spring**
Framework Type	Full-stack web framework for Python	Lightweight and extensible Python web framework	MVC web framework for Java-based enterprise applications
Admin Interface	Built-in admin interface	Rely on a external libraries	Admin interface with Spring Boot Admin
ORM Usage	Robust ORM system, supports for MySQL, Oracle, SQLite and PostgreSQL	Does not have a built-in ORM system; supports SQLAlchemy and Peewee	Supports integration with Hibernate, Java Persistence API (JPA) and Java Data Objects (JDO) for resource management, data access object implementations
Built-in functionalities	Creation forms, data validation and CSRF token validation, authentication system with permission levels and groups	Relies on external libraries	Composed by different module: Relational Data Access, Non Relational Data Access, Security, ...
Security	Protection against XSS, CSRF, SQL injection, Clickjacking by default	Protection against XSS in Jinja2	Spring Security: authentication, authorization, protection against attacks like session fixation, clickjacking, cross site request forgery, ...
Community	Large community	Smaller community than Django	Smaller community than Django

Table 4.2: Comparison table between MySQL, MongoDB and HBase

	MySQL	**MongoDB**	**HBase**
Data Format	Data is stored in tables and rows	Data is stored in json documents that can be grouped in collections	Data is stored in column format
Schema	Predefined schema, tables must be defined before storing data	No defined schema, fields can be added on the fly	Schema-less design
Replication	Master-slave replication and master replication	Built-in replication, sharding, auto-elections	Master-slave replication
Index	If index is not defined, needs to scan the whole table to find relevant rows	If index is not found, every document must be scanned	Materialized views maintained by developers in code, or with coprocessors
Consistency	Immediate consistent	Eventually consistent and immediate consistent	Strong consistency at row level
Access Control	Users with fine-grained authorization concept	Access rights for user and roles	Access to data can be granted at a table or per column family basis
Best Scenario	When data security is an high priority	When most of the services are cloud based	When aggregation functions are relevant

Chapter 5

Development and Implementation

The development of the P-DS prototype can be divided into two main sections. The first one involves the definition of the backend design of the system, which includes the data model, the REST APIs that expose the P-DS functionalities and all the server-side utility functions. The backend elements have been implemented using version *3.8.2* of Python language, while for the Django framework I used the version *2.2.12*. The second section covers the frontend implementation: the P-DS prototype has been designed to provide a simple user interface, where the data owner can perform basic management actions on his stored personal information. For these elements, I mainly used HTML for the web pages skeletons, JavaScript to make the user interface dynamic and Bootstrap for style and appearance.

5.1 Database Setup

The backend of the prototype relies on a MongoDB database, running in a Docker[23] container on my personal computer. For the development phase, I decided to use a containerized version of MongoDB to have a high level of flexibility in the management of the database, with the possibility to easily create, stop, start database instances as needed. This elasticity is in

[23] https://www.docker.com/

fact, provided by Docker and by its lightweight virtualization mechanism that allows to set up even complex environments in few instructions; for all the P-DS implementation, in fact, I just used the following few Docker commands:

- *docker pull mongo:4.2.6*: the command downloads and starts a local instance of MongoDB with version *4.2.6*.
- *docker start [containerID]*: it starts the container with the specified ID.
- *docker stop [containerID]*: it stops the specified container.

5.2 Data Structure Design

Server-side high-level components can be grouped into three main macro-arguments:

- **Schema:** the P-DS configuration is based on a *schema*, 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 the project goals. The schema can be consulted by users, but its content is defined and managed by the system administrator.
- **Data model:** it includes the classes that I defined at source-code-level to represent the data domain. In particular two entities have been defined, the *User* class, that represents P-DS users, and the *PersonalInformation* class, which describes the principal characteristics of each P-DS entry.
- **REST APIs:** the P-DS functionalities about users and personal information are exposed as REST APIs to external modules, to have a simpler communication as possible.

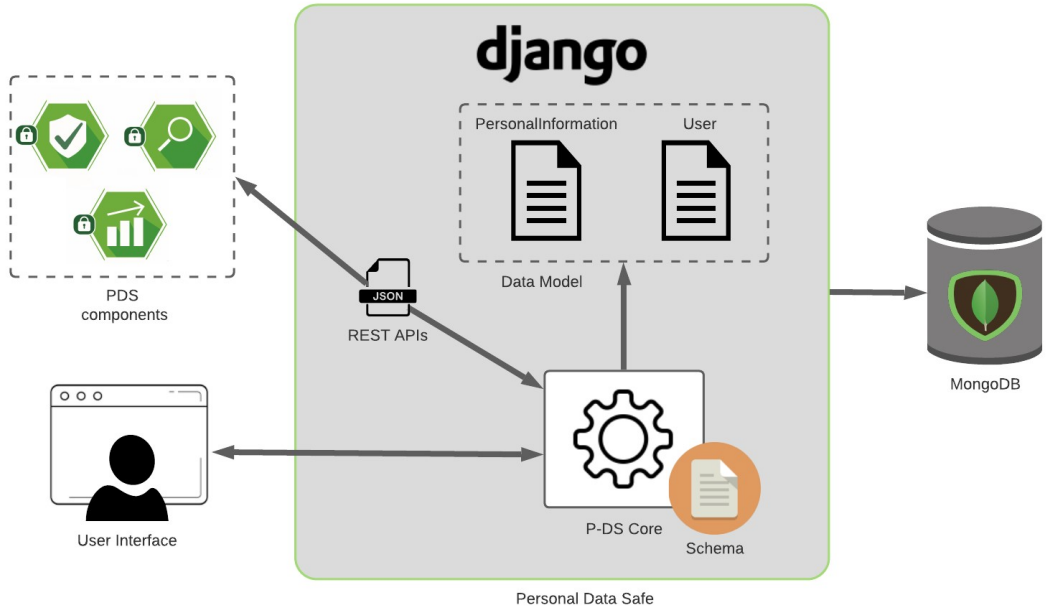


Figure 5.1: General structure of the Personal Data Safe

5.2.1 Schema

The schema represents the starting point of the P-DS structure. It is a YAML file defined by the system administrator and contains the main configuration of the data store, describing the kind of data that will be saved inside it. The main purpose of the schema is to provide a high level of flexibility in a controlled way. The P-DS in fact, need to be as elastic as possible and should be implemented in a way that does not force users to insert just certain types of data, with a given and fixed structure. Theoretically, it should be possible to insert in the Data Safe any kind of information, from common personal information, such as year of birth, cell phone number and email address, to more complex and structured data, like financial data, health-care information and so on. This precludes the possibility to have a limited set of accepted information with a fixed structure and specific fields because the flexibility goal imposes a general way to describe P-DS data. At the same time, it would be too

complex to handle a system, that gives the possibility to users to insert any information, with different types or a different number of fields. A form of control is necessary.

