Politecnico di Torino

Politecnico di Torino

Master of Science (M.Sc.) in Computer Engineering

A.y. 2025/2026

Graduation Session March 2026

Privacy Dashboard

Design and Implementation of a Web Application for GDPR Management integrated with a Smart Home system

Supervisor
Prof. Luca Ardito

Candidate
Giorgio Silvestre

Summary

The growing trend of adopting IoT devices, such as sensors, cameras, and voice assistants, makes the Smart Home a dense environment for personal data. The fact that these devices are installed directly inside our homes gives them a privileged position that enables them to collect a significant amount of sensitive data, including the devices we use, our behavior, daily habits, and personal preferences, posing a challenge to the management of user privacy and security.

In this scenario, the aim of the project, carried out in collaboration with two other colleagues, was to refactor and add new functionalities to an existing web application, initially developed in the context of the European project "SIFIS-Home", dedicated to the management of obligations and rights conferred by the GDPR. The new Privacy Dashboard was developed with the primary objective of being integrated with "Smartotum", a scalable and privacy-focused Home Automation System, providing a centralized platform to manage consents, GDPR-related documents, and assist both end users and Data Controllers/Data Protection Officers. In particular, we focused on the possibility of reconfiguring devices both automatically and manually, depending on the preferences expressed by end users. A crucial use case involves home cameras: when a user withdraws consent, the video feed must be disabled automatically and re-enabled only once consent is granted again.

This thesis begins with an overview of the existing software and an assessment of its limitations, from which our requirements were gathered and defined. A more in depth analysis of the new features is then presented, followed by an explanation of the user interface, which is personalized based on the role of each user. Afterwards, my personal contribution to this project is addressed, by first providing a background on the chosen software architectures and technologies, and then by assessing how they were adopted to take part in the implementation of our web application. Finally, a retrospective of our work is conducted, in order to examine the constraints of our software and to discuss how it could be further improved in the future.

Table of Contents

List of Figures

List of Listings

Glossary

ACID

Atomicity, Consistency, Isolation, Durability

API

Application Programming Interface

CD

Continuous Delivery

CI

Continuous Integration

CRUD

Create, Read, Update, Delete

CSRF

Cross-Site Request Forgery

DBMS

Database Management System

DHT

Distributed Hash Table

DOM

Document Object Model

ENISA

European Union Agency for Cybersecurity

GDPR

General Data Protection Regulation

HTML

HyperText Markup Language

HTTP

Hypertext Transfer Protocol

IoT

Internet of Things

JSON

JavaScript Object Notation

JWT

JSON Web Token

MPA

Multi-Page Application

MVCC

Multi-Version Concurrency Control

NSSD

Not-So-Smart Devices

ORM

Object–Relational Mapping

PDF

Portable Document Format

PIA

Privacy Impact Assessment

PTP

Policy Translation Point

RBAC

Role-Based Access Control

REST

Representational State Transfer

SIFIS-Home

Secure Interoperable Full-Stack Internet of Things for Smart Home

SPA

Single Page Application

SQL

Structured Query Language

URL

Uniform Resource Locator

UUID

Universally Unique Identifier

WSGI

Web Server Gateway Interface

WYSIWYG

What You See Is What You Get

XACML

eXtensible Access Control Markup Language

XSS

Cross-Site Scripting

Chapter 1

Introduction

1.1 Context

The recent growth in the adoption of IoT (Internet of Things) devices is playing a key role in the modernization and digital transformation of institutions, businesses and industries, and is becoming more significant in private domestic environments as well. The increasing amount of *smart* devices present in households has lead to the concept of *Smart Home*, where a network of interconnected devices is deployed with the main goal of improving the resident's quality of life, with features that range from the automation of simple tasks to an improvement of security systems. However, in order to reliably provide its functionalities, the software operating within a Smart Home system needs to continuously gather data from its environment. The fact that a growing amount of data is then gathered directly from within households raises increasing concerns related to the management of end user privacy and security.

To address this issues, the European project *SIFIS-Home* was developed with the aim of providing a secure-by-design and consistent software framework for improving resilience of Smart Home systems [1]. In the context of SIFIS-Home, the web application *Privacy Dashboard* was initially developed with the main purpose of centralizing and improving privacy management and GDPR compliance for applications that operate in Smart Home systems [2].

