Analysis and Prototyping of Personal Data Store Solutions and Applications

Maximilian Lucas Manfred Mayr

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 24, 2019

Maximilian Lucas Manfred Mayr

Contents

Abstract

Users have no control over data they uploaded or that was collected about them on the internet as nearly all of the data is stored in databases of different platforms. The companies behind these platforms often hide the details on how the data is used behind long terms of use agreements. This way, users often do not know that their data is used to create extensive profiles about them. These profiles allow third-party companies to target advertisements to specific user groups. The profiles can also be used to manipulate and spy on users as well as to spread fake news. The potential abuses and the problems concerning the lock-in on platforms—which forces users to stay on existing platforms—are a problem and need to be addressed.

Before presenting a solution, the thesis provides insights into *Personal Information Management*. The reasons why information management is important are discussed. Various types of personal data are explained to showcase the amount of data that is relevant to users. Afterwards, different tools, their problems, as well as shortcomings, are presented.

The next chapter explains the idea of *Personal Data Stores* (PDS), starting with their historical development and the definition of essential terminology. Reasons, why PDS could be a solution to the problems mentioned earlier, are presented. Afterwards, a short market analysis of current PDS solutions is done. Next, the thesis reviews the necessary key technologies in the areas of semantics, as well as authorization and authentication. They are followed by a compilation concerning the regulatory framework of PDS solutions.

Finally, an application that uses an existing PDS solution as the backend is developed. During the development, several problems with the existing PDS solution occurred. The encountered problems are analyzed and are solved in a prototype PDS solution that gets designed and implemented.

Kurzfassung

Benutzer haben keine direkte Kontrolle über die meisten Daten die sie im Internet hochgeladen haben oder die über sie im Internet gesammelt werden. Die Gründe dafür sind, dass fast alle Daten auf Servern verschiedenster Plattformen gespeichert werden. Die Unternehmen hinter diesen Plattformen verbergen oftmals die Details darüber, wie die Daten verwendet werden, hinter umfangreichen Servicevereinbarungen. Auf diese Weise wissen die Benutzer oft nicht, dass ihre Daten verwendet werden, um umfangreiche Profile über sie zu erstellen. Diese Profile ermöglichen es Drittunternehmen, Werbung gezielt auf bestimmte Benutzergruppen auszurichten. Die Profile können auch dazu verwendet werden, Benutzer zu manipulieren und auszuspionieren sowie gefälschte Nachrichten zu verbreiten. Der potenzielle Missbrauch und die Probleme im Zusammenhang mit dem Lock-In auf Plattformen (der Fakt, dass Benutzer auf bestehenden Plattformen gehalten werden) sind ein Problem und müssen betrachtet werden.

Bevor eine Lösung präsentiert wird, wird eine Einführung in *Personal Data Stores* (PDS) gegeben. Die Gründe, warum Informationsmanagement wichtig ist, werden diskutiert. Verschiedenen Arten von personenbezogenen Daten werden erläutert, um die für Benutzer relevanten Datenmengen zu verdeutlichen. Diverse Anwendungen und Werkzeuge sowie deren Probleme und Mängel werden vorgestellt.

Das nächste Kapitel erklärt die Idee hinter *Personal Data Stores*, beginnend mit ihrer historischen Entwicklung und der Definition der wesentlichen Terminologie. Es werden Gründe dargelegt, warum PDS eine Lösung für die zuvor genannten Probleme sein könnten. Danach erfolgt eine kurze Marktanalyse von aktuellen PDS-Lösungen. Anschließend werden die notwendigen Schlüsseltechnologien in den Bereichen Semantik sowie Autorisierung und Authentifizierung untersucht. Darauf erfolgt eine Übersicht über den regulatorischen Rahmen in dem sich PDS bewegen.

Anschließend wird eine Anwendung entwickelt, die eine bestehende PDS-Lösung als Backend nutzt. Während der Entwicklung traten mehrere Probleme mit der bestehenden PDS-Lösung auf die angesprochen werden. Die gefundenen Problematiken werden analysiert und in einem Prototyp gelöst, welcher von Grund auf neu entwickelt wurde.

Chapter 1

Introduction

This chapter provides an overview of the motivation of this thesis and outlines the problem that it tries to solve.

1.1 Problem Definition and Motivation

The amount of personal data is expected to tenfold until 2025 [42]. That alone would not be a problem, but the place where that data is going to be stored is crucial. Nearly all of the data generated by users or about users is saved in databases provided by different applications or platforms. While users have to trust companies with the safekeeping of their data, companies may betray that trust and abuse user data to generate more revenue. The abuse of personal data and the vendor lock-in on platforms—that forces users to stay in closed ecosystems—only worsen the situation. The conditions seemed to improve worldwide when the *General Data Protection Regulation Act* (GDPR) went into effect in 2018 [63]. The regulation forced companies to change their system architectures and adapt their terms of use in a way that is more beneficial to users. These guidelines are often referred to as *Privacy by Design* or *Privacy by Default*. Observations over the last year show that the regulation helped to strengthen monopolies far more than it helped users [57]. Nonetheless did the regulation help to show users that data breaches happen far more often than previously reported [82]. The GDPR also promised users access to data collected about them or created by them. Companies often refuse or stall requests that they are obligated to fulfill by law [74].

With improvements in technology for personalization and personal assistants in recent years, the internet needs to undergo a paradigm shift in the way how personal data is saved and used. One solution could be *Personal Data Stores* (PDS), an old idea that recently gained a lot more traction as storage space is becoming exceedingly more inexpensive.

1.2 Goal of the Work

This thesis aims to educate the reader about *Personal Information Management* (PIM), the different types of data, and the information users encounter or can experience. The basic knowledge about analog PIM systems, as well as electronically assisted PIM systems and their limitations, are explained. Building on these principles, the idea, motivation, and history of *Personal Data Stores* are explained. Various solutions on the market are analyzed and compared. An application gets implemented that showcases the interaction of an application with an existing PDS solution and aims to tackle the problem of missing adoption of PDS solutions. After showcasing the outcome of the prototype, the problems of existing PDS solutions are explained. A prototype for a PDS system is implemented that solves the issues that have been encountered.

Chapter 2

Information Management

Jones describes in [5, p. 5] that people spend most of their lives looking for things such as their car keys, their phone, an address, or a website. Once they found what they are looking for, they need to keep track of that information. The challenge is to organize the information in such a way that it can be easily found in the future. In the past, most data was organized physically on paper. Since personal computers are in every modern household now, people switched their preferred way of storing information. Information is saved digitally either on a local personal computer or in the *Cloud*[1].

2.1 Personal Information

Before diving into managing and organizing the information that a person found or has collected, a clarification about the type of information is needed. This thesis focuses on information about persons or relevant to persons. This makes the data collected personal information. Personal information is not only generated or collected by the user as Table 2.1 shows. Most types of data people come in contact with can be assigned

[1]A buzzword for storage that can get accessed from anywhere, hosted in the web.

Table 2.1: Overview over different personal data types [5, p. 34].

Type	**Examples**
Owned/controlled by “me”	emails, files on the computer’s hard drive
About “me”	browsing history, credit history, medical records
Directed towards “me”	personalized ads, phone calls
Sent/Posted/Provided by “me”	email, personal web site, published articles, postings, shared pictures
Experienced by “me”	visited websites, books in a library, watched TV programs, movies, TV series
Relevant/Useful to “me”	unknown but useful/relevant information available “out there”

to one of these categories. Some of the data in these categories is directly accessible by the user, while other data is stored at an unknown location.

2.2 Personal Information Management

Jones provides the following definition for *Personal Information Management* [5, p. 5]:

> *Personal Information Management* (PIM) refers to both the practice and the study of the activities a person performs in order to acquire or create, store, organize, maintain, retrieve, use and distribute the information needed to meet life's many goals (everyday and long-term, work-related and not) and to fulfill life's many roles and responsibilities (as parent, spouse, friend, employee, member of community, etc.). PIM places special emphasis on the organization and maintenance of personal information collections in which information items, such as paper documents, electronic documents, email messages, web references, handwritten notes, are stored for later use and repeated re-use.

This definition offers a good overview of PIM but fails to address points such as legal challenges in the process of information flow. As Jones [5, p. 5] further explains does PIM also involve the managing of privacy and permission related issues. Not every employee of a company, for example, is allowed to see every piece of information in the company. To summarize, the tasks of *Personal Information Management* involve [5, p. 5]:

- Creating or acquiring information,
- Organizing information,
- Storing information,
- Maintaining information,
- Finding or retrieving information and
- Distributing information.

These main tasks are well researched but keep changing over time due to technological and social progress. One example of such a change is the importance of specific tasks. Information overload, for example, is critical in a society that is permanently connected. Some workers need to be reachable even in their spare time if something goes wrong, pupils need to be shielded and instructed to put away their mobile phones to properly comprehend a lecture. Another example is advertisements that try to catch the attention of someone in the hope of motivating them to buy a specific product. The time and attention of people need to be protected as these examples show [5, p. 5]. *Personal Information Management* is one way of helping people to better use and manage their resources (money, time, and attention) as Jones [5, p. 9] explains. Because of that, the need for PIM will continue to increase in the future.

2.3 Tools for Information Management

The tools used to support or assist users with *Personal Information Management* are often called PIM Tools. The selection of the right tool for users depends on the individual technical skills and knowledge of the user. For advanced users a hand-full of PIM tools, such as *Keep It*[2], *Chandler*[3] and *EssentialPIM*[4], are available. These projects allow to synchronize data across various devices and offer the possibility to save different kinds of data at a central location. Most users do not want to use such a complex system and instead prefer to use multiple smaller applications. Each of these smaller applications usually focuses on a single task. To manage lists and tasks, for example users often use applications and services such as *Google Keep*[5], *Todoist*[6], *Wunderlist*[7] and *OneNote*[8]. Other users do not want to use any tools and instead prefer simple text files to note things down quickly. These files are often organized in a folder structure to enable simple navigation to relevant files. Placing files in folder structures might be the most common way of organizing data. Other file explorers like the *Finder* on *macOS* even allow tagging of files for quick access.

2.4 Shortcomings

Personal Information Management and the tools used have shortcomings both physically on paper and on hard-drives. If the amount of information that is stored increases, it gets harder to store data that belongs together in the same place without losing the overview. The advanced tools that allow users to store the data at one place that belongs together are too complicated for regular users. The simple tools these non-technical users prefer do however not allow them to store all data in one place. The technical limitations of the tools that are used for PIM are in most cases the problem. Traditional file structures—that most users use to order their documents, pictures, and other files—for example, are not flexible. To store a file in two different locations—what would help the user to remember the file—a link would have to be created and maintained. Another example would be if the user would want to add additional information to the file. Traditional data formats cannot be extended while maintaining their portability. This limits the information that a user is able to add to the file and depends on the type of the file. In most cases, users have to think about other solutions, maintain them over years, and export the additional information manually every time the file is copied. Image viewers that support filtering of pictures by faces have the same limitations. They allow to locate pictures of one person across multiple albums, but the information about the faces only gets stored in the application itself. Once the picture is exported, the information is lost as there is no standardized way to store such information in a picture.

[2] http://reinventedsoftware.com/keepit/
[3] http://www.chandlerproject.org/
[4] https://www.essentialpim.com/
[5] https://keep.google.com/
[6] https://todoist.com/
[7] https://www.wunderlist.com/
[8] https://products.office.com/de-at/onenote/digital-note-taking-app

Chapter 3

Personal Data Stores

Jones [6, p. 91] shows that *Personal Data Stores* are not a new idea at all. Vannevar Bush proposed such a system in 1945, called *Memex*, in his famous article “As We May Think”. The *Memex* (short for memory extender) system described in [40] was a hypothetical device which could store all books, pictures, movies, communication, experiences, and other records of a person. It would continuously capture all the data and store it automatically in a searchable format. This would allow the owner to query the system with enormous speed and flexibility to look up any record.

Several points of Bush’s vision have already been realized as [6, p. 91] points out. Systems designed to continuously capture what a person sees or does are used in specialized areas such as law enforcement. Mobile phones are also turning more and more into a *Memex* system. Most people always carry a phone with them that is equipped with dozens of sensors. The sensors can continuously capture a variety of data if enabled. Special moments can be captured using a built-in camera and accessed later with ease. Not only the image itself is captured in such cases, but also longitude, latitude, and various other information gets saved and can be accessed later. Additionally, they provide access to appointments and old memories in the form of photos, text, and video as well as the world wide web itself. Other parts of Bush’s vision such as connected thoughts and implicit querying were realized in the 1960s by Douglas Engelbart in the system showcased in the famous “The Mother of All Demos”[1].

In recent years, after the collapse of the dot net bubble, the idea gained traction again as [6, pp. 91–92] explicates. Systems such as *MIT’s Haystack* project and *Microsoft’s Stuff I’ve Seen* demonstrate personal archives. Over the years, with regards to the increasing startup spirit worldwide and the problems showcased in Section 3.2, several projects emerged that aim to provide *Personal Data Stores.*

3.1 Personal Cloud vs. Personal Data Store

Bolychevsky [38] shows that the market for *Personal Clouds* is relatively mature. *Personal Clouds* only support raw files such as documents, pictures, and video and provide a way to upload, manage, and share that data with others. Some examples of *Personal*

[1] http://www.dougengelbart.org/content/view/209/448/

Clouds are online file storage services such as *Google Drive*[2], *OneDrive*[3], *Dropbox*[4], *NextCloud*[5], and *OwnCloud*[6]. Some of these products offer downloadable extensions to save other kinds of data as well. However these systems do not solve the problems of data collection by companies or interaction with the data via third-parties [38].

Personal Data Stores are by design not limited to raw data and instead advocate to save all kinds of data produced or owned by the user in a structured way. These systems are mostly backed by a database, significantly increasing the possibilities to perform advanced queries that can generate valuable insights to their users.

3.2 Reasons for Personal Data Stores

Keeping memories connected to pictures, important documents, and general records has always been important. Usually, users keep files on their phone, their hard drive, and various web platforms. Having multiple copies is good for file redundancy but creates the new problem of keeping the data in sync. When users return from a trip with some pictures, they have to copy them from their phone to their computer, organize them into folders, possibly rename them and then upload them into their *Personal Cloud* to share them with their friends and family. Therefore multiple copies start to exist and need to be kept in sync. When this problem is approached not only from a raw file perspective but from a structured data perspective, the problem becomes even bigger. Structured data or metadata such as addresses, preferences, watched movies, current employer, and more change regularly. Keeping this kind of data in sync is important over the lifetime of a person.

Data collection is considered as the new oil rush and opens possibilities which nobody could have imagined only a few years ago [77]. Big companies like *Google*, *Facebook*, and *Amazon*, noticed the importance of big data years ago and built monopolies which have the sole purpose of collecting as much data as possible. The problem has already gotten so big that the topic was even on the *World Economic Forum*'s agenda. The description by George Soros is quite fitting [72]:

> These companies have often played an innovative and liberating role. But as *Facebook* and *Google* have grown into ever more powerful monopolies, they have become obstacles to innovation, and they have caused a variety of problems of which we are only now beginning to become aware. [...]
>
> This is particularly nefarious because social media companies influence how people think and behave without them even being aware of it. This has far-reaching adverse consequences on the functioning of democracy, particularly on the integrity of elections.

PDSs are one way to solve the problems mentioned earlier and offer additional benefits for users, giving them a great incentive to use them.

[2]https://www.google.com/drive/
[3]https://onedrive.live.com/about/auth/
[4]https://www.dropbox.com/
[5]https://nextcloud.com/
[6]https://owncloud.org/

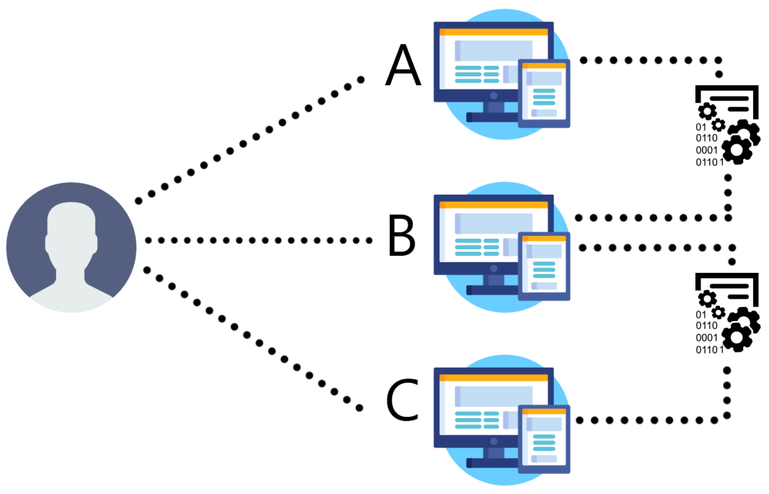


Figure 3.1: This figure showcases the acquiring data model and how the platforms A,B, and C interact with each other. If one platform needs data from another it requests that data via the *OAuth 2.0* process. All data remains on servers of the platforms. Icon source [53].