This is where the idea of the schema comes from. The configuration of a single P-DS is contained inside the schema file, that is loaded before actually starting the system and defines the data that is accepted by the Data Safe; in particular, the core of the schema is represented by a list, whose elements show the characteristics of each data type that the user can insert. 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.

Moreover, the schema can achieve the desired flexibility and extensibility goal, because it can be easily modified, enabling to have a P-DS that can potentially accept any kind of (validated) data. When users want to store new kind of information, the system admin just needs to add a new element to the *types* list and the Data Safe will be able to handle this new information. Also, the already existing group names can be updated, adding new features and actions: for example, for a particular group, users

can be given the possibility to add/update entries directly from the user interface, as well as upload a zip file for batch insertion or extract data in JSON format. The main idea is that the schema can be updated in any possible way, as long as it contains the entire configuration of the P-DS.

The schema includes just one hard-coded element, that is the possible values of the *type* field. For simplicity sake, I decided to make the P-DS support just a limited set of data types: *string*, *int*, *float*, *date*, *boolean* and *dict*. The dict type identifies a JSON-like object, called dictionary, composed of a list of key-value pair. This type has been included to handle more complex information that doesn't match the elementary types and need a set of sub-fields to be fully described. In the schema, the dict type is followed by the *fields* key, which lists all the sub-fields that are accepted for that P-DS entry.

Listing 5.1: Example of basic P-DS schema

```
name: ‘‘PIMCity default P-DS schema’’
version: 0.1
author: ‘‘John Doe’’
content:
- group-name: personal-information
types:
- name: first-name
type: string
- name: last-name
type: string
- name: birth-data
type: date
- group-name: browsing-history
types:
- name: visited-url
historical: true
type: dict
fields:
- url: string
- page-title: string
- time: date
```

5.2.2 Data Model

As explained in its dedicated subsection, the Django framework comes with an ORM system, that enables the developer to define high-level classes, called *models*, that logically describe how a single entity is defined; the main advantage of ORMs is the fact that all the low-level operations, such as database connection and queries, are automatically performed by the framework itself. Django does not provide a native ORM layer for MongoDB, so I decided to use the Djongo[24] connector. Djongo is an open-source project that provides Django users with a way to connect in an efficient way their applications with a MongoDB backend; Djongo makes zero changes to the existing Django ORM framework, which means unnecessary bugs and security vulnerabilities do not crop up. It simply translates a SQL query string into a MongoDB query document. As a result, all Django features, models, etc. work as-is. For the P-DS prototype, I used version *1.3.3* of Djongo.

Two main entities have been defined to describe the logical domain of the P-DS: the *PersonalInformation* model and the *User* model.

Personal Information

The *PersonalInformation* class constitutes the base for all the information stored in the P-DS. In order to have a system that is able to accept heterogeneous kinds of data, each *PersonalInformation* instance is basically defined as a type-value pair, where the value is the actual content of the information, with the declared type. In this way, the Data Safe can store any kind of data, as long the value is consistent with the type and the schema; the model, in fact, is provided with a *clean* method, that performs two main controls:

- **Schema Validation:** the possible data that can be inserted are limited to the schema definition; each inserted entry is compared with the schema content, to identify if the entry is compliant to the P-DS configuration. Besides the *value* and *type* field, the *PersonalInformation* class is defined with two additional fields, *group_name* and

[24] https://nesdis.github.io/djongo/

metadata, that can be used to control that the structure of the inserted information is accordant to what is declared in the schema. The *metadata* field represents just the common name that identifies a single entry in the P-DS, so it's equivalent to the *name* field of the schema, while the *group_name* determines the semantic set to which the entry belongs to.

- **Type Validation:** if the entry structure matches the schema, a second control must be performed to check if the declared type of the information really fits the actual one of the entry. The control is performed with a static cast: if the operation raises some errors or exceptions, the type validation fails and the information will be discarded because it is not compliant.

The *value* field of the PersonalInformation class exploits the JSONField class of Djongo: I decided to treat each information value as a JSON object in order to comply with how MongoDB stores natively data, so the JSON format. This solution is the cleanest one that I found and enables also to effortlessly process dict information since they are intrinsically in JSON format. For data that have an elementary type, the *value* field has a single subfield, where the key is the type of the inserted information.

The last important property of the *PersonalInformation* class is the *user* field, that represents the link with the user associated with that personal information instance. The relation between these two entities is defined in a SQL-like fashion, using a foreign key. Each personal information entry, therefore, is enriched with the reference to the user ID: in this way, no user information is duplicated in the personal information collection and any update on users are not reflected on personal information, since they know just their user id, which is stable and does not change. This solution exploits the *ForeignKey* field of Django and emulates a technique that is typical of the relational world, join. The decision of using this particular implementation to represent the relation between users and their personal information came out from the testing phase: this solution, in fact, proved to be the most efficient, because the embedded documents approach could let to store in memory huge volumes of information, while the reference approach performs poorly with high numbers for users or personal information, due to heavy lookup operation at database-level.

In any case, queries that filter data using the id field of *User* objects don't require JOIN operations, since each *PersonalInformation* document

stores just the id of the associated user; in this way filter queries can be performed efficiently.