This thesis work consisted of taking part in a student group project whose main goal was to rewrite the existing Java application by having a clear separation of the backend in Python from the frontend in TypeScript, and by adding several new functionalities and improving the user interface and experience. The core new feature of our Privacy Dashboard is the integration of *Smartotum*, a home automation system that enables the interaction of our software with a real Smart Home environment.

1.2 Project Goal

As mentioned previously, our group project work was based on an existing web application built entirely in Java, which we analyzed to understand its functionalities and address its limitations. In addition to the existing Privacy Dashboard, we took into account PTP (Policy Translation Point), a Java software that translates high-level security policies into low-level policies in a XACML formalism [3]. Both these applications were developed to work in a standalone manner without an interaction with a home automation system, thus our main objective was to redesign and rewrite them in a single platform that could join all of their functionalities, while also implementing the missing integration with the Smartotum domotic system.

Based on this preliminary analysis, the requirements for our application were collectively defined and revised, to then move on to the choices of our system architecture and technologies. One of the goals of our project was to design a software taking into account important software engineering principles, such as separation of concerns, modularity, scalability and security. For this reason, we opted for well known and widely adopted web frameworks: the frontend was developed in TypeScript using React, while the backend was implemented in Python using Flask to develop a scalable RESTful API that persists its data in a PostgreSQL database. Besides these common goals, we split our work into individual contributions to the project:

- One colleague handled the integration with Smartotum, the frontend design and shared state architecture, the Marketplace proof of concept and the CI (Continuous Integration)/CD (Continuous Delivery) workflow.
- Another colleague focused on the PTP, including ontology based translation, conflict detection, XACML generation and interaction with the DHT for high level privacy rules.
- My personal contribution, which is the main subject of this thesis, focused on the core backend infrastructure, on end-to-end testing, and on the frontend components dedicated to GDPR document management, such as Privacy Notices and Privacy Impact Assessments.

Real time collaboration on the project implementation was made possible by continuously pushing our code to a remote GitLab repository [4].

1.3 Thesis Goal

Although much of the work in this project was performed collaboratively, the main goal of this thesis is to illustrate my personal contribution to the project, which can be summarized with the following objectives:

- **Backend Implementation**: Having analyzed the backend of the legacy Privacy Dashboard, my first goal was to design and implement the core backend infrastructure in Python using the Flask framework, including its integration with a PostgreSQL database achieved through an ORM (Object–Relational Mapping) implemented with SQLAlchemy. After having modeled the entities and relationships we persist in our database, the remaining backend infrastructure was designed, by adopting the Controller–Service–Repository pattern. In particular, the controller layer was implemented with the flask-smorest library to streamline the development of our endpoints and with the marshmallow library to add request and response validation and serialization. In addition, a token-based authentication system was set up to ensure RBAC (Role-Based Access Control) to all of our endpoints.

- **End-to-end Testing**: Parallel to the development of our endpoints, a process of testing was necessary to verify their functionality. For this reason, a Postman collection was created and progressively updated whenever I finished the implementation of an endpoint, in order to add the corresponding end-to-end testing. This type of testing ensured that all of the layers of our backend were functioning and interacting with each other as expected.

- **Frontend Implementation**: My main contribution to the frontend was to design and implement the user interfaces dedicated to the creation and management of documents related to GDPR compliance, namely the Privacy Notice, the PIA (Privacy Impact Assessment) and the Questionnaire. The main goal was to implement separate pages for each of them that should, however, feel coherent among each other as to improve the ease of use and overall appearance of our client. Moreover, new functionalities were added with respect to the legacy application, from the PIA page itself to the new text editor for Privacy Notices, to the feature of downloading these three types of documents as PDF files. Finally, the previous pages dedicated to messages and contacts were redesigned into a single page that mimics modern messaging platforms, so that users now face a more familiar and intuitive interface.