3.3 Comparing the Paradigms

The *Web 2.0* introduced users to the acquiring model of the web. Users upload, enter, and create data on websites, but their data remains on the platform. If a website needs any of the data stored by another platform, it depends on a publicly available interface. The user can authorize the website to gain access to the data via the API. This process is usually handled using the *OAuth 2.0* authorization framework—see Section 4.3.1. *OAuth 2.0* allows users to grant or reject access to data which another platform attempts to access. Users can control the flow for each data type separately restricting platforms only to the data they are willing to exchange. The acquiring model shown in Figure 3.1 is convenient because users do not need to worry about storage and hosting. Users do however also lose control over their data, as explained in Section 3.4.3.

PDSs want to change this by centering the users and their PDSs in the process, effectively turning it into an inquiring model—see Figure 3.2. Instead of authorizing platforms to communicate with each other, they would only interact with the PDS. Thereby the users themselves can define what data gets exchanged. This gives the user more control over the permissions than most platforms currently offer, as well as control over the data itself—more about the other advantages in the following sections.

3.4 Single Source of Truth

In software development, engineers faced the same problem as described in Section 3.2. Data that belongs together gets stored in different places and needs to be synchronized to keep the user interface consistent. This problem led to the invention of the *Single Point of Truth* (SPOT) or *Single Source of Truth* (SSOT) design pattern that is nowadays

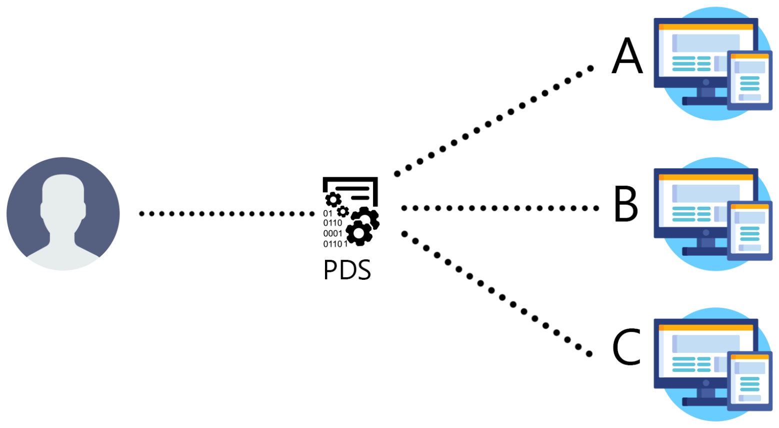


Figure 3.2: The inquiring data model centers the user and his PDS in the data exchange process. All data is stored in the PDS system and the user can decide what platform has access to what information. Icon source [53].

widely used in enterprise frontend and backend development. Examples for SSOT's are *Redux*[7] and relational databases such as *MySQL*[8], as well as *NoSQL* databases such as *mongoDB*[9]. This pattern advocates the practice of storing data and information only once. If others need the data, they can access it directly or link to it with a reference.

In Section 3.2 the existence of multiple copies and versions of the same data were explained. A *Single Source of Truth* for data enables users to manage their data synchronization easily. If a user for example needs to change his or her address after moving, just a single record needs to be changed in the PDS. Websites, applications, and other shareholders than retrieve the updated information directly from the PDS when they need it. This way, the data will always be up to date when it is needed.

3.4.1 Single Identity

Applying the SSOT pattern to personal data and the identity of the user creates a unique opportunity. First of all, the PDS can be designed in a way that allows users to use it to log in on other websites. This type of authentication, called *Single-Sign-On* is widely known and used, as shown in Figure 3.3. In the future one of these buttons could be labeled "Log in with PDS". Choosing that option would prompt the user to provide the URL of his PDS to log in. Due to the underlying technology, the website could then request the required data from the PDS system, as explained in Section 3.4.2. When PDSs are used for the log in on third-party websites users only need a single password. Current data shows that the number of accounts leaked is higher than the world's population[10], whereby only about 55%[11] of the population currently have access to the internet. In conclusion, statistics show that every user is already affected multiple

[7] https://redux.js.org/
[8] https://www.mysql.com/
[9] https://www.mongodb.com/
[10] https://haveibeenpwned.com/
[11] https://en.wikipedia.org/wiki/Global_Internet_usage

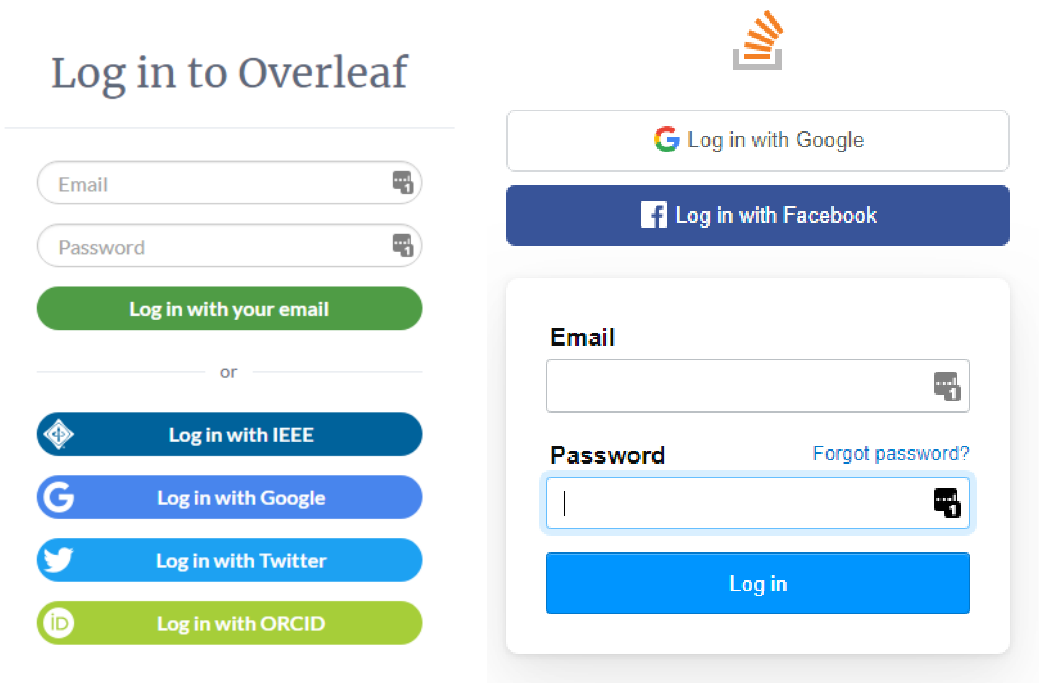


Figure 3.3: This image showcases the various login options presented to users on the platforms *Overleaf* and *StackOverflow*.

times. Leaks are often caused by bad passwords that users usually choose because they have to remember so many of them. If users have to remember a single password or no password at all, for example by using a hardware token, that number is likely to decline.

3.4.2 Control Access to Data

The users are the only ones who have full control over the data stored in their *Personal Data Store*. If users do not trust a website anymore, they can revoke the website's access to the data stored in their PDS. This mechanism of control is already familiar to users from mobile devices—as showcased in Figure 3.4—where applications request access to sensors or data they need. This form of requesting access forces users to think about why a calendar app would need access to private pictures. Figure 3.5 shows an example of how data gets requested from a third-party to data stored by *Facebook*.

3.4.3 Ownership of Data

As described in Section 3.4 is the SSOT pattern the basic idea behind PDS systems. With PDS users have the choice whether they want to host the PDS system themselves or if they want to use a more convenient service that manages the hosting for them. Both ways will be present in the future web due to the diversity of its users and their technical expertise. Even if users decide to use a service that handles hosting for them, are they still the only ones with access to the data. Because less data is stored in one

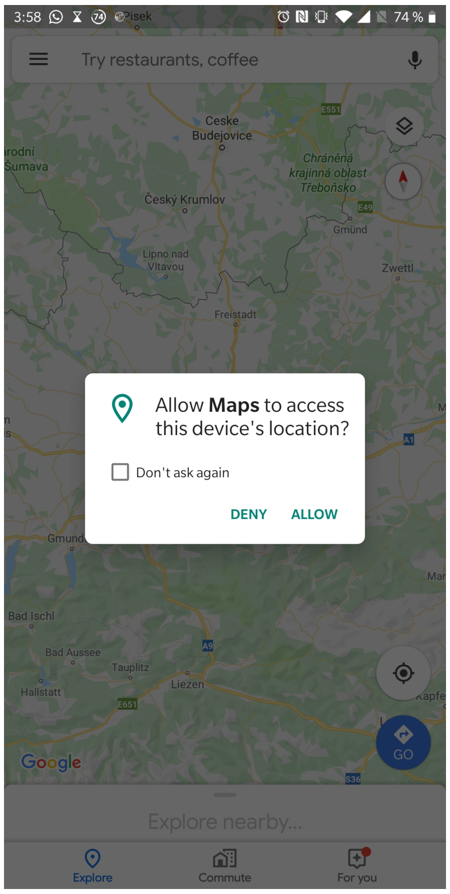


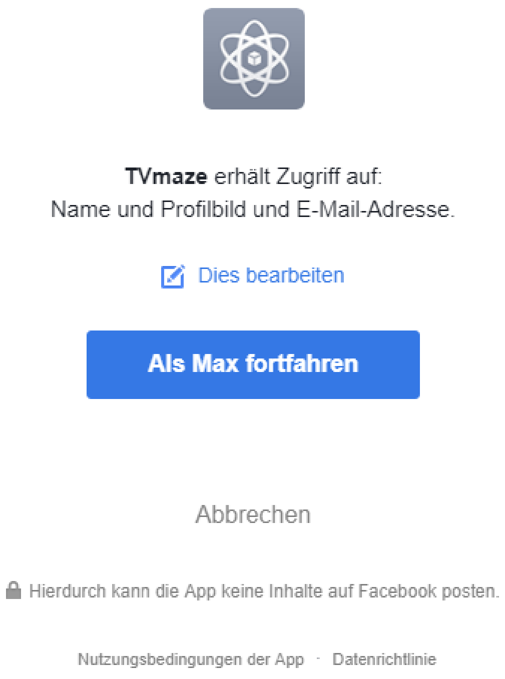
Figure 3.4: Application requesting access to sensor information on an *Android* device.

place large scale data leaks[12] could be a thing of the past. This fact could be one of the main drivers of adoption for PDS systems.

Business Models on the Web

Currently, big platforms gain revenue by selling data about their users, either by showing them personalized advertisements or by selling their data to third-parties. By taking the data from the platforms and hosting it somewhere else, their business model has to change. Users may still prefer to give away data in exchange for free usage of services. Users are given a choice, whether they want to pay for a service or if they want to share their data. Future data markets will allow users to sell their anonymized data to the highest bidder. Companies will use the bought data to improve their machine learning algorithms in order to provide the best service on the global market.

[12] https://informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/



TVmaze erhält Zugriff auf:
Name und Profilbild und E-Mail-Adresse.

Hierdurch kann die App keine Inhalte auf Facebook posten.

Figure 3.5: Data request from the third-party application *TVmaze* to user data on *Facebook*.

3.4.4 Freedom of Choice

Users have the choice of using any application or service they want because the data is decoupled and stored in their PDS. This fact allows users to change or try out a new user interface within a few mouse clicks. This possibility will likely spark innovation in the web and will benefit users who can choose between dozens of available applications and can select the one which best fits their preferences. The stress-free process of switching applications allows small companies to compete with current big players on the web because it levels the playing field. Developers can use the provided infrastructure of the PDSs and build their applications on top of them, allowing them to develop new applications and services rapidly.

3.5 State of Personal Data Stores

The main problems of *Personal Data Stores* that need to be solved seem to be the accessibility for non-technical users as well as the missing adoption. Several projects exist which offer an excellent entry experience by providing a free PDS that comes preconfigured. By providing free entry PDSs with limited storage, users are more likely to try them out. Another solution how the adoption could be increased will be outlined in Section 5.1. The problem of accessibility is mitigated by providing preconfigured

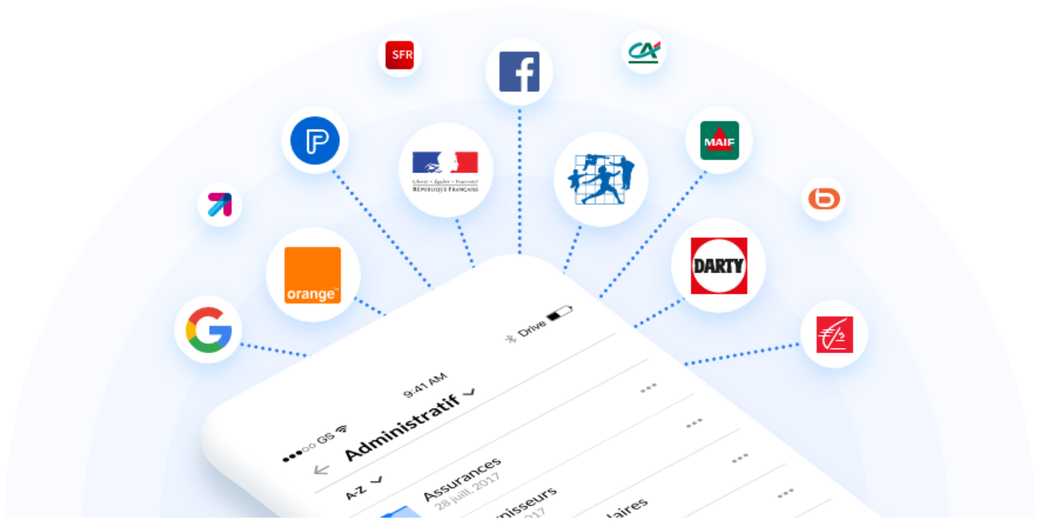


Figure 3.6: The idea behind *Cozy Cloud* is to collect data from multiple sources [44].

systems that do not need maintenance. Nonetheless, PDSs need a simple user interface for users to interact with the PDS.

3.6 Aggregation Platforms

Aggregation platforms are *Personal Clouds* with the advanced functionality that allows them to pull data from various providers and collect them in a database. This approach is helpful to have an overview of which data gets stored on what platform. These platforms usually provide a way to access the collected data from third-party applications to analyze and gain new insights. Aggregation platforms increase the connectivity between platforms. One of the most significant disadvantages of the aggregation approach is that the platforms usually have low extensibility due to their initial design goals as the following examples show.

3.6.1 Cozy Cloud

Cozy Cloud [44] is a French project that aims to extend typical cloud hosting by creating a digital home for user data, as shown in Figure 3.6. The project is fully open-source and can be self-hosted if wanted. The software design is *General Data Protection Regulation* (see Section 4.4.1) compliant and designed with *Privacy by Design* (see Section 4.4.2) principles. What makes *Cozy Cloud* interesting is the fact that they have a full-fledged *NoSQL* document database running in the background. Therefore the platform is, in theory, highly extensible and allows excellent scalability.

To access the data, developers create apps that communicate directly with *Cozy Cloud*. These apps are hosted in a user's personal *Cozy Cloud* server. This way of serving applications allows strict control over data flow using the *Content-Security-Policy* and *Cross-Origin-Resource-Sharing* (CORS) HTTP headers which prevent leaking of data to other websites. Hosting apps in the *Cozy Cloud* server and preventing websites from talking to other services is also a problem. It dramatically limits the types of services and usability of applications that would create a competitive market.

3.6.2 digi.me

The slogan of *digi.me* is “Take control of the data powering your digital life” [46]. This quote leads to the impression that users can control their existing data on various platforms. In reality *digi.me* only provides control mechanisms on the data for new applications that build on top of the *digi.me* architecture. These new applications can be controlled by the user with the fine-grained permission controls. *digi.me* does not need a server as all data is stored in an encrypted data file that can get saved to a cloud provider of the user’s choice. The available options are *Google Drive*, *Dropbox* and *OneDrive*. The major disadvantage of the *digi.me* system is the fact that it is closed source. Furthermore, data can only be accessed on phones through apps by using the provided SDK from *digi.me*.

3.7 Identity Platforms

Identity platforms go one step further than aggregate platforms by establishing a single identity and an SSOT. As shown in Section 3.4 is an SSOT one of the main benefits behind PDSs. These solutions allow applications to read and write various data to the PDS and enable the user to manage and control the data flow as the following examples showcase.

3.7.1 LifeScope

LifeScope [59] allows the user to search for data that was collected through a supported provider. Search queries of a user can be saved, data can be tagged arbitrarily, and collections can be shared as shown in Figure 3.7. *LifeScope* also provides different views of the same data. A unique feature of *LifeScope* is the option to detect special events and connect them across various providers creating a map view and timeline of what happened during events such as a vacation.