Listing 5.2: PersonalInformation Class

```
class PersonalInformation(models.Model):
        value = models.JSONField(db_index=True)
        metadata = models.CharField(max_length=100, blank=True)
        type_ = models.CharField(max_length=100)
        group_name = models.CharField(max_length=100, db_index=True)
        description = models.TextField(blank=True)
        created = models.DateTimeField(editable=False)
        user = models.ForeignKey(
                User,
                db_index=True,
                blank=True,
                null=True,
                on_delete=models.SET_NULL
        )
```

User

The *User* class extends the *AbstractUser* class provided by the django-contrib-auth-models package, so it inherits some predefined fields, such as username and password, related to the authentication system of the framework. The inherited fields are used mostly by the authentication system offered by Django and provide information about the role of the user, his privileges, details about login and registration operations. The class *User* do not add any additional information related to the domain model of the Personal Data Safe. Each user is linked to a set of entries stored in the personal information collection, but all the information about this relationship are defined on the other side so in the *Personalinformation* class. The id of each user, in fact, is stored as a property in the personal information instances, using a *ForeignKey* field.

Listing 5.3: User Class

```
class User(AbstractUser):
        USER_CHOICES= [("ds", "DataSubject"),("da", "DataAggregator")]
        user_type= models.CharField(max_length=100, choices=USER_CHOICES
    , default="ds")
```

5.2.3 REST APIs

The P-DS functionalities about *PersonalInformation* and *User* entities are exposed to external components as REST APIs. REST is the acronym of REpresentational State Transfer and represents a programming style that enables the building of distributed systems, achieving properties such as scalability, the possibility of evolving, efficiency and resilience. REST paradigm is based on the following principles:

- **Client–server model:** the client-server architecture enables to separate user interface concerns from data storage concerns, improving the portability of the user interface across multiple platforms and improving the scalability of the server, whose components are simplified.
- **Stateless:** each request from clients to server must contain all of the information necessary to understand the request and cannot take advantage of any stored context on the server. All the information about sessions is stored client-side.
- **Cacheable:** cache constraints require that the data within a response to a request to be implicitly or explicitly labelled as cacheable or non-cacheable
- **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.
- **Layered system:** the layered system style allows an architecture to be composed of hierarchical layers.
- **Code on demand:** REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

In a REST architecture, the server is composed of one or more services. Each service manages a set of information, that have a globally unique name (URI), can have multiple equivalent representations (generally JSON) and support CRUD[25] operations. Data associated with a

[25]Create Read Delete Update

particular URL can refer to single object, collections (lists, sets) or functional operations results.

The main advantages of REST are the facts that it introduces a standard way to define distributed systems since URLs names follow a uniform convention and it fits well with the HTTP protocol, whose verbs support natively a CRUD environment:

- Create →POST.
- Read →GET.
- Update →PUT.
- Delete →DELETE.

In order to define a REST structure, the P-DS backend has been integrated with the version *3.11.0* of Django REST framework[26], which is a flexible toolkit for building REST-compliant Web APIs, based on Django models. DRF offers the developers three main objects:

- **Serializer:** they allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.
- **Viewset:** Django REST framework allows developers to combine the logic for a set of related views in a single class, called a ViewSet. A ViewSet class is simply a type of class-based View, that provides actions such as *.list()* and *.create()*.
- **Router:** DRF adds support for automatic URL routing to Django and provides a simple and consistent way of wiring view logic to a set of URLs.

In the P-DS prototype, I defined a PersonalInformation's *ModelViewSet*, which provides all the four CRUD operations for the class, while for the User entity I defined a *ReadOnlyModelViewSet*, that just enables read operations on User objects. This decision has been made considering that

[26] https://www.django-rest-framework.org/

the Data Safe is a system that is focused on data, so all the components that interact with the P-DS must have a simple way to interact with it and with PersonalInformation entities, thus leveraging a complete set of RESTful APIs, whereas for User instances all the CRUD operations may not be required.

Listing 5.4: PersonalInformation User ViewSet

```
1
2  class PersonalInfoViewSet(viewsets.ModelViewSet):
3
4      queryset = PersonalInformation.objects.all()
5      serializer_class = PersonalInformationSerializer
6      permission_classes = [permissions.IsAuthenticatedOrReadOnly]
7      authentication_classes =
8        [TokenAuthentication, BasicAuthentication, SessionAuthentication]
9
10     def perform_create(self, serializer):
11        serializer.user = self.request.user
12        serializer.save()
13
14     def get_queryset(self):
15        user_ = User.objects.get(self.request.user.username)
16        return PersonalInformation.objects.filter(user__id=user_.id)
17
18 class UserViewSet(viewsets.ReadOnlyModelViewSet):
19     queryset = User.objects.all()
20     queryset_ = User.objects.all()
21     serializer_class = UserSerializer
22     permission_classes =
23          [permissions.IsAuthenticatedOrReadOnly, IsOwner]
24     authentication_classes =
25       [TokenAuthentication, BasicAuthentication, SessionAuthentication]
26
27     def perform_create(self, serializer):
28        serializer.save()
29
30     def get_queryset(self):
31        queryset = self.queryset_.filter(username=self.request.user)
32        return queryset
```

Table 5.1: Summary of the used packages

Package	Version
Python	3.8.2
Django	2.2.12
djongo	1.3.3
pymongo	3.7.2
djangorestframework	3.11.0

5.3 Frontend

The second important element of the Personal Data Store is the frontend, which consists of the components the user can directly interact with. The P-DS in fact is the means by which the individual can store his personal information and it offers also a web interface, that enables to view, organize and possibly update or insert data. The user interface is composed of a set of HTML files, that leverage Django templates, the Bootstrap[27] library for the graphic aspect and JavaScript code for all the client-side dynamic parts.

5.3.1 User Interface

The P-DS user interface has the main goal to allow the user to have a global view on his stored data, with the possibility to perform common data management operations, such as insertion, update and deletion. In the view of defining a prototype of a possible Data Safe, I implemented a base version of a plausible user interface, which allows the P-DS owner to perform simple actions on his information. Some more advanced functions have been inserted, in order to show the potentialities of the P-DS, but they will be integrated and extended by the other components of the PIMCity project.