1.4 Thesis Structure

This thesis is structured to first provide the reader with a general context and background in which our project evolved. Then, it showcases all of the features we developed, each with the corresponding user interface. It then shifts its focus on my personal contributions, by first providing a theoretical background to discuss the choices of our technological stack, and then by addressing more in detail how I implemented it to achieve the personal goals defined in the previous section. More specifically, the thesis chapters are organized as follows:

- **Chapter 2.** *Background and Legacy System Analysis* The context in which we developed our project is outlined in this chapter, where a general background is provided to introduce the concepts of domotic systems, IoT (Internet of Things) and GDPR (General Data Protection Regulation), in order to highlight the necessity of compliance to GDPR for Smart Home systems. This chapter then provides an overview of the existing web application, in order to analyze it and to discuss its limitations, upon which our project work was planned.
- **Chapter 3.** *The New Privacy Dashboard* This chapter begins with a detailed description of the roles that users can have in our platform, so that the authorizations granted to each role are clearly outlined. Next, this chapter illustrates all the features of our software, where, for each functionality, screenshots are attached to display the corresponding interface in the frontend. An end-to-end usage scenario follows to give an example of how these different features may be utilized together in a single flow of usage of the application. Finally, a table containing all of our functional requirements is provided.
- **Chapter 4.** *System Architecture and Technologies* This chapter does not cover any practical implementation aspects of our work, but rather provides a detailed theoretical background on the technologies we faced, so as to motivate, for each of them, the choice of a corresponding technological framework rather than another.
- **Chapter 5.** *Implementation* In this chapter, my personal contribution to the project work is analyzed in detail by covering, in particular, what framework or external library I used to implement each of the personal goals defined in the previous section.
- **Chapter 6.** *Conclusion* This chapter draws the conclusion for both the overall project work and the personal contributions, by highlighting both the results we managed to achieve with respect to the initial goals and the limitations our web application still faces, in order to provide a clear foundation for any future development.

Chapter 2

Background and Legacy System Analysis

2.1 Background

2.1.1 Smart Home and IoT

The term *IoT (Internet of Things)* is used to denote a network of physical devices, such as cameras, sensors, alarms and many more, that are characterized by the ability of exchanging information and commands with each other and with external services and applications connected to the Internet. IoT brings an array of advantages such as allowing data-driven decision making, since actions and decisions can be made taking into account the data that devices continuously gather and share, and increasing automation, which can be enabled, for example, by actuators performing autonomous physical actions after some particular changes are detected in the environment by sensors communicating to them over a wireless network.

The adoption of such class of smart devices is playing a key role in the modernization and digital transformation of institutions, businesses and industries, and is becoming more significant in private households too: indeed, it is estimated that the number of IoT devices all over the world will almost be tripled by 2030, increasing from 8.74 billion in 2020 to more than 25.4 billion in 2030 [5]. Among said sectors, we focus on the usage of IoT within domestic environments, which leads to the concept of *Smart Home*.

A Smart Home is defined as a household in which an interconnected network of smart devices is deployed, enabling bidirectional communication both among the devices themselves and between the devices and the end user via the Internet. Communication among devices can happen over a local wireless network and allow their embedded software to perform tasks autonomously, whereas communication

between devices and users can happen over the cloud and allow home owners to easily monitor and rule their Smart Home from a centralized software accessible from a device connected to the Internet, such as a smartphone or a Personal Computer. Such kind of system can be implemented using a three-layer architecture: the perception layer, the network layer, and the application layer [6]. The perception layer is the bottom layer where different home appliances collect data from the surrounding environment. Collected data are sent to the network layer through a Smart Home gateway. The network layer, in turn, sends the data in the cloud to the application layer, which is the only one the user directly interacts with, using the dedicated software mentioned above. In many real-world implementations, the application layer relies on cloud services as a central point of aggregation and accessibility of all the information regarding the Smart Home. A graphical representation of the three-layer architecture is illustrated in Figure 2.1.