LifeScope offers a *Single-Sign-On* using *OAuth 2.0* and allows to query and to add data using a *GraphQL*[13] interface. The platform currently only supports a fixed set of objects and does not provide a way to extend the set of objects.

3.7.2 Hub-of-All-Things

Hub-of-All-Things (HAT) [52] is one of the most advanced projects trying to create a PDS. The HAT ecosystem started as a research project by multiple British universities, including the *University of Cambridge* [52]. The ecosystem is built around the HAT server, which wraps a dedicated database with numerous inner micro-services. HAT offers so-called data plugs which allow the user to import data from various sources into the HAT database. Similarly to Section 3.7.1 can third-party applications request

[13] https://graphql.org/

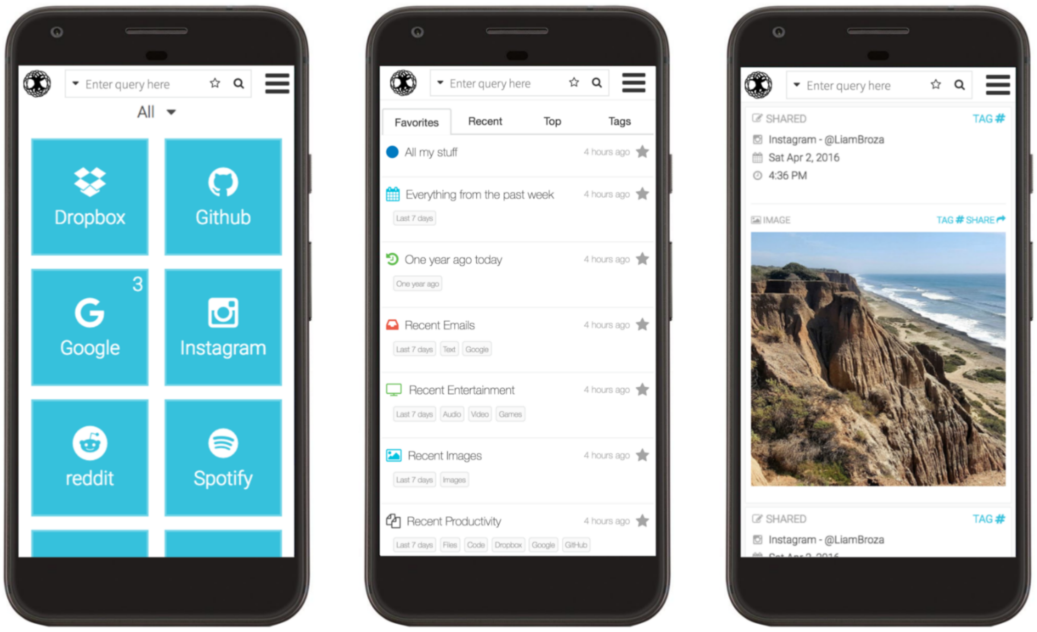


Figure 3.7: Overview over the *LifeScope* user interface [59].

access to the data stored by HAT. Users can grant and revoke permissions to apps at any time. HAT goes a step further than other solutions by also providing a solution to the AI problem. That problem describes that existing players always have the advantage when it comes to AI because they have more data. This results in AI's of big platforms that get increasingly more powerful because of the data they possess. HAT, therefore, provides AI services [52] that analyze the data on behalf of the user and save the data back to the HAT to be consumed by the user. These AI services are provided by the company behind HAT as well as by other developers who have to go through a verification process.

3.7.3 Solid

Solid (Social Linked Data) [55] is another identity platform, created by Tim Berners-Lee[14], which aims to decentralize the web by creating a standard on how user data is saved and handled as shown in Figure 3.8. *Solid* is a fully open-source project that builds on several *Linked Data* standards that can be used to build the next generation of the web. Berners-Lee's project tries to solve the problems of account proliferation, data lock-in on platforms, and missing interoperability between apps [78].

Solid does not use a database and instead stores everything as a file in a structured format to keep it machine-readable [78]. This makes running *Solid* servers cheap as almost no server resources are needed. Due to the RDF file format, all kinds of data can get stored without having to think about portability—as outlined in Section 4.2.1. Furthermore, permissions are given on a file base using the *WebACL* standard [56].

[14] https://www.w3.org/People/Berners-Lee

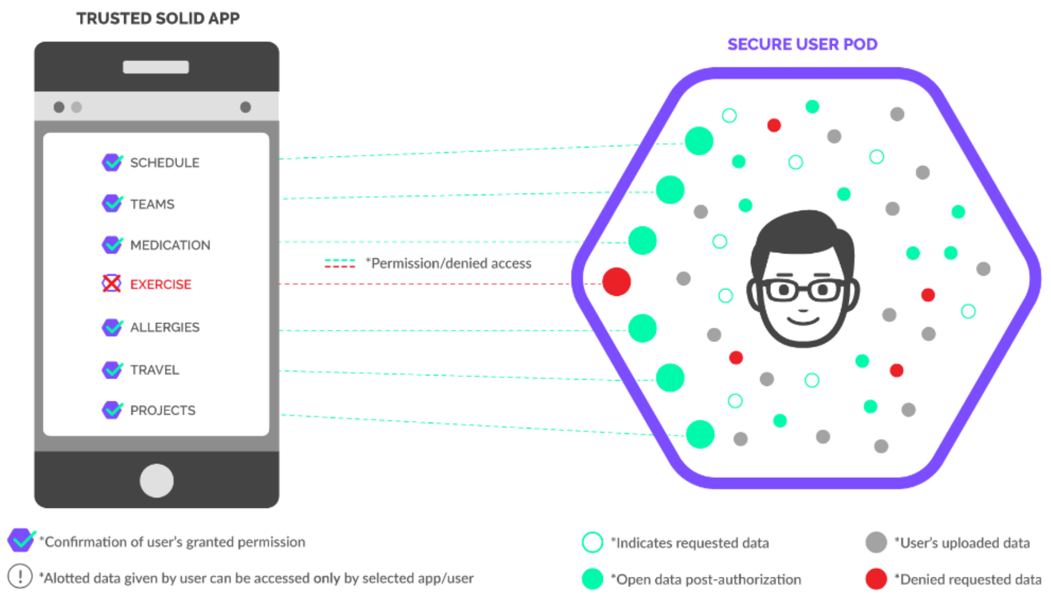

TRUSTED SOLID APP
SCHEDULE
TEAMS
MEDICATION
EXERCISE
ALLERGIES
TRAVEL
PROJECTS
*Permission/denied access
SECURE USER POD
*Confirmation of user's granted permission
*Alotted data given by user can be accessed only by selected app/user
*Indicates requested data
*Open data post-authorization
*User's uploaded data
*Denied requested data

Figure 3.8: Secure user pod concept from *Solid*, that showcases how users can manage permissions [54].

Chapter 4

Technologies

This chapter explains the basic technologies used in the prototypes that are showcased later. It should help the reader to understand the inner workings of the prototypes.

4.1 Semantic Web

The term *Semantic Web* coined by Tim Berners-Lee [1, p. 177] describes a world wide web that can be processed by machines. It is an extension of the existing web with common data formats and standards defined by the *World Wide Web Consortium* (W3C).

Berners-Lee describes in [1, p. 178] that finding information on the web was always a problem. Search engines tackled this problem by creating broad indexes of documents and websites via ranking them by word count and inter-link statistics. When the idea of the semantic web was born, it was not possible to evaluate the real content quality or the meaning of the text automatically. The semantic web was created to provide a solution to this problem and introduce real semantic search capabilities for the web. In other words, the goal is to create a web of data where machines can gather data directly from a website without having to try to interpret the text. Nowadays, newer servers often provide *Application Programming Interfaces* (API) in addition to static web pages. As a consequence, the definition of the semantic web might as well be adapted to incorporate this fact.

There has been some progress in the development of the semantic web, mostly driven by scientists and researchers, as well as companies that want to improve their search algorithms. A significant achievement has been the establishment of a standardized ontology—basically a vocabulary—which is called *schema.org*[1]. This vocabulary is used by over ten million sites to annotate data in various encodings on their pages [75].

4.2 Semantic Technology

This field of computer science prepares and saves data in a way that allows humans and machines to understand the meaning behind information. This area grew rapidly in the last years due to the developments in *Machine Learning* and *Image Recognition* as well

[1]https://schema.org/

Program 4.1: Example that shows RDF in XML syntax.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:foaf="http://xmlns.com/0.1/foaf/" >
    <rdf:Description rdf:about="">
        <dc:creator rdf:parseType="Resource">
            <foaf:name>Maximilian Mayr</foaf:name>
        </dc:creator>
        <dc:title>Personal Data Stores</dc:title>
    </rdf:Description>
</rdf:RDF>
```

as the amount of structured and unstructured data which is available.

Semantic technology was built around the idea of entities and the relationships they have with other entities [70, p. 4]. Each entity such as a computer belongs to somebody; that person lives in a country and that country is on a specific continent. These examples show, it is possible to create relationships for anything and anyone. The harder part is to save data in a way that is flexible enough to represent all kind of entities and relationships which are often unknown beforehand. There are different ways to store the data, as shown in the following sections.

4.2.1 Resource Description Framework (RDF)

The *Resource Description Framework* (RDF) is part of the backbone of the *Semantic Web* [70, p. 8]. It describes the relations and properties of entities as triplets. These triplets use *Uniform Resource Identifiers* (URI) to link to ontologies. An ontology is a special form of vocabulary which describes entities and relations. The most common representation or syntax of RDF is the XML syntax developed by the W3C[2].

Program 4.1 shows a an RDF document that describes a document called "Personal Data Stores" written by "Maximilian Mayr". Encoding data in RDF is difficult as Palmer explains in [35]. To provide an easier way to encode data in RDF, Tim Berners-Lee created a new syntax commonly known as *Notation3*:

```
<> <http://purl.org/dc/elements/1.1/creator> _:x0 .
this <http://purl.org/dc/elements/1.1/title> "Personal Data Stores" .
_:x0 <http://xmlns.com/foaf/spec/#term_name> "Maximilian Mayr" .
```

This example shows that *Notation3* is much simpler to write. For a human reader, the triplets are also more readable, which makes it easier to interpret what is going on. `<>` describes a new object [81]. The next part of the triplet describes the property or relation using the *dublincore* ontology [37]. In this case, the relation "creator" gets described in the ontology as "An entity primarily responsible for making the resource" [37]. The last part of the triplet references the third line where the name of the creator is given using an ontology called *foaf* [39]. The second line uses the "this" keyword to tell the parser that that line belongs to the entity defined in the first line. The middle part defines a property or reference using *dublincore* [37]. Moreover, in the last part, a literal is defined

[2] https://www.w3.org/

by enclosing it with double quote marks. Literals always have to be marked that way clarify what characters are part of the literal [81].

As Palmer [35] further explains, there is also a way to omit a URI, to express that an object exists without stating an explicit URI.

```
_:p1 <http://xmlns.com/foaf/spec/#term_name> "Hans Zimmer" .
```

The previous line shows an example which demonstrates that there exists a person called Hans Zimmer. The symbol "_" is a placeholder—called anonymous node—for the URI and by using ":p1" the author can label to the node to reference it in other triplets. As [35] demonstrates is it also possible to define aliases in order not to have to repeat the full URL—the first part of the URI—every time. The alias mechanism of *Notation3* can be seen in the following example:

```
@prefix foaf: <http://xmlns.com/foaf/spec/#term_> .
foaf:Person foaf:name "Hans Zimmer" .
```

Using aliases when writing *Notation3* is even simpler and faster than without and therefore commonly used.

Although RDF is powerful, it still did not manage to gain widespread adoption [43]. The main problem is the lack of convenient libraries to use for developers that would simplify the use of RDF.

4.2.2 Graph Databases

Graph databases utilize a graph structure instead of the traditional table structure to save and query data. As explained in [4, p. 1], a graph is a collection of vertices and edges. Vertices are nodes which represent an object whereas edges are the connections or relationships between them. The flexible data representation allows storing various types of data, because unlike relational databases, graph databases do not need to enforce any schema.

When comparing the paradigms of graph databases and RDF, a pattern emerges, as explained in [32]. Graph databases also contain triplets as linking two nodes together creates a triplet. Anadiotis [32] explains that there are specialized graph databases which can use the containing triplets to create proofs and deduce conclusions on existing data sets stored in the graph. These specialized versions of graph databases are called semantic graph databases or tripletstores.

Graph databases belong to the group of *NoSQL* databases [4, p. 193]. These kinds of databases became popular in recent years as traditional relational database management systems could not keep up with the ever-increasing data that is produced nowadays. The main problem was the increased execution times of queries when data from multiple tables is needed. Several different types of graph databases will be introduced in the following sections.

Property Graphs

As explained in [4, pp. 206–207], property graph databases are the most common type used. Nodes in this model contain properties that are key-value pairs, as demonstrated in Figure 4.1. Each node can be assigned one or multiple labels that determine the type of the node. This allows filtering the nodes preliminary by only selecting nodes of the

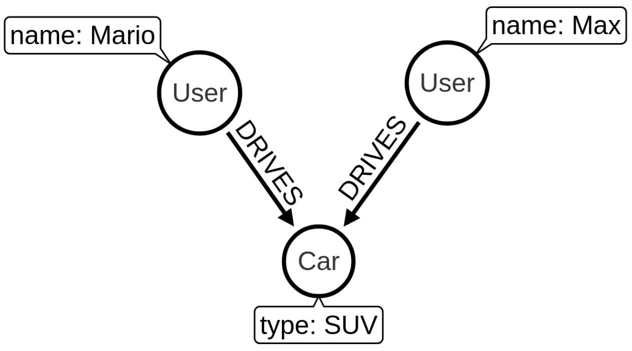


Figure 4.1: Example of a property graph.

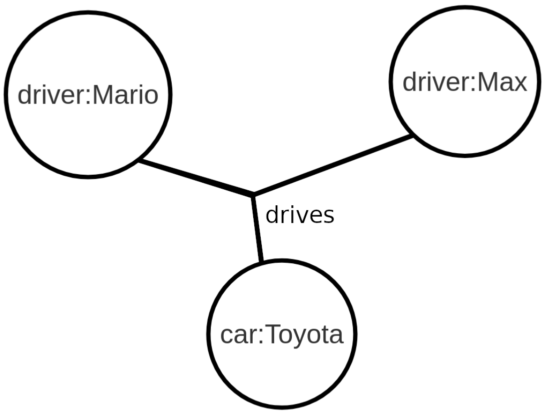


Figure 4.2: Example of a hypergraph.

type that gets requested. Due to this, property graphs are often used to store a variety of different types in a single database. Each node can be linked to any other node making it easy to store semantic relations based on node and relation types. Relations also have a label which determines its type as well as properties, what enables metadata and other information to be stored directly with the relation. Relations connect two nodes and may have a direction to strengthen semantics and readability further.

Hypergraphs

Hypergraphs connect any number of nodes using one relationship [4, pp. 207–208]. This makes them ideal for use-cases that are mainly dominated by many-to-many relationships. Nodes can optionally contain properties depending on the database implementation. Hypergraphs are multidimensional and allow a more generalized representation than property graphs. Figure 4.2 shows how the example from Figure 4.1 can be modeled as a hypergraph with a single relationship.

Triplet Stores

As described in [4, pp. 208–210], triplet stores are an outcome of the *Semantic Web* movement. They allow triplets—composed of a subject, predicate, and object—to be stored in their natural form. Triplets represent facts and knowledge. Therefore, triplet stores are often used to represent knowledge databases. Because of the stored facts, triplet stores can be used to infer connections between nodes. For these databases mostly *SPARQL* queries—see Section 4.2.3—are used to query nodes. Triplet stores are not native graph databases because they do not support index-free adjacency which permits them to directly walk connected nodes in a query. Instead, before accessing another node, they have to lookup the memory address in a global index.

4.2.3 Semantic Search Query Language (SPARQL)

As Palmer [35] explains, *SPARQL* is a query language and protocol for *Linked Data* such as RDF and *Tripletstores*. It enables users to query information from any source which can be mapped to the RDF format. Because *SPARQL* is a semantic search language, the user does not need to know how the data is stored or structured. Rather, the users ask what they want to know. Because every triplet consists of a subject, a predicate, and an object, it is possible to query for all entities of a specific type or for entities in a specific relation. This is achieved by using variables or wildcards where the query matches the given properties and returns all options for the wildcard that satisfy the query. Feigenbaum [48, p. 6] shows that *SPARQL* supports different types of queries:

- `SELECT` all results matching the wild card given in tabular form.
- `ASK` if there are any matches.
- `CONSTRUCT` new RDF triplets or graphs.
- `DESCRIBE` the matched resources by returning them as RDF.

With *SPARQL 1.1* [49], methods to update data have been introduced that enable to `INSERT`, `DELETE`, `LOAD`, `CLEAR`, `CREATE`, `DROP`, `COPY`, `MOVE` or `ADD` nodes or graphs to the store. These operations make it easier to maintain and manipulate a tripletstore.

4.3 Authorization and Identity Management

This section explains the state of the art technologies and protocols used for authorization and identity management.

4.3.1 OAuth 2.0

In a world full of platforms hosting a variety of content from sites such as *Google* and *Facebook* [61], access control is essential to enable secure exchange between platforms and third-parties. However, defining and controlling what resources can be accessed or modified is tricky. *OAuth* is an open standard which tackles that problem.

As Boyd explains in [2, pp. 1–2], before *OAuth* it was a significant security issue that users had to give their account details of their resource provider to a third-party so they could access the data hosted at the resource provider. Big companies tried to solve this issue by implementing proprietary authorization protocols such as *Yahoo's BBAuth* or *Google's AuthSub*. As Boyd further analyzed, this way of authorization solved some security issues, but it created a burden for developers who had to learn and implement multiple authorization protocols for different platforms. Fearing the implementation of proprietary protocols or the development of new proprietary standards, developers, and major companies decided to create a standardized protocol for web-based authorization workflows.

Boyd [2, p. 2] continues to explain the advantages of *OAuth* for developers. They can easily access data from *OAuth* enabled APIs, harnessing the data, and focusing on application development. Because applications and servers never get access to users passwords, developers do not need to build a secure infrastructure to handle password data. Boyd concludes [2, p. 3]:

> Having a common protocol for handling API authorization greatly improves the developer experience because it lessens the learning curve required to integrate with a new API. At the same time, an authorization standard creates more confidence in the security of APIs because a large community has vetted the standard.

As explained in [67], *OAuth* and *OAuth 2.0* share the same name and idea behind them, but they are two different protocols. *OAuth 2.0* is not backwards compatible with *OAuth 1.x* and was designed to simplify and improve the workflow of *OAuth 1.x*. The formal definition for *OAuth 2.0* according to the specification is [51, p. 1]:

> The *OAuth 2.0* authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

By now, it should be clear that *OAuth 2.0* is an authorization framework which is designed to limit access of applications or clients to protected resources.

Involved Parties

Richer and Sanso further analyzed the involved parties in *OAuth 2.0* [8, pp. 5–6]:

The *resource owner* is typically the end-user of the application. The user "owns" the resources/data on a server and can allow, deny, or delegate access to other applications or users. A typical example would be the owner of a *Google* account, and their resources would be the calendar, contacts, emails, and location data hosted on the *Google* platform.

Protected resources are objects that can only be accessed by the resource owner. Resources are served, modified, read, and deleted using an API interface. An example of a protected resource would be a printer, its functionalities are offered over an interface and allow different operations like print, send as mail, scan and read from memory stick.

The *client* is an application used to access/consume/modify resources of the resource owner. An example would be a photo application such as *Instagram*[3] which tries to access pictures hosted on *Google Drive*[4].

[3] https://www.instagram.com/
[4] https://www.google.com/drive/

Workflow

OAuth 2.0 introduces multiple workflows as shown in [65]. Each workflow was designed for a specific use-case, with *Authorization Code* and *Implicit Workflow* being the recommended and most commonly used ones.

Figure 4.3 shows the *Authorization Code* workflow on a basic level. The user visits the web application and wants to access a protected resource from the application. As the first step, the user clicks on the login button. Now the client redirects the user to the authorization server. There the user enters his or her credentials, and the authorization server asks the user for permission to grant access to the client application and the requested scopes. See Section 4.3.1 for more information about scopes. As explained in [51], if the user grants access, the authorization server redirects the user to the client application and appends the authorization code to the redirect URL. Now the client application parses the authorization code from the URL and exchanges it with the authorization server for an access token. With the received access token, the client application can now access the protected resource from the resource server.

Scopes

As Richer and Sanso [8, pp. 88–89] explain, are scopes an important mechanism in *OAuth*. As mentioned in Section 4.3.1, it was a major problem before *OAuth* to limit which resources clients can access. *OAuth* introduced scopes to represent a subset of access rights on the resource server. Scopes can be combined into a set by using a space-separated list [8, pp. 32–33]. Scopes are defined by the resource server and the access to specific scopes can be granted or denied using the authorization server. This mechanism of *OAuth* allows fine-grained permissions that the resource owner can manage. This type of permission management is familiar from the smartphones' system where apps ask for permission to perform specific actions.

Access Token

The access token, as described in [8, p. 32], is a string or object issued by the authorization server which represents the permissions of the client for a specific user. The format of the token is not defined, but mostly formats such as *JSON Web Token* (JWT)—see Section 4.3.1—are used.

To access a protected resource on the resource server, the client has to send a request with the access token (which is achieved by using the *Authorization-Header*). The resource server then checks the token and grants access if it is valid.

JSON Web Token (JWT)

Before structured tokens were invented—as shown in [8, p. 183]—the resource server had to do a database lookup at the authorization server for each incoming request to validate if the user is logged in. A commonly used implementation of a structured token is the *JSON Web Token* (JWT). A JWT contains all the information required by the resource server to determine if the client is allowed to access the protected resource. The following example of a basic JWT token shows that a JWT consists out of three parts separated by a dot.

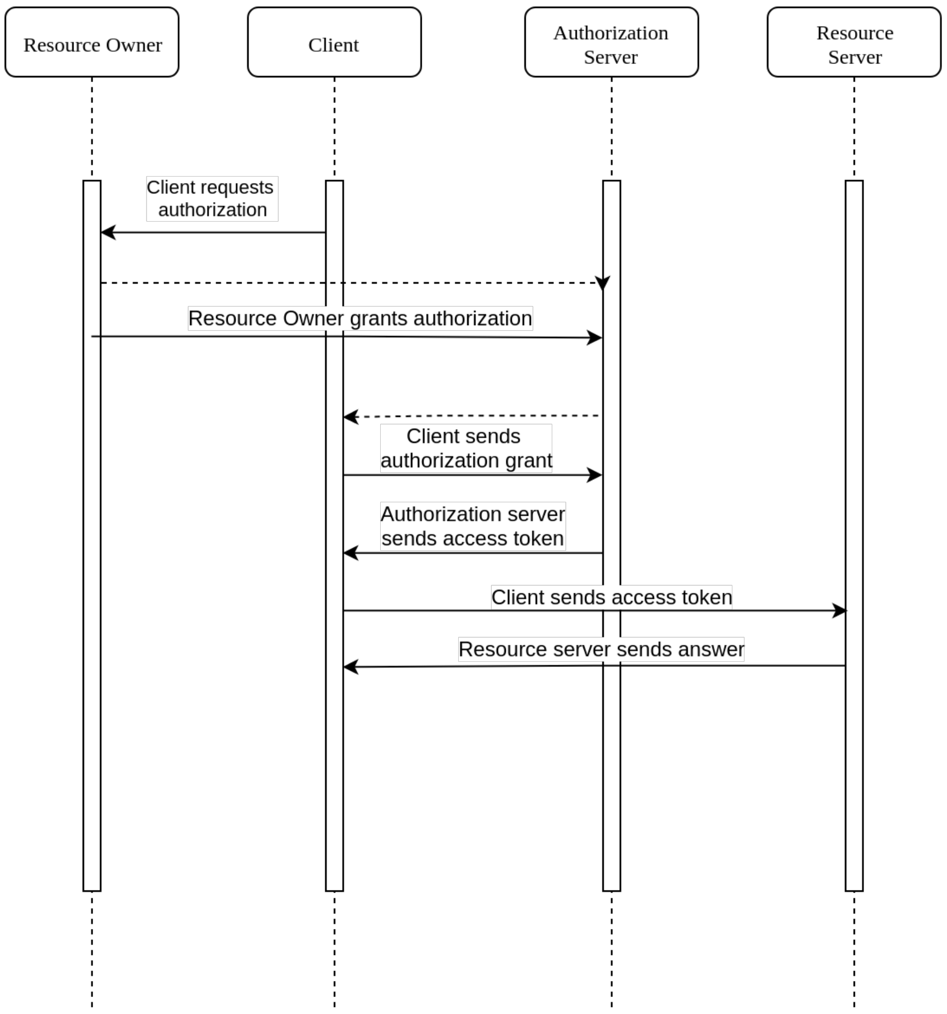

Resource Owner
Client
Authorization Server
Resource Server
Client requests authorization
Resource Owner grants authorization
Client sends authorization grant
Authorization server sends access token
Client sends access token
Resource server sends answer

Figure 4.3: High level overview of the *OAuth 2.0 Authorization Code* workflow.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJ1c2VybmFtZSI6Im1heXJfbWF4Iiwic2NvcGVzIjoibWV0YWRhdGE6cmVhZCJ9.sRnrU-
    yS0gtq15plGjawhepg72R0Gx6-gXZoa7hz_IY
```

The first part is an encoded JSON object with the type of the token and the algorithm used to sign the token. The second part contains the payload of the token, in this example a username and the approved scopes. The last part, as explained in [8, pp. 188–189], represents a signature verifying the author of the JWT and a message authentication code to guarantee the integrity and authenticity of the token.

4.3.2 OpenID Connect

Richer and Sansol [8, pp. 236–237] explain that *OAuth 2.0* does not provide a built-in authentication mechanism. Authentication indicates to the application who the current user is and if he or she is currently using the application. *OAuth 2.0* clients only receive an opaque token or string that does not tell the client anything about the user. Because a user delegates access to an application on his behalf *OAuth 2.0* is it often mistaken for an authentication protocol [66].

The *OpenID Connect* specification was introduced to overcome this issue as described in [8, p. 246]. *OpenID Connect* is the third generation of *OpenID* specifications. Unlike its predecessors, it builds directly on top of *OAuth 2.0*, making it easier for developers to integrate into their application. *OpenID Connect* provides an *ID token* through a signed *JSON Web Token*—see Section 4.3.1. As [8, p. 246] shows, is the *ID token* intended to be parsed by the client.

4.4 Regulations

This section explains current regulations that affect PDSs.

4.4.1 General Data Protection Regulation

The *General Data Protection Regulation* (GDPR) [71] was introduced in 2016 by the *European Union*. It has been active since May 2018 within the EU and the *European Economic Area*, but it also affects companies in other countries if they have customers or users who are EU citizens. The GDPR covers multiple points that are explained shortly in the following sections. The full information can be found at source [58].

Penalties

Companies or organizations that violate the GDPR can be fined up to 4% of their annual global turnover or up to 20 million Euro—depending what is bigger. These maximum fines can be imposed for severe infringements of the policy as processing or handing over data without user consent.

Consent

Companies are forced to ask for consent in an easily accessible form instead of hiding the consent or hint somewhere in their terms of usage. It must also be understandable for users what they are agreeing to before they give consent. The consent can also be withdrawn at any time.

Breach Notification

Companies need to report data breaches in all member states by reporting them to the authorities within 72 hours of becoming aware of the breach. Companies are also required to notify customers that a data breach has happened and which data got leaked.

Right to Access

This part guarantees users access to their data, free of charge in an electronically parsable format as well as the right to know if, where, and for what purposes their data is processed.

Right to be Forgotten

Users have the right to request the deletion of their data as well as the right to request that third-parties have to stop the processing of the data once invoked. This part only gives the user a way to request, the data holder then has to decide if the user's rights count or if the public interest is more important.

Data Portability

Data portability allows users to take their data from one platform (in a standard and machine-readable format) and transmit it to another platform.

Privacy by Design

Privacy by Design has already been around quite some time as a best practice. The GDPR now also makes it a legal requirement to follow the rules showcased in Section 4.4.2.

Data Protection Officers

Companies are required to have a designated person responsible for data protection. The dedicated person must report directly to the highest level of management and can be a member of the company or an external contractor.

4.4.2 Privacy by Design and Privacy by Default

Current storage paradigms are based on the *Privacy by Design* paradigm, which was developed by Ann Cavoukian as an approach to create rules and guidelines for systems engineering, as described in [3]. The principles got formalized in 1995 in a joint report created by the Canadian, Dutch and Netherlands governments. Since then the framework was adopted by the *International Assembly of Privacy Commissioners and Data Protection* in 2010 and the *European Union* in 2018, making it the de-facto standard for most internet platforms. *Privacy by Design* consists of seven core principles [3]:

- Proactive not reactive; preventative not remedial,
- Privacy as the default setting,
- Privacy embedded into the design,
- Full functionality—positive-sum, not zero-sum,
- End-to-end security—full life cycle protection,
- Visibility and transparency—keep it open—and,
- Respect for user privacy—keep it user-centric.

These principles describe that a system should work proactively on protecting the user's data. *Privacy by Default* asserts that users do not have to do anything additionally to protect their privacy within a system. All privacy related options are tuned to their favor by default. The other principles describe that privacy measures should exist upfront and should not be added later as an add-on. Companies should seek a win-win situation that is beneficiary for both them and their customers.

Personal Data Stores shift the responsibility from platforms to users, to take care of safely storing personal data [7]. They are future proof as even if platforms get shut down, users still have access to their data. *Privacy by Design* will still apply to future systems in order to take care of secure processing of data, but users will have control over what data the platform is allowed to process.

4.4.3 Directive on Copyright in the Digital Single Market (DCDSM)

The *Directive on Copyright in the Digital Single Market* [47] is a directive of the *European Union* that has been adopted but is not yet in effect. It created quite a lot of controversy as it is mainly beneficial for newspapers, publishers, and media groups. Opponents of the bill, such as major tech companies, internet users, and human rights advocates are worried that parts of the directive will bring the risk of censorship and will hinder free speech. The following paragraphs summarize the most essential points of the DCDSM, related to this thesis. The full information can be found in the regulation [47].

Article 3

This article creates a copyright exception for artificial intelligence and machine learning by excluding text and data mining for scientific research from the regulation.

Article 14

This article states that works which are in the public domain (unless the work is a reproduction and original creative work) cannot be subject to copyright or related rights.

Article 15

Under this part of the regulation, publishers are granted the right to the benefit of online usage of their press publications by news aggregators and media monitoring services. This mainly affects the teaser texts that should motivate users to click the link.

Article 17

Online content sharing service providers such as *YouTube*, *Facebook*, and others are now responsible for copyright infringements committed by users on their platforms. This article only targets commercial platforms and does not cover *Personal Cloud* storage, non-profit encyclopedias, and non-profit educational/scientific repositories. Websites which automatically reproduce or refer to significant amounts of copyrighted works have to conclude fair and balanced agreements with all rights-holders.

Chapter 5

Solid Prototype

As explained in Section 1.1, the initial motivation of this thesis was to accelerate the adoption of PDSs. After researching the available options, *Solid* that was showcased in Section 3.7.3 was chosen. Unlike other solutions is *Solid* an open-source software and allows to save various types of data without requiring users to implement additional functionality. Additionally, *Inrupt*, the company behind *Solid*, provides a free data store for new users, which makes it ideal as an entry system.

5.1 Proposal and Idea

The idea to attract new users to PDSs is to enter the market as *Facebook* did. *Facebook* started in a niche market only for students of *Harvard University* [36]. A niche market could provide a way to pull new users to PDS solutions as they currently lack an adequate solution for their problem. The thesis project explores how applications can use *Solid* as a *Personal Data Store*.

The proposed application maintains metadata about movies and TV series, such as watched movies, owned movies, and movies which the user wants to watch in the future. Metadata like this can currently be accessed on various platforms through viewing history, but references are only linked to the specific platform. In order to make that data reusable across multiple platforms, it needs to be linked to other sources within the web.

5.1.1 Linking and Discoverability

In order to make the data reusable across multiple applications and platforms, common reference sources have to be defined and referenced. Entities get linked to the *Internet Movie Database* (IMDB)[1]—an established reference platform for movies and TV series—to achieve this. Linking to a common and known identifier (in this case the entry id at IMDB) establishes a common reference point, but not a common data structure on how the data gets represented. The *Linked Data* specification [79] is utilized to save data as *Linked Data* using the RDF. This allows other applications to explore the data schema using an ontology without the need to know the exact schemas. Additionally, the widely

[1] https://www.imdb.com/

Program 5.1: Example for RDF encoded data.

```
<?xml version="1.0"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:cd="http://www.recshop.fake/cd#">
<rdf:Description
 rdf:about="http://www.recshop.fake/cd/Empire Burlesque">
  <cd:artist>Bob Dylan</cd:artist>
  <cd:country>USA</cd:country>
  <cd:company>Columbia</cd:company>
  <cd:price>10.90</cd:price>
  <cd:year>1985</cd:year>
</rdf:Description>
<rdf:Description
 rdf:about="http://www.recshop.fake/cd/Hide your heart">
  <cd:artist>Bonnie Tyler</cd:artist>
  <cd:country>UK</cd:country>
  <cd:company>CBS Records</cd:company>
  <cd:price>9.90</cd:price>
  <cd:year>1988</cd:year>
</rdf:Description>
.
.
.
</rdf:RDF>
```

used *schema.org*[2] ontology is used wherever possible as it is the leading standard in the web and widely used. When it is not possible to use a predefined schema, a new ontology is created that refers to known ontologies. The OWL ontology [80] is used to link an entry to the common identifier as it provides the `sameAs` relation that will be used to link entities together.