The core criteria that has been followed in the UI implementation is the

[27] https://getbootstrap.com/

flexibility property that dominates the entire prototype; the different interface components present the least possible number of hard-coded parts: all the elements have been designed to be as dynamic as possible, in order to have a structure that can evolve based on the P-DS content and the schema configuration.

The user interface can be divided into three main parts:

- **Authentication View:** the functionalities about security and authorization have been implemented using Django built-in features, whose end-points are automatically grouped under the */accounts/* URL. The login form is a standard Django form, which is enriched with some layout-oriented Bootstrap classes; if the authentication is successful, a User object is associated to each request to the server, so in this way, all the operations that can be performed are related to the specific user, avoiding unauthorized access to other people's data. I also inserted a basic form, that allows a user to reset their password by generating a one-time use link that can be used to reset the password and sending that link to the user's registered email address.

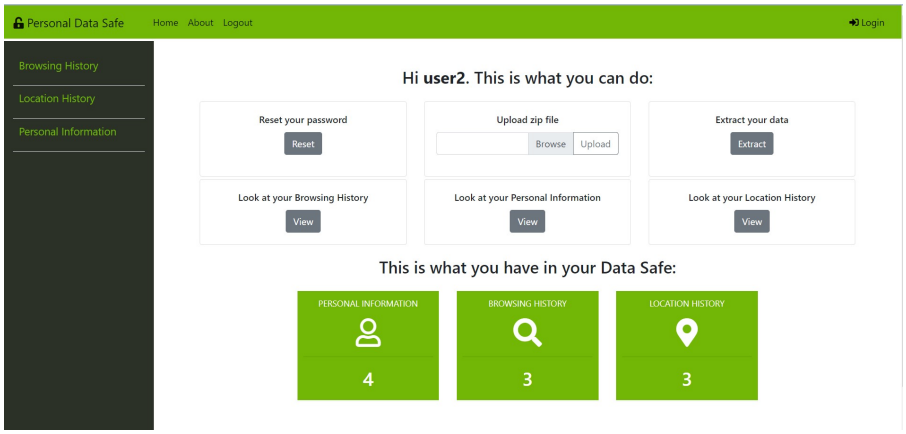
- **Home Page View:** the home page of the P-DS is composed of two elements. The upper one presents a summary of the possible actions the user can perform: some of them can be considered fixed for all the different P-DS, since they are cross-cutting operations, such as password reset, data import and data export, while the others enable the user to visualize his data; in particular, for each information group that the P-DS supports, I defined a dedicated button, so that the user home page can provide a more organized and structured view of the Data Safe content. These buttons are created at execution time according to the schema. The second element of the home page is the bottom one, which shows simple statistics about the data the user stored, divided per group. Under the hood all the elements that compose the home page are dynamically defined through JavaScript, allowing to retrieve the information and especially the schema structure from the server. In this way, the prototype can fit different P-DS configurations and can reflect its content without change the base components of the web pages.

- **Data Display View:** this part of the user interface permits the user to perform CRUD operations on the stored information. According

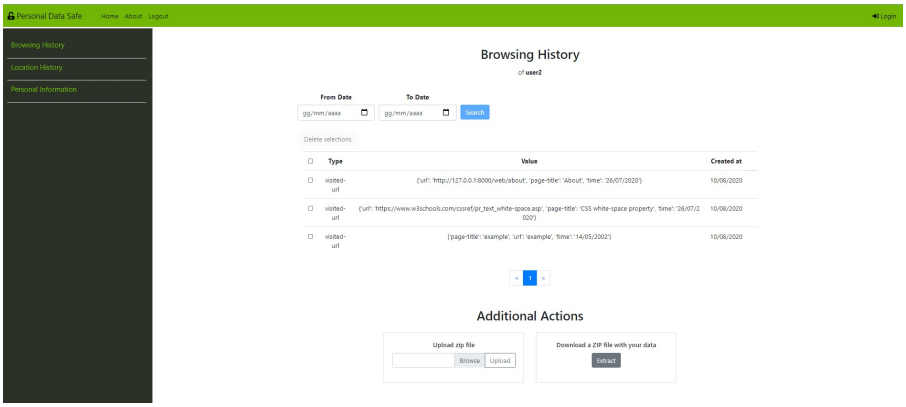
to the selected group, the view displays a table that lists all the data belonging to that specific group. If present in the schema group configuration, some possible additional actions can be executed:

- *add (update):* the add button open a pop-up that enables the user to insert a new data entry for the current group. In order to avoid insertions not compliant with the schema, a drop-down menu shows the possible types of data the user can insert. Controls on type consistency are then applied server-side. A general guideline for add, as well update, operations is to avoid the user deal directly with dictionaries, since people may not be accustomed to managing JSON-like objects. This kind of data will presumably be handled by automatic scripts or modules, that can build/update effortlessly JSON dictionaries.
  Similar is the update button, which allows the user just to change the value of the selected data, to minimize the possibility of errors.
- *delete:* for all the P-DS groups, the user can select a subset of information through checkboxes and delete it using a button, which opens a confirmation pop-up, aiming to avoid unwanted deletion.
- *data extraction:* since the GDPR states that each data subject has the right to get a copy of his data, for each group the user can extract a zip file, which contains a list of the stored information in JSON format.
- *data import:* some information types are suitable to be imported in the P-DS using an automatic loading system. For example, the user may want to load the browsing history he downloaded from Google Takeout directly in the Data Safe, without manually inserting entries one by one, or he would like to import his whole location history from Google Maps. This is why for certain groups (e.g. browsing-history, location-history), the page offers the possibility to upload a zip file containing the information the user wants to load automatically. The zip content is parsed as a normal information insertion, so control on schema and type compliance are executed.
- *filter:* it seems reasonable to let the user filter information, for example by date in case of entries with time information. Also, in

this case, the filter is displayed just if the schema contains some indication about it.