5.1.2 Storing

To save the created entries RDF—see Program 5.1—is used with the *Turtle* syntax as shown in Program 5.2. The RDF encoded data is stored in plain text files, which has the advantage of compatibility. Even if RDF is end-of-lifed, the files could still be processed by writing a parser for them. *Solid*—see Section 3.7.3—is used for hosting and querying of the data, as it provides a *Linked Data Platform* that can naturally deal with such kinds of data.

5.1.3 Data Retrieval and Querying Data

The interaction between the client application and *Solid* is planned to use REST operations. For advanced queries, *SPARQL* is used as this allows executing semantic queries against multiple files at once. Because the *SPARQL* query needs to be sent to the *Solid* server, the *SPARQL over HTTP* specification is used. With *SPARQL over HTTP*, the

[2]https://schema.org/

Program 5.2: Example for RDF encoded data in the *Turtle* format.

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

<#green-goblin>
    rel:enemyOf <#spiderman> ;
    a foaf:Person ;    # in the context of the Marvel universe
    foaf:name "Green Goblin" .

<#spiderman>
    rel:enemyOf <#green-goblin> ;
    a foaf:Person ;
    foaf:name "Spiderman" .
```

Program 5.3: Example of *Linked Data* encoded in the *JSON-LD* format.

```
{
  "@context": "http://schema.org",
  "@type": "ScreeningEvent",
  "name": "Captain Marvel",
  "description": "Captain Marvel movie screening.",
  "location": {
    "@type": "MovieTheater",
    "name": "Cineplex",
    "screenCount": 8
  },
  "workPresented": {
    "@type": "Movie",
    "name": "Captain Marvel",
    "sameAs": "https://www.imdb.com/title/tt4154664/"
  },
  "inLanguage": "en"
}
```

basic query will be appended to the HTTP query string in URL-encoded form. If for example the *SPARQL* query

```
SELECT * WHERE { ?s ?p ?o. }
```

is encoded and appended a request might look like the following:

```
GET /test/?query=SELECT%20*%20WHERE%20%7B%20%3Fs%20%3Fp%20%3Fo%20.%20%7D HTTP/1.1
Host: example.org
```

When retrieving data from *Solid*, different formats can be requested via the standard content negotiation of the browser. For the client application the *JSON-LD* standard shown in Program 5.3 will be used. *JSON-LD*, in theory, should allow the client application to process the data the same way as standard JSON.

5.2 Implementation

This section explains the implementation of the application prototype.

5.2.1 Technology

For the implementation of the metadata application—called *CollectIt*—the *Angular* framework [9] from *Google*, as well as *TypeScript* [30] from *Microsoft*, are used. To ease the styling of the application, the *Clarity Design System* [13] is used. *Clarity* was created by *VMWare*[3] to define a set of styles and components that allows rapid prototyping, as well as the creation of polished products.

5.2.2 Interaction

To interact with *Solid*, the `solid-auth-client` [26] library is used. The library, written in *JavaScript*, allows applications to securely log in to *Solid* to read and write data from it. While the built-in authentication works as expected, the functionality for reading and writing data is minimal and lacks functionality required for the planned prototype. Additionally, the documentation of the library and of *Solid* is limited and misses important details on how data should be saved and how stored data can be explored. The mentioned points are the foundation of *Personal Data Stores* and need to be solved.

5.2.3 Architecture

To effectively use *Solid* as a PDS system, one of the most important aspects is to link and discover data. This is achieved by saving the information as *Linked Data* to the *Solid* server with a reference to a common identifier for movies and TV series called IMDB[4].

The architecture of this project is shown in Figure 5.1. The *Solid* user pod—shown in Figure 3.8—is acting as a storage space for the data and includes no business logic. Everything needs to be performed within the *Angular* application. This results in affordable servers and hosting, but forces developers of the client applications to guarantee that data is fully valid when saving and querying. For saving of the data, the already explained *SPARQL over HTTP* standard is used. There is no need to register the application with the *Solid* server. Instead, users simply have to log in to the data pod. Subsequently, the application can start to query data from the pod. Data written by the *CollectIT* application and data written by other applications need to be differentiated. Data that was created by the client application is known and can be expected on the storage location that the application saves its data to. If another application wants to access the data created by *CollectIT*, it would first have to explore the data structure saved on the *Solid* pod to find it. After querying the data, the applications can enrich, modify, and change the data freely. The applications only provide a view on the data and can be exchanged easily.

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

[4] https://www.imdb.com/

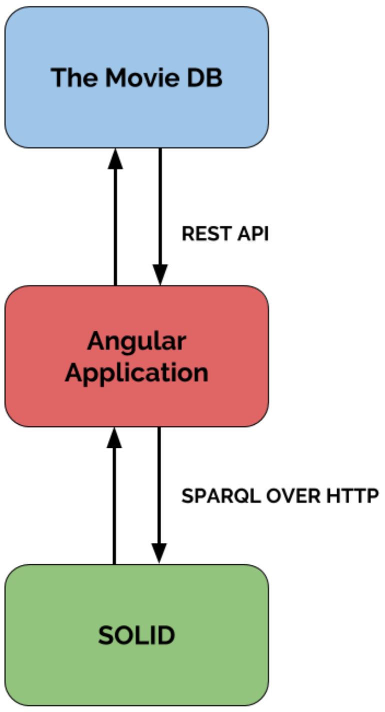


Figure 5.1: Basic architecture of the *CollectIT* application.

5.2.4 Data Structures

Only basic information and references are saved to the pod to avoid synchronization problems. The saved references point to resources on other servers where clients can find more information about the entity. Currently, however, the amount of public *Linked Data* servers or websites is limited as the lack of maintained sources for movies and TV-series shows.

The initial idea for data handling in the frontend application was to use annotated *TypeScript* classes, as shown in Program 5.4, to enhance the JSON underneath with *Linked Data* references. Due to the unavailability of a library which handles the transformation of the annotations, a solution was attempted but was given up due to failing to provide accurate mappings for edge cases such as nested structures.

The next idea was to write a mapper which allows developers to manually handle the edge cases to allow the mapping of JSON-objects to RDF using a mapping file. An existing mapper [31] was ported to *TypeScript* and adapted for this prototype. In Program 5.5 a mapped entry for a movie can be seen. Because of the generic implementation of the mapper, the output result is enormous and saves a lot of unnecessary data. Because each movie or TV-series has to be saved separately due to the triplet structure, the idea with the mapper was discarded. Instead, an alternative approach was chosen, in which the frontend only works with plain JSON objects and adds the *Linked Data* annotations only when data is saved was chosen. Instead of sending the result object

Program 5.4: *TypeScript* classes annotated with *Linked Data* references.

```
@JsonldType('https://schema.org/Movie')
export class Movie {
 @JsonldId()
 id: string;
 @JsonldProperty('http://schema.org/name')
 name: string;
 @JsonldProperty('http://schema.org/creationDate')
 dateCreated: Date;
 @JsonldProperty('http://www.w3.org/2002/07/owl#sameAs')
 sameAs: SameAsDTO[];
}
```

via the HTTP body, the *SPARQL* capabilities of the *Linked Data Platform* are used. For this, the *SPARQL over HTTP* standard is used where the *SPARQL* query is encoded and sent via the query string. An example query that creates a movie in the *Solid* data pod is shown in Program 5.6. To store metadata for a movie, a query as shown in Program 5.7 can be executed. Thereby the previously created movie gets referenced in the object property by URI. The *WebID* placeholder is replaced with the URL of the *WebID* of the user. As these examples show does *SPARQL* make it much easier to handle the storage of the metadata compared to the previously explored options.

5.3 State of the Prototype and Learnings

A *Personal Data Store* solution needs to be as open as possible and has to allow multiple applications to work with the same data. *Solid* allows saving of all kinds of data as it does not have a fixed schema of how data has to be saved. At first, this seemed to be great in terms of flexibility, but during implementation, it became clear that missing standards and no clear guidelines are a big problem. Furthermore, a considerable amount of knowledge in different web standards is needed to develop for *Solid*. This creates a situation that acts as a showstopper for developers and companies.

A PDS solution that aims to be successful needs to have the flexibility of *Solid*, a secure and easy to use server, unequivocal responsibility for the data and its validity, as well as an easy and secure way to manage permissions for data access. The server needs to be more than a simple file server as otherwise, a single application could corrupt all data files by changing parts of the files or redirecting URIs to invalid addresses. The server should provide an easy way for users to control access to the data for each application. *Solid* only supports the *WebACL* standard that provides the server-side granularity only on a file basis, but that does not offer an accessible way to decide rights on a client per client basis. *Solid* currently provides no way to register and maintain different clients. Instead, applications are handled the same way as users. *Solid* is a great idea built on open standards with the goal to change the internet. This goal might be reached once the technology has matured and *Solid* offers a smooth transition for developers and existing applications. The author feels that *Solid* is currently too young and misses features that could fuel adoption. What *Solid* however achieved is to spark

Program 5.5: Excerpt of a mapped data entry using the mapper.

```
{
    "subject": {
      "termType": "NamedNode",
      "value": "https://max.solid.community/public/metadata/movie/60a4b0a6-14cb-462f
    -a4d2-badb9ae7f595-1548861938"
    },
    "predicate": {
      "termType": "NamedNode",
      "value": "https://schema.org/dateCreated"
    },
    "object": {
      "termType": "Literal",
      "value": "2019-01-30T15:25:38Z",
      "datatype": {
        "termType": "NamedNode",
        "value": "http://www.w3.org/2001/XMLSchema#dateTime"
      }
    },
    "why": {
      "termType": "NamedNode",
      "value": "https://max.solid.community/public/metadata/movie/60a4b0a6-14cb-462f
    -a4d2-badb9ae7f595-1548861938"
    }
  },
  {
    "subject": {
      "termType": "NamedNode",
      "value": "https://max.solid.community/public/metadata/movie/60a4b0a6-14cb-462f
    -a4d2-badb9ae7f595-1548861938"
    },
    "predicate": {
      "termType": "NamedNode",
      "value": "http://max.test/ns/metadataOntology#imdb"
    },
    "object": {
      "termType": "Literal",
      "value": "tt1731141"
    },
    "why": {
      "termType": "NamedNode",
      "value": "https://max.solid.community/public/metadata/movie/60a4b0a6-14cb-462f
    -a4d2-badb9ae7f595-1548861938"
    }
  },
  {
    ...
  }
]
```

Program 5.6: Example for storing a movie using *SPARQL*.

```
INSERT DATA
{
  <${baseURL}>  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org
    /CreativeWork>, <http://schema.org/Movie>, <http://schema.org/Thing>;
    <http://schema.org/name> <${movieTitle}>;
    <http://schema.org/dateCreated> <${creationDate}>;
    <http://dbpedia.org/ontology/imdbId> <${imdbURL}>;
    <http://www.w3.org/2002/07/owl#sameAs> <${tmdbURL}>.
}
```

Program 5.7: Example for storing an action using *SPARQL*.

```
INSERT DATA
{
    <${actionURL}> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.
    org/Action>, <http://schema.org/ConsumeAction>, <http://schema.org/WatchAction>,
     <http://schema.org/Thing>;
    <http://schema.org/object> <${movieURL}>;
    <http://schema.org/agent> <${webid}>.
}
```

further research in the area of PDSs.

Due to all the previously mentioned reasons, the development of the *Solid* prototype has been stopped. Instead, a new PDS solution will be developed that tries to find solutions to the problems left out by *Solid*. The goal is to explore possible ways of authentication, access management, and simple interfaces while still providing the flexibility of *Solid*.

Chapter 6

Selfbox Prototype

This chapter explains the prototype for a PDS called *Selfbox*. *Selfbox* is designed from the ground up and tries to solve problems encountered while implementing an application for *Solid*.

6.1 Idea and Goal

The basic concept is to create a PDS based around the established and well understood standard *OAuth 2.0*—see Section 4.3.1. Instead of sandboxing client applications for security reasons, as other PDSs solutions, the *Selfbox* prototype focuses on the data that leaves the server. The goal is not to create a perfect solution, but to explore ways that would allow the usage of PDSs within the current web. To achieve this, the PDS does not enforce anything onto developers or users. If a third-party application wants to continue saving data to its servers, it should be able to do so. However, the developer of the application might want to use the PDS for authentication. This way, the PDS solution can continue to evolve by the requirements of its users and developers.

6.1.1 Authorization and Permissions

The server needs to be a *OAuth 2.0* provider to allow it to get used for login purposes on third-party websites. Therefore like existing platforms such as *Facebook*, *Google*, and others currently do, the server needs to allow other applications to register themselves as a client with the PDS. Normally the registration process is done by the developers by requesting a key and copying it in their application. This process would not make sense for the prototype as an application developer can not manually register his website on thousands of PDS. Instead, possible ways need to be explored of how the registration process could be done in a more decentralization friendly way. After an application has registered itself with the PDS, users should be able to control the data flow using *OAuth* scopes.

The server should also implement the *OpenID Connect* standard to provide third-party applications with information about the user's identity. *OpenID Connect* allows the server to be an identity provider that allows the user to login to any existing application that supports *OpenID Connect* for the login.

6.1.2 Storage and Exploration

Different ways need to be explored that would allow developers to save various kinds of data while maintaining flexibility and validation. Without validation, a single application could corrupt data and impact the integrity of other data. Because of the validation problem, the idea is to use a *NoSQL* database that supports schemas but does not require them. Additionally, users should be able to make sense of the stored data. Therefore, the representation of entities should map the real world as close and as naturally as possible. For this, a graph database is used that supports the property node model—see Section 4.2.2. Using this model, users can query for nodes of a given type and explore the data connected to these nodes.

A secure and extensible way that should require as little overhead as possible, to add new types is required, to demonstrate that various kinds of data are supported.

6.2 Implementation

The prototype is built with the *Nest* framework [19]. *Nest* is a framework inspired by *Angular* [9] and *Spring* [27] that aims to provide an enterprise framework for *NodeJS* [20] applications. It uses *TypeScript* [30] and decorators to glue together the application.

6.2.1 Authentication and Authorization

To implement *OAuth 2.0* and *OpenID Connect*, a module named `OauthModule` is created. This module implements a `configure` method that allows attaching middlewares to a route. The `configure` method then applies the `ensureLoggedIn` middleware from the `connect-ensure-login` [14] library. This library works together with *Passport* [24] to check if incoming requests are authenticated. If users are not logged in, they first need to login into the server. If a user has logged in, the following middleware checks which client application is trying to gain access. If the client is unknown, the server returns an error stating that this is an unknown client. Unknown in this context means that no entry or information about the client was saved on the server yet. If the client is known, the next step is to check if the resource owner has previously granted access to the client. If not, the server asks the resource owner for permission to grant access to the client. If however the client was previously authorized, the client gets immediately approved. Using this workflow, client applications can continuously gain access to the data without having to await approval of the user all the time. This process also results in a better user experience as users do not need to authorize clients every time. Another middleware configuration handles the `decision` endpoint that displays the permission dialog for the user when a client application first asks for access to the server. The last middleware configuration deals with the token endpoint. Thereby a posted token gets passed to the `oauth2rize` [21] library for further processing and error handling.

The `OauthController` exposes multiple routes needed for the *OAuth 2.0* and *OpenID Connect* workflows that is secured by the previously mentioned middlewares. One of the more significant problems that had to be solved for the login process was how the client learns about the correct routes for its actions. *OAuth 2.0* does not define any routes for this, so each client would have to be configured to work with the prototype. The server exposes the `.well-known/openid-configuration` endpoint that implements the *OpenID Connect Discovery 1.0* specification to overcome this issue. Using this information, the client can fetch the information about the server and configure itself automatically.

OpenID Connect Discovery

The *OpenID Connect Discovery* [69] document exposes all available scopes and claims that the PDS currently supports. This way, applications know what data they can request from a *Selfbox* and what data is unavailable. Program 6.1 shows the discovery document from the *Selfbox* prototype. The `issuer` field describes the identity provider itself. The `authorization_endpoint` is used to request the authorization itself and requests one of the supported response types specified in `response_types_supported`. In the request, the client can request multiple scopes from the `scopes_supported` array whereby some scopes cover the described claims in `claims_supported`. If the client requests an `id_token`, the returned token will be validated using the algorithm specified in `id_token_signing_alg_values_supported` and the keys provided by the `jwks_uri` endpoint [68].

Dynamic Client Registration

To support dynamic client registration, *Selfbox* implements the *OAuth 2.0 Dynamic Client Registration Protocol* [64]. Clients can use the `registration_endpoint` to send an *HTTP POST* request with the client metadata and the scopes the application needs. As the response, the server answers with the generated `client_id` and `client_secret`. Afterwards, the regular login process is triggered, and the client will be identified by the `client_id`, the application received before. The user still has to approve the scopes that the client requested.

Authorization Workflows

The prototype supports all *OAuth 2.0* and *OpenID Connect* workflows to make it usable with all kinds of applications. For this thesis, however, only the implicit grant and the authorization code grant have been tested. Figure 6.1 shows the full workflow of a *Single Page Application.* It is important to note that real clients serve multiple users and therefore need to prompt the user for the URL of the user's PDS system. The implemented frontend prototype currently only supports local deployment, but can easily be adapted. The `oauth2rize` [21] library is used to implement the implicit grant. The following example showcases the registration of the workflow with the server:

```
this.server.grant(oauth2orize.grant.token(this.issueToken.bind(this)));
```

The library takes care of parsing the requests and executing the registered functions to generate the needed tokens. For the implicit grant, a function to issue an access token needs to be implemented. The logic for this function is showcased in Program 6.2. To create the actual access token, a signed JWT token is created and returned as showcased in Program 6.3. The token contains the targeted audience (the client application) the subject (the user) and the agreed scope. To sign the token, a secret is used that is passed

Program 6.1: Overview over the *OpenID Connect Discovery* document from the *Selfbox* prototype.

```
{
    "issuer": "http://localhost:3000",
    "authorization_endpoint": "http://localhost:3000/oauth2/auth",
    "token_endpoint": "http://localhost:3000/oauth2/token",
    "userinfo_endpoint": "http://localhost:3000/oauth2/userinfo",
    "registration_endpoint": "http://localhost:3000/oauth2/register",
    "jwks_uri": "http://localhost:3000/oauth2/certs",
    "response_types_supported": [
        "code",
        "token",
        "id_token",
        "code id_token",
        "token id_token",
        "code id_token token",
        "code token"
    ],
    "subject_types_supported": [
        "public"
    ],
    "id_token_signing_alg_values_supported": [
        "RS256"
    ],
    "scopes_supported": [
        "openid",
        "email",
        "profile",
        "Movie:Create",
        "Movie:Read",
        "Movie:Update",
        "Movie:Delete",
        ...
    ],
    "token_endpoint_auth_methods_supported": [
        "client_secret_post",
        "client_secret_basic"
    ],
    "claims_supported": [
        "aud",
        "email",
        "email_verified",
        "exp",
        "family_name",
        "given_name",
        "iat",
        "iss",
        "locale",
        "name",
        "picture",
        "sub"
    ]
}
```

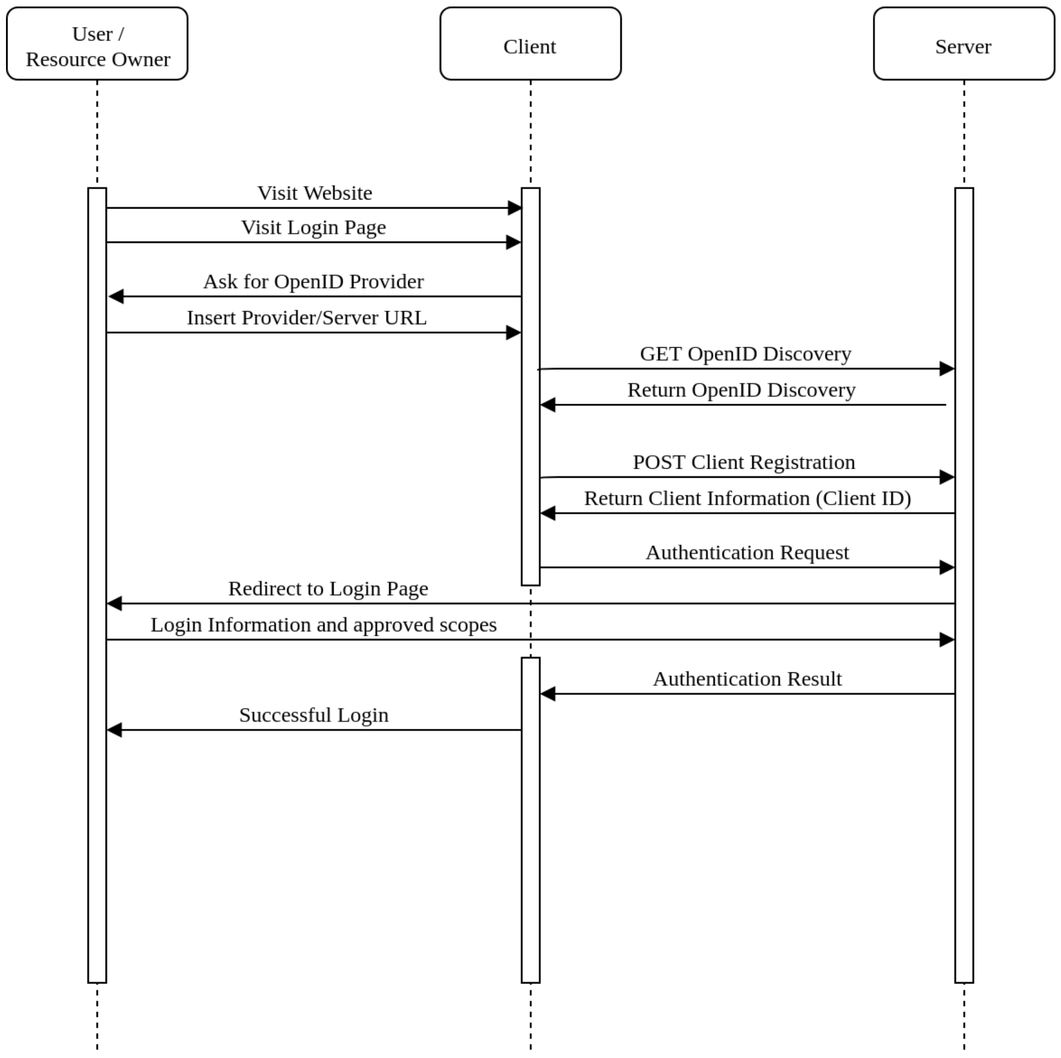


Figure 6.1: This images shows the full authorization workflow of client applications with the *Selfbox* PDS solution. This workflow includes the step of prompting the user for his *Selfbox* URL that will be required to establish the connection.

to the application using an environment variable. The access token can be used to make requests to protected resources on the server.

To get the identity of the user, the extension library `oauth2orize-openid` [22] is utilized to implement the `id_token` grant. Therefore the workflow first needs to be registered with the server:

```
this.server.grant(oauth2openid.grant.idToken(this.issueIDToken.bind(this)));
```

Next, the creation logic for the `id_token` is needed, as shown in Program 6.4. The `id_token` is a structured token in the JWT format [33]. For a valid `id_token`, the secret used to sign the token cannot be a string but needs to be an RSA-key in the PEM encoding [73]. The key is generated with *OpenSSL* [23] using the following command: `openssl genrsa -out private.pem 2048`. This key is loaded by the application and

Program 6.2: Token issuing logic for the *Authorization Code* grant implemented in the *Selfbox* prototype.

```
private async issueToken(client, user, ares, done) {
    Logger.log('Issuing Access Token', 'OauthServerService');

    const [createAccessTokenError, accessToken] = await to(this.oauthDBService.
     createAccessToken({
      clientId: client.clientId,
      userId: user.id,
      scope: ares.scope
    }));

    if (createAccessTokenError) {
      return done(createAccessTokenError);
    }

    const [createRefreshTokenError, refreshToken] = await to(this.oauthDBService.
     createRefreshToken({
      clientId: client.clientId,
      userId: user.id,
      scope: ares.scope
    }));

    if (createRefreshTokenError) {
      return done(createRefreshTokenError);
    }

    return done(null, accessToken, {
      refresh_token: refreshToken,
      expires_in: OauthConfig.accessTokenTTL
    });
}
```

Program 6.3: *Access Token* create function used to generate an JWT token with the approved scopes.

```
createAccessToken(data): Promise<string> {
    return new Promise<string>((resolve, reject) => {
        const token = jwt.sign({
            aud: data.clientId,
            sub: data.userId,
            scope: data.scope,
        }, OauthConfig.accessTokenSecret, {
            algorithm: OauthConfig.accessTokenAlgorithm,
            expiresIn: OauthConfig.accessTokenTTL
        });
        resolve(token);
    });
}
```

Program 6.4: *ID Token* create function used to generate an JWT token with the supported claims.

```
private async issueIDToken(client, user, scope, req, done) {
    Logger.log('Issuing ID Token', 'OauthServerService');

    const idToken = jwt.sign({
      sub: user.id,
      nonce: req.nonce,
      name: user.name,
      email: user.email
    }, OauthConfig.idTokenSecret, {
      expiresIn: OauthConfig.idTokenTTL,
      algorithm: OauthConfig.idTokenAlgorithm,
      issuer: OauthConfig.baseUrl,
      audience: client.clientId
    });

    return done(null, idToken);
}
```

is used as the `id_token` secret in combination with the *RS256* algorithm [34]. In order to verify the `id_token`, the client needs to check the signature with the identity of the server. To achieve this functionality, an endpoint is provided that exposes the public key part of the RSA key pair used to sign the token. The client gets the location of the route used to serve the public key via the *OpenID Discovery Document* [69]. The key is formatted in the *JSON Web Key* (JWK) format using the library `rsa-pem-to-jwk` [25].

6.2.2 Storage and Exploration

As mentioned before, a graph database is used to allow easy discovery and linking of different data nodes. After a market analysis, the *Neo4j* graph database [17] was selected. *Neo4j* uses the *Cypher* query language [45] which allows the user to write semantic queries with ease, as the following example shows:

```
MATCH (u:User {id: 1})
MATCH (m:Movie {id: 1})
CREATE (u)-[rel:USER_ACTION {action: WATCHED}]->(m)
RETURN rel { .action };
```

The concept of property graph databases was explained in Section 4.2.2. In the query example shown above, one can see this type of graph in action. In the first row, a node with the label "User" and a property named "id" with the value "1" is matched in the graph. Next, a node with the label "Movie" is searched with an "id" property with the value "1". Then a new relationship between the two nodes is created that is labeled "USER_ACTION" which has a property called "action" with the value "WATCHED". Because such a system would be too powerful and dangerous to expose directly, a layer between the client and the database is needed. One of the goals of this project is to find a way how such a layer could be implemented in a safe and in a controllable way.

One solution that was considered is to create a generic controller that would support

Program 6.5: Example that shows how movies can be queried from *Selfbox* via *GraphQL*.

```
query {
   Movie(orderBy: datePublished_desc, offset: 0) {
       id
       actionType {
         action
       }
   }
}
```

all REST operations for a single entity. Additionally, entities would be created and annotated to enable validation. On server startup, the application would look for all entities and create an instance of the generic controller for all found entities. The problem with the proposed solution is that with more entities the server would need to serve dozens of endpoints. Moreover, the support for creating relations between entities could not be solved in a type-safe manner. An alternative solution would be to create a *JSON API*[1] but these types of APIs are not used quite often. Furthermore, the number of libraries for developers to use with *JSON API*'s are minimal.

For the final solution, the decision was made to use *GraphQL* [15] as the query language for client applications. *GraphQL* allows the client to request any data from the server in one request from one endpoint [15]. *GraphQL* supports queries, mutations as well as subscriptions, which allows clients to request data, modify data, and to subscribe to get notified when the data changes. This allows developers to create rich experiences in client applications. Program 6.5 shows an example query that queries all movies ordered by publishing date and returns only the `id` and `actionType`.

To connect the *GraphQL* interface with *Neo4j*, the library `neo4j-graphql-js` [18] is used. This library allows to generate the required code from simple *GraphQL* type definitions. To integrate the library the approach by Lutz [60] was used. Inspired by the example on how to expose the *GraphQL* endpoint in *Nest* an `GqlConfigService` was developed that can be used with the `GraphQLModule` from *Nest*. The `GqlConfigService` runs on startup of the application and searches for *GraphQL* definition files in the folder that contains the application as shown in Program 6.6. After files are found, their content gets extracted and concatenated creating a single type definitions file that then gets annotated via the functions provided by `neo4j-graphql-js` as shown in Program 6.7 [62]. Afterwards, the augmented schema gets passed to the internal *Apollo* [12] server used by the `GraphQLModule`. The augmentation step is needed to resolve all *GraphQL* directives provided by `neo4j-graphql-js` and to automatically create the resolvers and query functions for the given types.

The whole process allows users and developers to add new types to the server by simply creating or uploading a *GraphQL* type definition file and restarting the server. Queries and mutations are checked by the *Apollo* server to ensure that no invalid data can be saved into the database. Client applications can be prototyped rapidly due to the exposed schema offered by the *GraphQL* endpoint.

[1] https://jsonapi.org/format/

Program 6.6: Function to merges all *GraphQL* type files that are in the source folder.

```
private getGraphQLDefinitions(): Promise<string> {
    const typeDef: string[] = [];

    return new Promise<string>((resolve, reject) => {
      glob(process.cwd() + '/src/**/*.graphql', {}, (err, files) => {
            if (err) {
                reject(err);
                return;
            }
            Logger.log('Found GraphQL Definitions [' + files.toString() + ']', '
    GraphQLConfigService');
            files.forEach((filePath: string) => {
                typeDef.push(fs.readFileSync(filePath, {encoding: 'utf8'}));
            });
            resolve(typeDef.join('\n'));
        });
    });
}
```

Program 6.7: Function to augment the *GraphQL* schema and resolves the built in schema directives.

```
private async getAugmentedSchema(): Promise<GraphQLSchema> {
    return new Promise<GraphQLSchema>(async (resolve, reject) => {
        const [error, typeDefs] = await to(this.getGraphQLDefinitions());

        if (error) {
            reject(error);
        }

        const schema = makeAugmentedSchema({
            typeDefs,
            logger: {
                log(message) {
                    Logger.error(`GraphQL Schema: ${message}`);
                }
            },
          config: { query: true, mutation: true, debug: true }
        });

        resolve(schema);
    });
}
```

6.2.3 Protection of Resources

The `GqlConfigService` also parses the type definitions file manually to calculate the available scopes used to protect the resources. The calculated scopes are exposed via the *OpenID Discovery Document* described in Section 6.2.1 and are enforced using the

Program 6.8: Annotated *GraphQL* types with `neo4j-graphql-js` directives.

```
type Movie {
    id: ID!
    title: String
    overview: String
    poster: String
    datePublished: DateTime
    imdbRating: Float
    tmdbId: Int
    actionType: [UserAction]
    genres: [Genre] @relation(name: "IN_GENRE", direction: "OUT")
}

type Genre {
    id: ID!
    name: String
    tmdbId: Int
    entities: [Movie] @relation(name: "IN_GENRE", direction: "IN")
}
```

`graphql-auth-directives` library [16]. As a result the generated resolvers, mutations and queries are annotated with `@hasScope` by `neo4j-graphql-js`. The JWT secret needs to be provided as an environment variable to allow the `graphql-auth-directives` library to decode the JWT access token and parse the scopes of the client.

Sensitive Data Storage

Because users and developers might extend the server with a malicious extension that aims to expose the hashed passwords of the user, an additional database for sensitive login information and *OAuth* data is added. That way, it is impossible to gain access to sensitive information via the *GraphQL* endpoint. This additional database is a *SQLite* database [28] because traffic will be limited and infrequent. To access the database, the library *TypeORM* [29] is used that allows developers to efficiently manage data access without the need to write SQL queries. The setup process thereby works similar to the previously described initialization. First, a connection to the database is established. This is done by creating a provider for the database connection, as shown in Program 6.9.

In the snippet shown in Program 6.9 *TypeORM* searches for all annotated *TypeScript* classes and generates the database structure accordingly. Program 6.10 shows one of the annotated classes used by the prototype. Afterwards, it is possible to create a typed instance for a given entity which allows to access data by creating a provider with a factory method. User management and setup was not part of this prototype. Therefore no security measures concerning password hashing have been taken.

6.3 Frontend

For the frontend of this prototype, the previous application from the *Solid* prototype was adapted. The same technologies, as explained in Section 5.2.1, are used. Additionally,

Program 6.9: *TypeORM* connection provider factory.

```
{
    provide: 'DATABASE_CONNECTION',
    useFactory: async () => await createConnection({
      type: 'sqlite',
      database: Env.DbFile,
      entities: [
        __dirname + '/../../**/*.entity{.ts,.js}'
      ],
      synchronize: true
    })
}
```

Program 6.10: Example of entity with *TypeORM* annotations.