(a) Home page



(b) Data display page

Figure 5.2: P-DS views

For all the HTML pages, I used the template mechanism offered by Django, which offers an inheritance system for HTML files: I defined a base template that represents the user interface skeleton, which contains a toolbar with a navigation menu, buttons for authentication operations and a sidenav with the list of the data group supported by the P-DS. These elements are common to all the pages of the UI, while the central part of the skeleton is defined as a placeholder, that is replaced by the HTML pages that extend the base template.

5.3.2 Bootstrap and AJAX

The HTML pages that compose the frontend include HTML skeletons and JavaScript functions for all the client-side business logic. The graphic part of the user interface has been enriched with elements and classes from the Bootstrap library.

Bootstrap is a free and open-source front-end development framework for the creation of websites and web apps. The Bootstrap framework is built on HTML, CSS, and JavaScript to facilitate the development of responsive, mobile-first sites and apps. The framework is very simple to use, can be integrated into any project and provides ready-to-use responsive components. The main advantages of Bootstrap are the fact that it provides a grid system, which facilitates the organization of HTML elements inside the web page, and that it allows the developers to (almost) forget about style and CSS, enabling them to focus on the core of the web pages. In the P-DS implementation, I made a basic use of Bootstrap classes, in order to have a consistent style among all the pages, without losing too much time on graphic. The prototype can be extended in various ways, even exploiting more complex front-end frameworks, such as Angular.

On the other side, beyond all the style elements, each page of the user interface is characterized by some logic that is executed by the client itself and it's defined in JavaScript files. These files contain all the functions that support the creation of a dynamic UI: some of the elements of the pages, in fact, are not defined directly through HTML but are built using JavaScript, asking the server the needed information. In particular, the interaction with the server has been implemented using principally AJAX queries. AJAX[28] is a set of web development techniques using many web technologies on the client-side to create asynchronous web applications. It is not defined as a technology but as a group of inter-related technologies, which include:

- **HTML and CSS:** these technologies are used for displaying content and style. It is mainly used for presentation.
- **DOM:** it is used for dynamic display and interaction with data.
- **XML or JSON:** for carrying data to and from server.

[28] Asynchronous Javascript and XML

- **XMLHttpRequest:** for asynchronous communication between client and server.
- **JavaScript:** it is used to bring above technologies together.

The most important component of AJAX is the XMLHttpRequest, that is used for asynchronous communication between client and server: it sends data from the client in background, it receives the data from the server and updates the webpage without reloading it. The XMLHttpRequest object thus enables to make asynchronous the user interface, since requests are performed in background and don't block the client, as opposed to traditional synchronous applications. For each AJAX request, a callback is registered so that it is invoked when the server returns some data allowing the page to be updated. All the possibly expensive operations that involve the database are performed in background and so the user is able to continue using the web application. Most of the interaction between the P-DS user interface and the Django server is implemented using AJAX request: in this way the client can retrieve information from the server whenever they are needed without interrupting user interaction. This could be a great advantage in case of big volumes of data.

Chapter 6

Evaluation

Once the prototype of the Personal Data Safe has been defined, the last phase of the development process has been the testing phase. For web sites or generally web services, the response time of requests and system speed are crucial factors for the success of the application itself. The infographic compiled by OnlineGraduatePrograms.com for example shows that one in four people abandons surfing to a website if its page takes longer than four seconds to load. Moreover, people tolerance of slow webpage speeds has a huge impact on the possible earnings of a company. Amazon, in fact, calculated that a page load slowdown of just one second could cost it $1.6 billion in sales each year[8].

The main functionalities of the P-DS backend have been tested in order to understand the performance of the system in real situations, with different numbers of users and of personal information for each user (basically the total number of entries in the *PersonalInformation* collection). The following sections describe the tests that have been performed and show the obtained results through boxplots. A boxplot is a method for graphically depicting groups of numerical data through their quartiles. Boxplots may also have lines extending from the boxes (whiskers) indicating variability outside the upper and lower quartiles. The box of the graph generally is delimited by the first and third quartiles, while whiskers have been chosen to limit values between the 5th and the 95th percentile.

6.1 Test

Considering the main actions that a normal user can potentially perform, I decide to perform tests on basic READ and WRITE operations. In particular:

- for read operations, I considered a plausible query that users, or even external companies, can perform on stored data, that is filtering by URL name. The P-DS server exposes an URL that is associated with a function that filters the personal information of type *browsing-history* of the current user considering a specific URL, passed as a request parameter. If the entry contains the given URL, the personal information is retrieved and sent back to the one that performed the request. Also, in this case, the query has been chosen considering that *browsing-history* may be a common information type in most Personal Data Safe and that filtering on URLs could be a frequent operation, especially in the case of user behaviour understanding and targeted advertisement.

- for insertion, I considered the endpoint *rest/personal-information/*, which enables to add a new entry associated with the current user, through a POST request. The main reason why I chose this particular endpoint is the fact that it could be exploited both by users, directly from the user interface and by software systems or PDS modules, that can automatically perform insertions using the REST APIs. The insertion of new personal information in the repository represents therefore a crucial operation that has to be as much efficient as possible.

Tests have been applied in different situations, according to two main dimensions, number of users and number of personal information for each user:

- **Number of users:** the system should be resilient in case of huge numbers of users. Insertions and read operations have been executed in different conditions, with a variable number of users. In particular, I defined three scenarios, one with 100 users, one with 5,000 users and the last one with 10,000 users. In all the cases, each user has been provided with 1,000 personal information.

- **Number of personal information:** the second dimension considers the fact that the number of entries for each user can grow, even rapidly if we consider information types such as browsing history or location history, whose volumes may be huge. In this case, I considered three situations, where the number of users remains stable (500), while the number of personal information varies between 100, 5,000 and 10,000 entries for each user.

For both dimensions, the test databases have been populated with *browsing-history* entries of type *visited-url*, whose *value.url* field has been chosen randomly in a dictionary of 500 possible URL names. For each scenario, I performed the requests 1000 times for read operations and 500 times for write operations, to have enough data to define a statistical distribution and draw related conclusions.