```
@Entity()
export class UserEntity {
  @PrimaryColumn()
  _id: number;

  @Column()
  @Generated('uuid')
  id: string;

  @Column()
  username: string;

  @Column()
  password: string;

  @Column()
  familyName: string;

  @Column()
  middleName: string;

  @Column()
  givenName: string;

  @Column()
  email: string;
}
```

the library `angular2-apollo` is used for communication with the server [11]. Apollo provides a client to interact with a *GraphQL* endpoint.

6.3.1 Authorization

To authorize the frontend with *Selfbox* the `angular2-oidc` library is used [10]. The library needs to be configured as shown in Program 6.11. It is important that the

Program 6.11: Configuration of the authorization library in the frontend.

```
export const authConfig: AuthConfig = {
    issuer: 'http://localhost:3000',
    oidc: true,
    clearHashAfterLogin: false,
    redirectUri: window.location.origin + '/index.html',
    clientId: 'abc123',
    responseType: 'token id_token',
    showDebugInformation: true,
    disableAtHashCheck: true,
    sessionChecksEnabled: false,
    scope: 'openid profile email Movie:Create Movie:Update Movie:Delete Movie:Read
     Genre:Write Genre:Read User:Create User:Read',
};
```

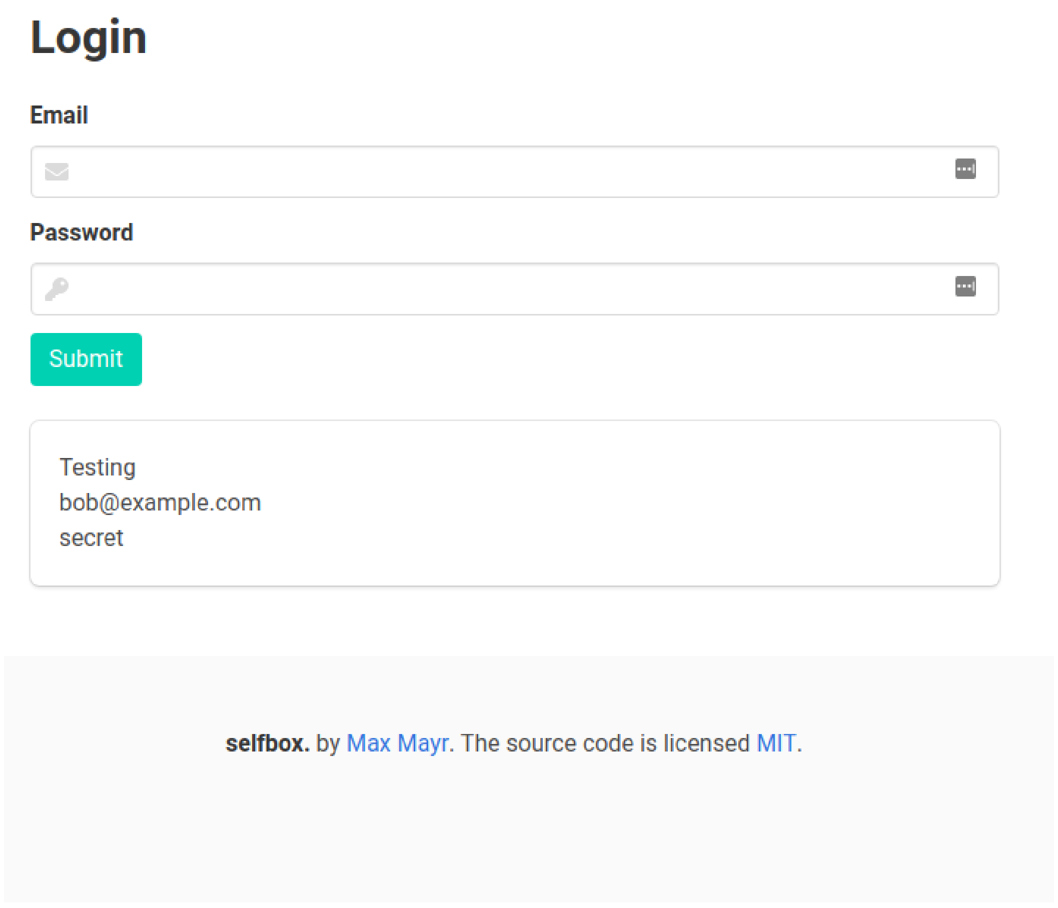
Program 6.12: Trigger auto-configuration of the library using the discovery document.

```
@Injectable({
    providedIn: 'root',
})
export class AuthService {
    constructor(private oauthService: OAuthService) {
        this.oauthService.configure(authConfig);
        this.oauthService.tokenValidationHandler = new JwksValidationHandler();
        this.oauthService.requireHttps = false;
        this.oauthService.oidc = true;
        this.oauthService.setStorage(localStorage);
        this.oauthService.loadDiscoveryDocumentAndTryLogin();
    }

    getUserId(): string {
        const claims = this.oauthService.getIdentityClaims();
        return claims['sub'];
    }
}
```

response type is `token id_token` if the application needs access to resources as well as to the identity of the user. After defining the configuration, the client can load the discovery document as showcased in Program 6.12. A real application would query the user for the URL of his PDS solution beforehand. After the previous steps the frontend can initialize the implicit grant workflow by calling `this.oauthService.initImplicitFlow()`. That call will then redirect the user to the server, as shown in Figure 6.2. After the user logged in into the server, the server asks for permission to grant the application access to the server, as shown in Figure 6.3. If the user approves the request, the server will redirect the user to the client application. Thereby the requested tokens will be returned in the URL string.

To use *GraphQL*, the *Apollo Client* needs to send the retrieved access token as "Authorization" header with the "Bearer" prefix, as shown in Program 6.13.



Testing
bob@example.com
secret

selfbox. by Max Mayr. The source code is licensed MIT.

Figure 6.2: *Selfbox* login screen.

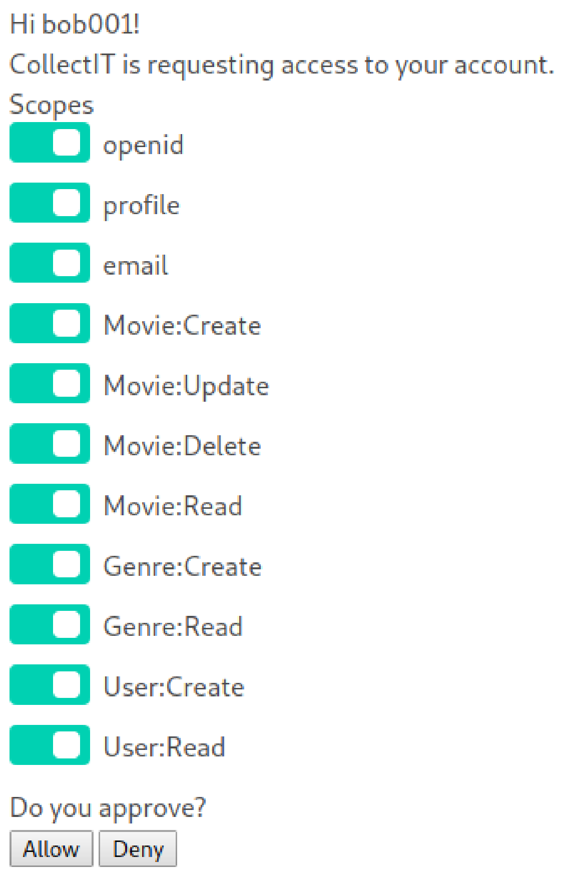
6.3.2 Interaction

After a successful login, the users can visit the dashboard shown in Figure 6.4 where they can see all movies that have been added to their PDS system. Each movie can have multiple associated actions that can be loaded with the movie. Depending on what actions are present, different options are shown in the cards in Figure 6.4.

The frontend uses the provided *GraphQL* interface from *Selfbox* to add, update, and delete entities from the PDS. Program 6.14 shows the code to add a movie entry to the PDS.

6.4 State of the Prototype and Outlook

This prototype shows a PDS solution that offers a simple interface, the possibility to save various kinds of data and provides validation for that data. The major obstacles, such as the extension of new data types and access controls, have been solved by using *GraphQL* types and provide a mechanism that generates all boilerplate code automatically. *GraphQL* proofed to be an excellent choice for this project as websites only need

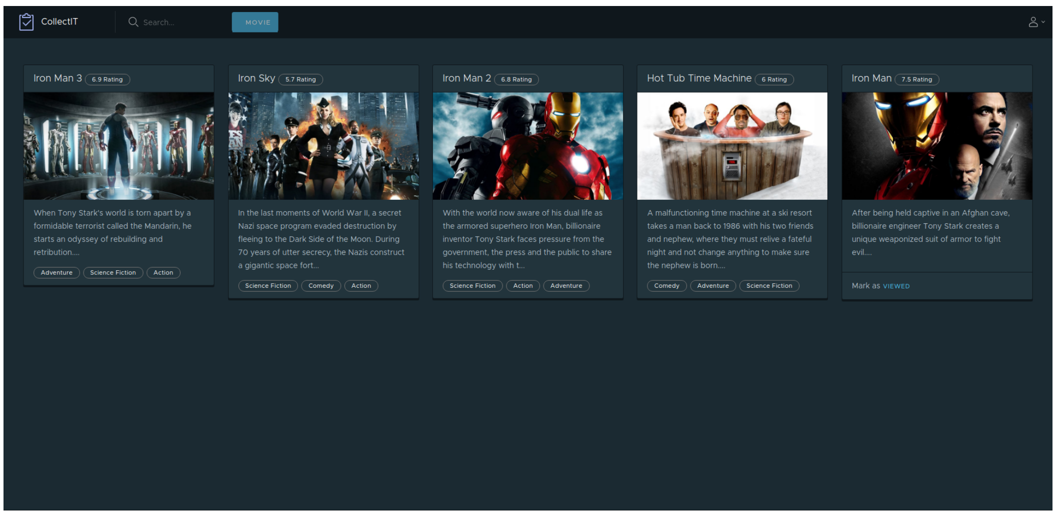


Hi bob001!
CollectIT is requesting access to your account.
Scopes

- [✓] openid
- [✓] profile
- [✓] email
- [✓] Movie:Create
- [✓] Movie:Update
- [✓] Movie:Delete
- [✓] Movie:Read
- [✓] Genre:Create
- [✓] Genre:Read
- [✓] User:Create
- [✓] User:Read

Do you approve?

Figure 6.3: The *Selfbox* authorization dialog allows users to allow or reject scopes requested by an application.



Iron Man 3 6.9 Rating

When Tony Stark's world is torn apart by a formidable terrorist called the Mandarin, he starts an odyssey of rebuilding and retribution....

Adventure · Science Fiction · Action

Iron Sky 5.7 Rating

In the last moments of World War II, a secret Nazi space program evaded destruction by fleeing to the Dark Side of the Moon. During 70 years of utter secrecy, the Nazis construct a gigantic space fort...

Science Fiction · Comedy · Action

Iron Man 2 6.8 Rating

With the world now aware of his dual life as the armored superhero Iron Man, billionaire inventor Tony Stark faces pressure from the government, the press and the public to share his technology with t...

Science Fiction · Action · Adventure

Hot Tub Time Machine 6 Rating

A malfunctioning time machine at a ski resort takes a man back to 1986 with his two friends and nephew, where they must relive a fateful night and not change anything to make sure the nephew is born....

Comedy · Adventure · Science Fiction

Iron Man 7.5 Rating

After being held captive in an Afghan cave, billionaire engineer Tony Stark creates a unique weaponized suit of armor to fight evil....

Mark as VIEWED

Figure 6.4: Screenshot of the *CollectIT* dashboard.

Program 6.13: Configuration of the *GraphQL Apollo* client.

```
export function createApollo(httpLink: HttpLink) {
    const http = httpLink.create({uri: 'http://localhost:3000/graphql'});

    const authLink = new ApolloLink((operation, forward) => {
        const token = localStorage.getItem('access_token');
        operation.setContext({
            headers: {
                'Authorization': token ? `Bearer ${token}` : ''
            }
        });
        return forward(operation);
    });

    return {
        link: authLink.concat(http),
        cache: new InMemoryCache(),
    };
}

@NgModule({
    exports: [ApolloModule, HttpLinkModule],
    providers: [
        {
            provide: APOLLO_OPTIONS,
            useFactory: createApollo,
            deps: [HttpLink],
        },
    ],
})
export class GraphQLModule {
}
```

one request to query all the data they need.

The next steps for the prototype are to strengthen security measures and implement best practices in that area such as password hashing. Furthermore, a setup wizard will be implemented that allows every user to setup the PDS without the need for any technical knowledge. Once these basic features are finished, additional functionality such as file storage will be implemented. The project is designed mainly for personal use but might also get commercialized in the future if there is any demand.

Program 6.14: Function to add a movie via *GraphQL* mutation.

```
addMovie(movie: MovieModel) {
    const createMovie = gql`
        mutation createMovie($title: String!, $overview:String, $poster:String, $datePublished:String, $tmdbId: Int, $imdbRating: Float) {
            CreateMovie(title: $title, overview: $overview, poster: $poster, datePublished: { formatted: $datePublished }, tmdbId: $tmdbId, imdbRating: $imdbRating) {
                id
            }
        }
    `;