6.2 Benchmark

6.2.1 Filter operation

As stated in the previous section, the read operations that have been taken in consideration perform a simple query on user data, filtering the personal information of type *browsing-history* that contains a particular URL name, passed as an argument of the request. Considering the data model, two fields of the *PersonalInformation* class are crucial in running this query:

- **the *user* field:** this property represents the link between a personal information entry and a user in the database, through the id of the user. If the number of users in the system is huge, this field can have a big impact on query performance; the filter condition on the user should be as efficient as possible so that the system can rapidly discard entries that are not associated to the desired user.

- **the *value* field:** this field represents the content of the personal information entry. In the case of *browsing-history* information, it contains the URL value on which the filter is executed. The system should have an efficient way to access this field in order to fetch just the needed entries in a reasonable time.

Due to these two main considerations, I decided to add two indexes in the *personal-information* collection. Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. They store the value of a specific field or set of fields, ordered by the value of the field. The main advantage is that indexes support the efficient execution of queries in MongoDB. Without indexes, in fact, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. Needed indexes have been created directly from the shell of MongoDB, inside the docker container.

Test scenario: increasing number of personal information

For the first test scenario, I considered a situation in which the number of users is fixed, but the number of personal information for each user increases on a log scale. Queries to retrieve browsing-history entries matching a given URL name have been performed in three different setups:

- database with 500 users and 100 personal information per user.
- database with 500 users and 5,000 personal information per user.
- database with 500 users and 10,000 personal information per user.

For each setup, the GET request has been executed 1,000 times and timings of the *curl* command have been recorded. Figure 6.1 summarizes the results of the filter test.

Considering the figure, I can say that for all the possible scenarios the median is approximately the same, with a value that is around 493 ms. Since the median is in the centre of the boxes, data is not skewed and the reduced dimensions of the boxes tell us that the collected data are condensed: 50% of response time varies between 492 ms and 494 ms. The reduced dispersion of data can explain the high number of outliers for the second scenario (500 users with 5,000 personal information each): outliers in fact don't differ much from values in the box plot, but since variability is very narrow even few microseconds can transform a value in an outlier. Minimum response times range between 487 ms and 490 ms, while maximum values between 496 ms and 499 ms.

Collected results state that the system performs query operations efficiently, with an average response time below 500 ms and queries seem

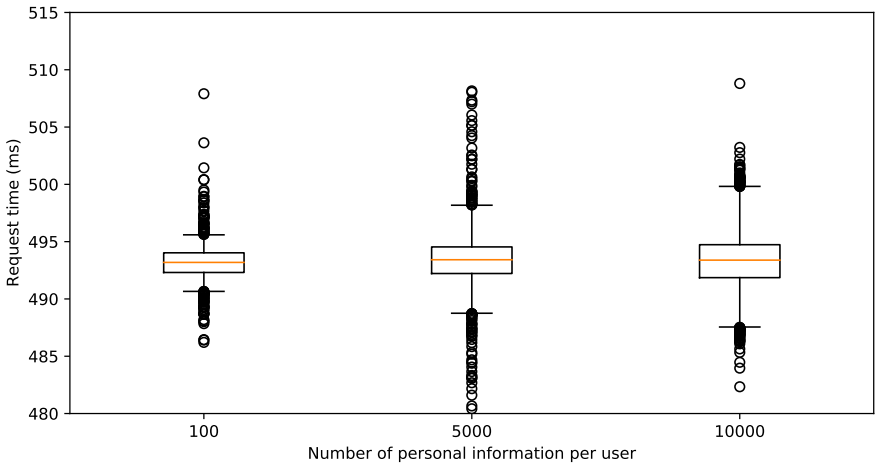


Figure 6.1: Results of URL filter tests with increasing number of personal information per user

to be unaffected by the chosen number of users personal information. To understand the potentiality of the system, I then decided to perform requests in a stress situation, where each test user is associated with 100,000 entries: response time increases up to 1.2 seconds. Therefore, in this case, the number of personal information per users has a bigger influence on system performance.

It's necessary to say that with an increasing volume of the stored personal information, the effectiveness of index becomes fundamental. First tests, in fact, have been performed on an indexless database and performance of query resulted very poor. The introduction of indexes enabled to pass from seconds-lasting operations to query executed in few milliseconds.

	5th percentile	**25th percentile**	**median**	**75th percentile**	**95th percentile**
setup 1 (100)	490.7 ms	492.3 ms	493.2 ms	494.0 ms	495.6 ms
setup 2 (5,000)	488.8 ms	492.2 ms	493.4 ms	494.5 ms	498.2 ms
setup 3 (10,000)	487.6 ms	491.9 ms	493.4 ms	494.7 ms	499.8 ms

Table 6.1: Results of filter tests with increasing number of personal information

Test scenario: increasing number of users

For this second scenario I considered the other major dimensions, so the number of users. Unlike the previous configuration, in this case, I decided to vary the number of users, while keeping fixed the number of personal information per user. Three different setups have been defined:

- database with 100 users and 1,000 personal information per user.
- database with 5,000 users and 1,000 personal information per user.
- database with 10,000 users and 1,000 personal information per user.

Also for this scenario, I performed URL filter requests 1,000 times for each configuration. Results are presented in the following figure.