    return this.apollo.mutate({
        mutation: createMovie, variables: {
            ...movie
        }
    })
    .pipe(
        map((graphQLResult: FetchResult<{ CreateMovie: { id: string } }>) => graphQLResult.data.CreateMovie.id),
        switchMap((movieId: string) =>
            forkJoin(
                _.map(movie.genres, (genre: GenreModel) => this.getGenreByTmdbId(genre.tmdbId)
                    .pipe(
                        map((genreResult: ApolloQueryResult<{ Genre: { id: string }[] }>) => genreResult.data.Genre[0].id),
                        switchMap((genreId: string) => this.addGenreToMovie(movieId, genreId))
                    )
                )
            )
        ),
        map<any[], string>((queryResult: any[]) => queryResult[0].data.AddMovieGenres.from.id)
    );
}
```

Chapter 7

Conclusion

This section concludes the collected experiences that were gained during the creation of this thesis and explores the author's views on the evolution of the web.

7.1 Future of the Web

The paper elaborated what is possible with current web technologies. If PDS will ever reach mainstream is unknown, however one thing is inevitable. The web will continue to evolve as it always has since its creation; where that will lead to nobody can answer for sure, but some educated guesses can be made. If current trends continue, then the web will be more regulated as it is now and possibly even more separated as developments in authoritarian states show. Regulations which force users to give up their privacy are already demanded by some governments and might be put into regulation in the next years. However, politicians seem to ignore the current evolution the web is undergoing.

Startups focused on decentralization are founded all over the globe offering existing services in a decentralized matter with secure, state-of-the-art encryption and built-in privacy. These services are currently uncomfortable to use, but increasing concerns about privacy drive users away from current centralized applications like *WhatsApp*. The messenger even plans to show targeted advertisements in the client applications to increase revenue at the cost of the user [76]. Trends like this, on the cost of user privacy, can be seen everywhere, and are notice by users which start to moving to different services. Therefore monopolies already fear the loss of users and revenue. The situation is so critical already that companies like *Google* and *Facebook* announced that they are going to improve privacy on their platforms. This promise slows down the change but can not stop it as centralized platforms cannot compete with decentralized applications.

One of the first areas that will change rapidly is storage. As costs, service level agreements, redundancy, flexibility, and security of decentralized applications outperform existing solutions by a factor of ten [50]. Technologies related to block-chain, created much discussion in the last years with *Bitcoin*[1] leading the way. *Bitcoin* is an example for the *Web 3.0* that focuses only on currency transfer. It inspired hundreds of other crypto-currencies each one with a particular focus. One of the leading solutions

[1] https://bitcoin.org/en/

is *Ethereum*[2], a platform for the next generation of the web based on a programmable block-chain under the hood. *Ethereum* allows the creation of so-called *DApps* applications that process requests and get hosted on the decentralized platform [41]. That means the logic of these apps runs on the distributed power of the network instead of a single point of failure like a server farm. *Ethereum* is currently competing for market share with other decentralized platforms based on block-chain technology.

Decentralization is one aspect of *Web 3.0*. Another commonly agreed characteristic is the semantic web, a semantic web that will look different than the one we imagine now. The semantic web will not be based on semantic annotations on websites but on structured interfaces between servers where autonomous agents will interact with each other.

7.2 Goals and Experiences

The goal of this thesis was to explore the possibilities of *Personal Data Stores*. The reader was introduced to the topic of PIM and examples that show the relevance and importance of this topic as the amount of data continually grows. New data that will get collected through IoT and other innovations will need to be saved. This thesis showed a way to deal with that data in a structured and semantic way. PDSs will always have to adapt to new data and data structures. Representing data in its natural form, in a semantic way, using relationships and nodes in a graph provides a future proof solution and allows semantic exploration.

The thesis continued to introduce the reader to the current state of the art for PDSs and the problems with these solutions. Thereby it is important to note that the thesis looked at the topic not only from a developer perspective but also from user and company perspectives. PDSs should be a win-win situation for all involved parties. Some parties, such as companies need to have the possibility to gradually switch to PDSs, instead of forcing them to support two different ecosystems. But not only the users' or companies' needs are essential. Politics introduce more and more regulations and laws that aim to improve the life of users. The thesis highlighted regulations that are the most important, not just for Europe but also for the world as a whole. The thesis then dived further into *Solid* and explored ways to utilize it as a PDS solution. During the implementation the absence of essential features like cryptography and protection of the data were encountered, that lead to the creation of a new PDS solution. The prototype explored ways of how data could be protected and saved universally. The presented solution demonstrates how PDSs can coexist an enhance the current web by providing a central point of identity for the owner of the PDS. Companies benefit from this as they do not have to implement and maintain authorization and authentication solutions. Furthermore, *Selfbox* shows how various data can be saved while ensuring that only valid data is accepted, protecting other implementations from corrupt data that they can not handle.

The thesis hopes to motivate others to start researching in this area as well. The problems in the web need a solution as soon as possible to ensure free speech and an open discussion in the years to come.

[2]https://www.ethereum.org/

Appendix A

Content of the CD-ROM

A.1 PDF-Files

Path: /thesis

thesis_2019.pdf	Thesis as PDF document

A.2 Prototypes

Path: /projects

README.md	*Markdown* document with general information about the projects
/collectIT	*Solid* prototype of *CollectIT* application
/selfbox	*Selfbox* prototype—contains `README.md` with instructions
/selfbox-metadata	*Selfbox* prototype of *CollectIT* application

References

Literature

[1] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor.* Paw Prints, June 2008 (cit. on p. 17).

[2] Ryan Boyd. *Getting Started with OAuth 2.0.* O'Reilly Media, Inc., 2012 (cit. on pp. 21, 22).

[3] Ann Cavoukian. "Privacy by Design: Origins, meaning, and prospects for assuring privacy and trust in the information era". In: *Privacy Protection Measures and Technologies in Business Organizations: Aspects and Standards.* Ed. by George O.M. Yee. IGI Global, 2012. Chap. 7, pp. 170–208 (cit. on p. 26).

[4] Emil Eifrem, Jim Webber, and Ian Robinson. *Graph Databases.* 2nd ed. 2015 (cit. on pp. 19–21).

[5] William P. Jones. *Keeping Found Things Found: The Study and Practice of Personal Information Management.* Morgan Kaufmann Publishers, 2008 (cit. on pp. 3, 4).

[6] William P. Jones and Jaime Teevan. *Personal Information Management.* University of Washington Press, July 2011 (cit. on p. 6).

[7] Max Van Kleek and Kieron OHara. "The Future of Social Is Personal: The Potential of the Personal Data Store". In: *Computational Social Sciences.* Ed. by Daniele Miorandi et al. Springer International Publishing, 2014, pp. 125–158 (cit. on p. 27).

[8] Justin Richer and Antonio Sanso. *OAuth 2 in Action.* Manning Publications, 2017 (cit. on pp. 22–25).

Software

[9] *Angular.* URL: https://angular.io/ (visited on 02/04/2019) (cit. on pp. 31, 37).

[10] *Angular OAuth OIDC.* URL: https://github.com/manfredsteyer/angular-oauth2-oidc (visited on 05/31/2019) (cit. on p. 46).

[11] *Apollo Angular Client.* URL: https://github.com/apollographql/apollo-angular (visited on 05/31/2019) (cit. on p. 46).

[12] *Apollo GraphQL*. URL: https://www.apollographql.com/ (visited on 05/18/2019) (cit. on p. 43).

[13] *Clarity Design System*. URL: https://clarity.design/ (visited on 04/11/2019) (cit. on p. 31).

[14] *Connect Ensure Login*. URL: https://www.npmjs.com/package/connect-ensure-login (visited on 05/11/2019) (cit. on p. 37).

[15] *GraphQL*. URL: https://graphql.org/ (visited on 05/18/2019) (cit. on p. 43).

[16] *GraphQL Auth Directives*. URL: https://github.com/grand-stack/graphql-auth-directives (visited on 05/18/2019) (cit. on p. 45).

[17] *Neo4j Graph Platform*. URL: https://neo4j.com/ (visited on 06/13/2019) (cit. on p. 42).

[18] *neo4j-graphql-js*. URL: https://github.com/neo4j-graphql/neo4j-graphql-js (visited on 05/18/2019) (cit. on p. 43).

[19] *NestJS - A progressive Node.js web framework*. URL: https://nestjs.com/ (visited on 05/09/2019) (cit. on p. 37).

[20] *Node.js*. URL: https://nodejs.org/ (visited on 05/11/2019) (cit. on p. 37).

[21] *OAuth2rize*. URL: https://github.com/jaredhanson/oauth2orize (visited on 05/11/2019) (cit. on pp. 37, 38).

[22] *OAuth2rize - OpenId Connect Extensions*. URL: https://github.com/jaredhanson/oauth2orize-openid (visited on 06/11/2019) (cit. on p. 40).

[23] *OpenSSL*. URL: https://www.openssl.org/ (visited on 06/13/2019) (cit. on p. 40).

[24] *Passport.js*. URL: http://www.passportjs.org/ (visited on 05/11/2019) (cit. on p. 37).

[25] *RSA-PEM-to-JWK*. URL: https://github.com/OADA/rsa-pem-to-jwk (visited on 06/13/2019) (cit. on p. 42).

[26] *Solid Auth Client*. URL: https://github.com/solid/solid-auth-client (visited on 01/04/2019) (cit. on p. 31).

[27] *Spring Framework*. URL: https://github.com/spring-projects/spring-framework (visited on 04/11/2019) (cit. on p. 37).

[28] *SQLite*. URL: https://sqlite.org (visited on 05/31/2019) (cit. on p. 45).

[29] *TypeORM*. URL: https://typeorm.io (visited on 05/31/2019) (cit. on p. 45).

[30] *TypeScript*. URL: http://www.typescriptlang.org/ (visited on 02/04/2019) (cit. on pp. 31, 37).

[31] Jørn Wildt. *ORDFMapper*. URL: https://github.com/JornWildt/SolidRC/blob/master/wwwroot/js/ORDFMapper.js (visited on 03/07/2019) (cit. on p. 32).

Online sources

[32] George Anadiotis. *Graph databases and RDF: It's a family affair*. May 2017. URL: https://www.zdnet.com/article/graph-databases-and-rdf-its-a-family-affair/ (visited on 06/09/2019) (cit. on p. 19).

[33] Auth0. *Introduction to JSON Web Tokens*. URL: https://jwt.io/introduction/ (visited on 06/13/2019) (cit. on p. 40).

[34] Auth0. *Navigating RS256 and JWKS*. URL: https://auth0.com/blog/navigating-rs256-and-jwks/ (visited on 06/13/2019) (cit. on p. 42).

[35] Sean B. Palmer. *The Semantic Web: An Introduction*. URL: http://infomesh.net/2001/swintro/ (visited on 04/26/2019) (cit. on pp. 18, 19, 21).

[36] Mary Bellis. *The History of How Mark Zuckerberg Invented Facebook*. Jan. 2019. URL: https://www.thoughtco.com/who-invented-facebook-1991791 (visited on 06/09/2019) (cit. on p. 28).

[37] DCMI Usage Board. *DCMI: DCMI Metadata Terms*. June 2012. URL: http://www.dublincore.org/specifications/dublin-core/dcmi-terms/ (visited on 06/09/2019) (cit. on p. 18).

[38] Irina Bolychevsky. *Are Personal Data Stores about to become the NEXT BIG THING?* Oct. 2018. URL: https://medium.com/@shevski/are-personal-data-stores-about-to-become-the-next-big-thing-b767295ed842 (visited on 06/06/2019) (cit. on pp. 6, 7).

[39] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification*. Jan. 2014. URL: http://xmlns.com/foaf/spec/ (visited on 06/09/2019) (cit. on p. 18).

[40] Vannevar Bush. *As We May Think*. July 1945. URL: https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/ (visited on 01/07/2019) (cit. on p. 6).

[41] Vitalik Buterin. *DAOs, DACs, DAs and More: An Incomplete Terminology Guide*. 2014. URL: https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide/ (visited on 06/10/2019) (cit. on p. 53).

[42] Andrew Cave. *What Will We Do When The World's Data Hits 163 Zettabytes In 2025?* URL: https://www.forbes.com/sites/andrewcave/2017/04/13/what-will-we-do-when-the-worlds-data-hits-163-zettabytes-in-2025/ (visited on 06/05/2019) (cit. on p. 1).

[43] Web Data Commons. *RDFa, Microdata, Embedded JSON-LD, and Microformats Data Sets*. Oct. 2016. URL: http://webdatacommons.org/structureddata/2016-10/stats/stats.html (visited on 06/09/2019) (cit. on p. 19).

[44] Cozy. *Cozy Cloud - A Personal Cloud to gather all your data*. 2019. URL: https://cozy.io/en/ (visited on 03/05/2019) (cit. on p. 13).

[45] *Cypher Query Language*. URL: https://neo4j.com/developer/cypher/ (visited on 06/13/2019) (cit. on p. 42).

[46] Digi.me. *Digi.me - Your life, Your terms*. 2019. URL: https://digi.me/ (visited on 03/02/2019) (cit. on p. 14).

[47] EU. *Proposal on copyright in the Digital Single Market.* 2016. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52016PC0593 (visited on 05/13/2019) (cit. on p. 27).

[48] Lee Feigenbaum. *SPARQL By Example: The Cheat Sheet.* Sept. 2008. URL: https://www.iro.umontreal.ca/~lapalme/ift6281/sparql-1_1-cheat-sheet.pdf (visited on 04/17/2019) (cit. on p. 21).

[49] Paula Gearon, Alexandre Passant, and Axel Polleres. *SPARQL 1.1 Update.* Apr. 2013. URL: https://www.w3.org/TR/sparql11-update/#updateLanguage (visited on 04/27/2019) (cit. on p. 21).

[50] Cameron Gray. *Storj Vs. Dropbox: Why Decentralized Storage Is The Future.* 2014. URL: https://bitcoinmagazine.com/articles/storj-vs-dropbox-decentralized-storage-future-1408177107/ (visited on 06/10/2019) (cit. on p. 52).

[51] Dick Hardt. *The OAuth 2.0 Authorization Framework.* URL: https://tools.ietf.org/html/rfc6749 (visited on 04/27/2019) (cit. on pp. 22, 23).

[52] Hub-of-All-Things. *Hub-of-All-Things.* 2019. URL: https://www.hubofallthings.com/ (visited on 03/05/2019) (cit. on pp. 14, 15).

[53] *Icons by Freepik, CCBY 3.0.* URL: www.freepik.com (visited on 04/22/2019) (cit. on pp. 8, 9).

[54] Inrupt. *Inrupt.* 2019. URL: https://inrupt.com/solid (visited on 03/09/2019) (cit. on p. 16).

[55] Inrupt. *Solid.* 2019. URL: https://solid.inrupt.com/ (visited on 03/05/2019) (cit. on p. 15).

[56] Inrupt. *Solid Specification v.0.7.0.* 2019. URL: https://github.com/solid/solid-spec/ (visited on 02/02/2019) (cit. on p. 15).

[57] Leetaru Kalev. *As GDPR Turns One Is It A Success Or A Failure?* URL: https://www.forbes.com/sites/kalevleetaru/2019/05/06/as-gdpr-turns-one-is-it-a-success-or-a-failure/ (visited on 06/05/2019) (cit. on p. 1).

[58] *Key Changes with the General Data Protection Regulation – EUGDPR.* URL: https://eugdpr.org/the-regulation/ (visited on 05/13/2019) (cit. on p. 25).

[59] LifeScope. *LifeScope.* 2019. URL: https://lifescope.io/ (visited on 03/05/2019) (cit. on pp. 14, 15).

[60] Christian Lutz. *Idea Integration into Nest.JS.* URL: https://stackoverflow.com/questions/53544876/how-to-integrate-neo4j-database-nestjs-framework-and-graphql/55318344#55318344 (visited on 05/18/2019) (cit. on p. 43).

[61] Chris Anderson Michael Wolff. *The Web Is Dead. Long Live the Internet.* Aug. 2010. URL: https://www.wired.com/2010/08/ff-webrip/ (visited on 06/09/2019) (cit. on p. 21).

[62] *Neo4j GraphQL JS - Documentation.* URL: https://grandstack.io/docs/neo4j-graphql-js.html#schema-augmentation (visited on 05/18/2019) (cit. on p. 43).

[63] Nikolaj Nielsen. *GDPR - a global 'gold standard'?* URL: https://euobserver.com/justice/141906 (visited on 06/05/2019) (cit. on p. 1).

[64] *OAuth 2.0 Dynamic Client Registration Protocol.* URL: https://tools.ietf.org/html/rfc7591 (visited on 06/15/2019) (cit. on p. 38).

[65] OAuth.io. *OAuth2 Introduction Through Flow Diagrams in 5-minutes.* Sept. 2018. URL: https://blog.oauth.io/introduction-oauth2-flow-diagrams/ (visited on 06/09/2019) (cit. on p. 23).

[66] Okta. *Authorization vs Authentication.* URL: https://www.oauth.com/oauth2-servers/openid-connect/authorization-vs-authentication/ (visited on 06/09/2019) (cit. on p. 25).

[67] Okta. *Differences Between OAuth 1 and 2.* URL: https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/ (visited on 06/09/2019) (cit. on p. 22).

[68] *OpenID Connect.* URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 06/11/2019) (cit. on p. 38).

[69] *OpenID Connect Discovery 1.0.* URL: https://openid.net/specs/openid-connect-discovery-1_0.html (visited on 06/11/2019) (cit. on pp. 38, 42).

[70] I. Polikoff and D. Allemang. *Semantic Technology.* Sept. 2003. URL: https://lists.oasis-open.org/archives/regrep-semantic/200402/pdf00000.pdf (visited on 06/09/2019) (cit. on p. 18).

[71] *Regulation (EU) 2016/679 of the European Parliament and of the Council.* Journal reference: L119, 4 May 2016, p. 1–88. Apr. 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (visited on 05/13/2019) (cit. on p. 25).

[72] *Remarks delivered at the World Economic Forum.* URL: https://www.georgesoros.com/2018/01/25/remarks-delivered-at-the-world-economic-forum/ (visited on 03/31/2019) (cit. on p. 7).

[73] *RSA Key Formats.* URL: https://www.cryptosys.net/pki/rsakeyformats.html (visited on 06/11/2019) (cit. on p. 40).

[74] Verborgh Ruben. *Getting my personal data out of Facebook.* 2019. URL: https://ruben.verborgh.org/facebook/ (visited on 06/05/2019) (cit. on p. 1).

[75] *Schema.org.* URL: https://schema.org/ (visited on 06/23/2019) (cit. on p. 17).

[76] Olivia Tambini. *WhatsApp ads to appear in app from 2020.* URL: https://www.techradar.com/news/whatsapp-ads-to-appear-in-app-from-2020 (visited on 06/10/2019) (cit. on p. 52).

[77] *The world's most valuable resource is no longer oil, but data.* May 2017. URL: https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data (visited on 06/06/2019) (cit. on p. 7).

[78] Ruben Verborgh. *An introduction to the Solid ecosystem.* 2018. URL: https://rubenverborgh.github.io/Web-Foundation-2018/ (visited on 02/02/2019) (cit. on p. 15).

[79] W3C. *Linked Data Platform 1.0.* Feb. 2015. URL: https://www.w3.org/TR/2015/REC-ldp-20150226/ (visited on 06/09/2019) (cit. on p. 28).

[80] W3C. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition).* Dec. 2012. URL: https://www.w3.org/TR/owl2-syntax/ (visited on 06/09/2019) (cit. on p. 29).

[81] W3C. *RDFa Core 1.1 - Third Edition.* Apr. 2015. URL: https://www.w3.org/TR/2015/REC-rdfa-core-20150317/ (visited on 06/09/2019) (cit. on pp. 18, 19).

[82] Josephine Wolff. *How Is the EU's Data Privacy Regulation Doing So Far?* Mar. 2019. URL: https://slate.com/technology/2019/03/gdpr-one-year-anniversary-breach-notification-fines.html (visited on 06/05/2019) (cit. on p. 1).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —

width = 100mm
height = 50mm

— Diese Seite nach dem Druck entfernen! —