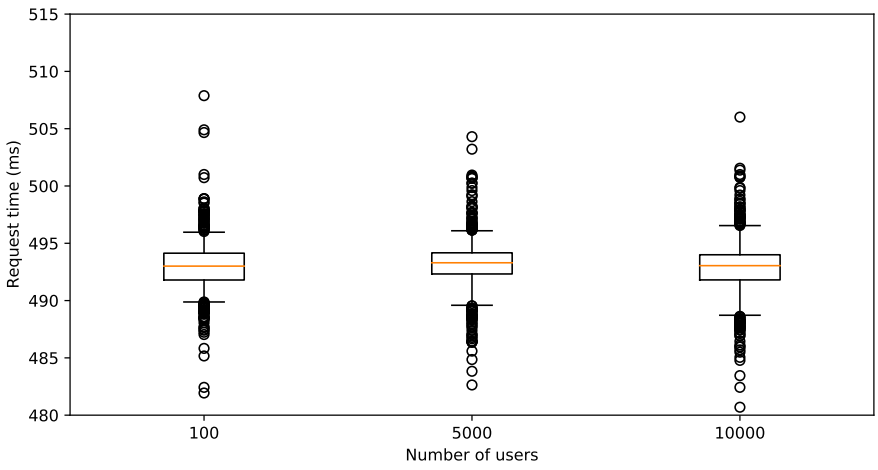


Figure 6.2: Results of URL filter tests with increasing number of users

Figure 6.2 shows that the chosen values for users numbers do not affect much the response time of requests, that present an average time of 493 ms for all the setups. All the different configurations present a minimum value around 489 ms and a maximum value around 497 ms and the 50% of data falls between 492 ms and 494 ms. For the third configuration, the boxplot displays slightly negative skewed data.

	5th percentile	25th percentile	median	75th percentile	95th percentile
setup 1 (100)	489.9 ms	491.8 ms	493.0 ms	494.1 ms	496.0 ms
setup 2 (5,000)	489.6 ms	492.3 ms	493.3 ms	494.2 ms	496.0 ms
setup 3 (10,000)	488.7 ms	491.8 ms	493.0 ms	493.9 ms	496.5 ms

Table 6.2: Results of URL filter tests with increasing number of users

Consideration

Tests on GET requests proved that the average response time remained below 500 ms, that can be considered as a good value in the context of web sites. With reduced loading times in fact, also the probability of the user bouncing to other sites decreases[5]. Possible improvements can be reached with a fine-grained index tuning or with replica set or data replication in MongoDB.

6.2.2 Insertion operation

For write operations, I considered performing requests to the REST endpoint dedicated to personal information; the server, in fact, exposes the URL *rest/personal-information/* which can accept both POST and GET requests. In the case of POST requests, the content of new personal information is sent to a function that performs validation and consistency controls on input data, creates a new *PersonalInformation* object and then saves the instance in the database, linking the new entry with the current user. I decided to consider this endpoint because it can be used both by users through the web application or by software systems, which can perform automatic insertions.

The test scenario that has been considered is the one in which the number of users is fixed while the number of entries for each user increases. The different setups involved:

- a database with 500 users and 100 personal information per user.
- a database with 500 users and 5,000 personal information per user.
- a database with 500 users and 10,000 personal information per user.

For each configuration, I executed 500 write operations and collected the response time of the POST requests. Results are displayed in the following figure.

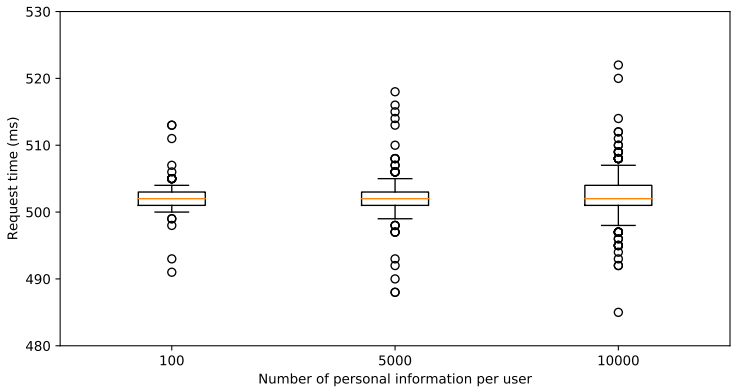


Figure 6.3: Results of insertion tests

Also for write operations, the considered numbers of personal information seem to have little impact on system performance. The box plots show for each configuration an average request time of 502 ms. The first two setups present very similar characteristic, with the 50% of data concentrated between 501 and 502 ms, minimum response time of 498 ms and maximum response time of 506 ms. The third configuration is slightly different: whiskers are longer, with a minimum value of 497 ms and a maximum value of 508 ms, while the higher height of the central box shows more distributed data. Also, the median is not in the centre of the box, which implies positively skewed data. This might indicate a higher influence of the personal information number dimension on system performance.

	lower whisker	**lower quartile**	**median**	**upper quartile**	**upper whisker**
setup 1 (100)	500 ms	501 ms	502 ms	503 ms	504 ms
setup 2 (5,000)	499 ms	501 ms	502 ms	503 ms	505 ms
setup 3 (10,000)	498 ms	501 ms	502 ms	504 ms	507 ms

Table 6.3: Results of insertion tests

Chapter 7

Conclusions

In this work I presented a prototype of a user information repository which defines one of the main modules of the PIMCity Development Kit and which can be integrated into more complex Personal Data Stores.

The system has been developed starting from backend components, to delineate the main features that need to be provided when building a PDS. The core principle that has been followed for the entire project is flexibility; the idea was to develop a system that could be as flexible as possible and could handle different and heterogeneous types of data, without having a rigid structure to follow. A fixed data pattern, in fact, could have been easier to manage developer-side, but it would have restricted final users too much. To have a dynamic and elastic environment, a schema has been introduced: a YAML file that contains the configuration of the whole system, listing the types of information that the P-DS can accept, the different fields of each data type and the possible actions that can be performed on each entry. Moreover, the schema allows to execute some consistency controls on inserted data but does not limit users, since it can be updated or extended in order to expand the set of accepted information. The prototype schema has been designed to allow users insert data belonging to *personal information* group (first-name, last-name, year of birth, etc.), to *browsing history* group (visited URLs) and to *location history* group (visited locations); this three categories have been chosen as common data groups that users may frequently store in their Personal Data Safe.

The flexibility principle has been mirrored also in the user interface, which contains as few hard-coded elements as possible. The base skeleton

of the pages is filled with the information that is retrieved from the backend and from the schema, which suggests what are the allowed actions and so the graphic elements to display.

Finally, tests performed for read and write operations proved good performance in real situations, both for high numbers of users (up to 10,000) and for high numbers of personal information for each user (up to 10,000). The MongoDB database managed to perform operations efficiently in all the carried-out tests, with reasonable response times.

7.1 Future work

A possible interesting and challenging future work could be changing completely the database type that supports the prototype and passing from a non-relational system to a relational one, such as PostgreSQL[29]. In the development of the Personal Data Safe, in fact, I have not fully exploited the potentialities of MongoDB, due to the difficulty of integrating Django and MongoDB through Djongo. For example, the relation between *User* class and *PersonalInformation* class is defined in a SQL fashion, using a foreign key, while non-relational environments prefer other techniques. The limitations that I encountered made me think that maybe the integration of non-relational databases with web frameworks, especially Django, is not at a fairly mature level. It could be interesting to explore relational solutions and understand how the system behaves with a database Django is optimized to work with.

Regarding the backend features, futures releases of the P-DS prototype may include the integration with other PIMCity modules. The PDK in fact is composed of several components that can be combined to build a complete Personal Data Store. The P-DS, for example, need to communicate mainly with the Consent Manager, which is the module responsible for defining the access privileges to user data; all the access to the data repository must be preceded by the Consent Manager intervention, that tells the P-DS who has the right to access a particular set of the user data. In future, I would like to define the communication mechanism between the P-DS and other components, towards the definition of a complete and integrable prototype.

[29] https://www.postgresql.org/

For user interface, I would like to explore different solutions, besides basic HTML and JavaScript code. Nowadays, in fact, there is a huge number of possibilities for implementing client applications, from vanilla systems where everything is defined from scratch to frameworks that allow building complex applications with less or no effort. For the current version of the prototype, I focused more on the offered features of the user interface, rather than the style of the whole application, which is composed mainly by basic HTML pages and JavaScript for the dynamic elements. A possible alternative that can be examined in future works is the integration of the prototype with the Angular framework. Angular is very popular today and enables the definition of single-page applications, which are gaining more and more ground. This could lead to a more user-friendly interface with a general aspect that users are most used to.

Finally, future development branches could involve the other category that will use the system, so data buyers. Data buyers are organizations and companies that are interested in buying user personal information to perform statistics and analysis aimed at targeted advertisements. In order to build a fair market for individuals, accesses to users' personal information must be regulated by the Consent Manager module, that defines rights and permissions toward data. Therefore, the Personal Data Save should provide interfaces and functions for data buyers, giving them the possibility to retrieve users' data in a controlled way.

Acronyms

AWS	Amazon Web Service.
GDPR	General Data Protection Regulation.
HDFS	Hadoop Distributed File System.
IAB	Interactive Advertising Bureau.
ORM	Object-Relational Mapping.
P-DS	Personal Data Safe.
PDA	Personal Data Account.
PDK	PIMCity Development Kit.
PDS	Personal Data Store.
PIMS	Personal Information Management System.
RDBMS	Relational DataBase Management System.
RTB	Real Time Bidding.
S3	Simple Storage Service.
SSL	Secure Sockets Layer.
URI	Uniform Resource Identifier.
XSS	Cross-Site Scripting.
YAML	YAML Ain't Markup Language.

Glossary

CouchDB	Open source database that focuses on ease of use and on being “a database that completely embraces the web”. It is a document-oriented NoSQL database that uses JSON to store data, JavaScript as its query language using MapReduce, and HTTP for an API.
curl	Command line tool to transfer data to or from a server, using any of the supported protocols (HTTP, FTP, IMAP, POP3, SCP, SFTP, SMTP, TFTP, TELNET, LDAP or FILE).
data subject	An identifiable natural person who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person.
Docker	Lightweight virtualization system that enables to create a portable, self-contained, lightweight container to package applications.
MVC	Stands for "Model-View-Controller". MVC is an application design model comprised of three interconnected parts. They include the model (data), the view (user interface), and the controller (processes that handle input). The MVC model or "pattern" is commonly used for developing modern user interfaces. It provides the fundamental pieces for designing programs for desktop or mobile, as well as web applications.

POJO	Acronym for Plain Old Java Object. It's a normal Java object class and does not serve any other special role nor does it implement any special interfaces of any of the Java frameworks.
REST	Representational state transfer is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the internet.

Bibliography

[1] *2018 reform of EU data protection rules*. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679. European Commission, May 25, 2018. (Visited on 09/08/2020).

[2] Tom Christie. *Django Rest Framework (Version 3.11)*. https://www.django-rest-framework.org/.

[3] *Djongo (Version 1.3.3)*. https://nesdis.github.io/djongo/.

[4] *ePrivacy Directive*. https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32002L0058&from=EN. European Commission, Nov. 25, 2009. (Visited on 09/19/2020).

[5] *Find out how you stack up to new industry benchmarks for mobile page speed*. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/. (Visited on 10/03/2020).

[6] Django Software Foundation. *Django (Version 2.2.12)*. https://djangoproject.com.

[7] Armando Fox and Eric Brewer. "Harvest, yield, and scalable tolerant systems". In: *IEEE CS* (1999). DOI: 10.1109/HOTOS.1999.798396.

[8] *How One Second Could Cost Amazon $1.6 Billion In Sales*. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales. (Visited on 10/03/2020).

[9] Yves-Alexandre de Montjoye, Erez Shmueli, Samuel S. Wang, and Alex Sandy Pentland. "OpenPDS: Protecting the Privacy of Metadata through SafeAnswers". In: *PloS one* 9.7 (2014).

[10] *Real Time Bidding (RTB) Project*. IAB Technology Laboratory. https://www.iab.com/wp-content/uploads/2016/03/OpenRTB-API-Specification-Version-2-5-FINAL.pdf, Dec. 2016.

[11] Martino Trevisan and Federico Torta. *Personal Data Safe*. https://gitlab.com/pimcity/wp2/personal-data-safe.

[12] Martino Trevisan, Stefano Traverso, Eleonora Bassi, and Marco Mellia. "4 Years of EU Cookie Law: Results and Lessons Learned". In: *Proceedings on Privacy Enhancing Technologies*. 2019.