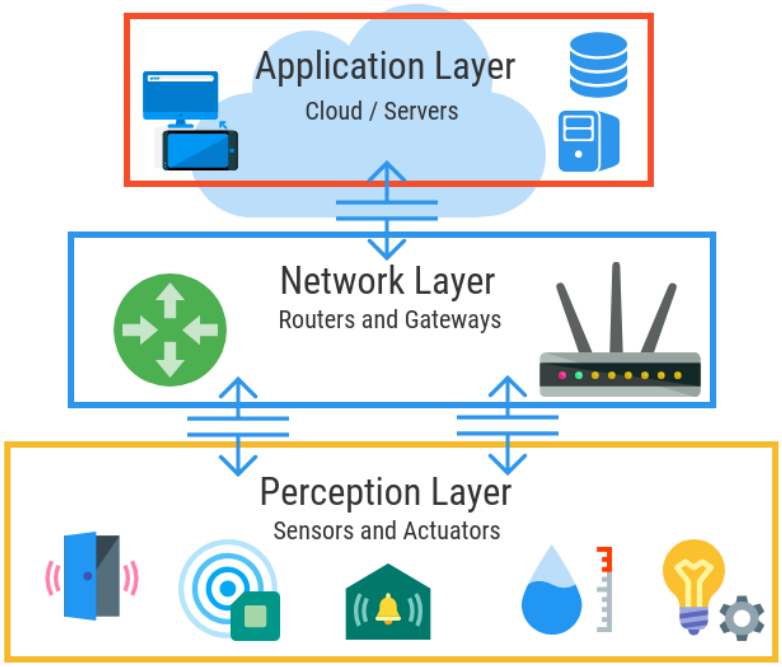


Figure 2.1: The three layer model of IoT architecture.

This type of architecture enables Smart Homes to make autonomous decisions without owner manual intervention and provide assistive and personalized services, which ultimately aims to the main goal of improving the owner's quality of life [7]. For instance, when the homeowner leaves, the Smart Home can automatically turn off lights and enable security alerts if unusual activity is detected.

2.1.2 Privacy Threats in Smart Homes

The increase in amount and complexity of IoT devices brings many advantages compared to traditional devices, as seen in 2.1.1, but unfortunately it encompasses various drawbacks as well, among which the concern of user's privacy and safety stands out: data threats, third party sharing and user unawareness are different ways the Smart Home owners' privacy may be put at risk.

Data Threats The fact that these devices are a continuous source of personal data collected inside our homes makes them a valuable target for attackers. A Smart Home system might be victim of different types of threats against data, which the ENISA (European Union Agency for Cybersecurity) has classified and defined as follows [8]:

- **Data breach** is an intentional cyberattack executed by a cybercriminal with the goal of gaining unauthorised access to release sensitive confidential or protected data.
- **Data leak** is an event (*e.g.* due to misconfigurations, vulnerabilities or human errors) that can cause the unintentional loss or exposure of sensitive, confidential or protected data.
- **Data manipulation** is a category of attacks that aims to manipulate trustworthy data into untrustworthy, bugged data, targeting the perception of reality by people.

As we will analyze in 5.1.1, our software internally stores data related to each user who has an account on the platform, so it is important to implement measures to prevent and mitigate the described data threats. These measures include having a reliable authentication and authorization system for a strict a RBAC (Role-Based Access Control), always storing and verifying hashed passwords instead of plain-text, implementing input validation on each request, adding database constraints and rollback features, and never exposing internal database IDs in responses.

Third party sharing The connectivity that smart devices have to the Internet allows them to potentially share information about their owners to third parties, *i.e.* companies that are not directly related to the manufacture or maintenance of said devices or the network they are using, which utilize it to improve user profiling and personal advertising. A research conducted in 2019 performed tests on 81 commercial devices and found that 72 of them have at least one destination that is not a first party and that more than half contact destinations outside their region, thus highlighting another critical risk for user's privacy [9].

User unawareness As Smart Home systems become more complex, end users become more likely to underestimate the potential threats and risks related to their privacy. A survey carried out in 2017 by researchers from the University of Washington found that many participants acknowledged that privacy could be an asset, particularly in the form of audio or behavior logs. However, half of these participants were not particularly concerned about privacy risks, and expressed different reasons for their lack of concern, ranging from explicit trust in companies handling user data to not considering themselves a worthwhile target. Some participants also believed that they had taken sufficient steps to secure their systems, such as with strong passwords, so they did not need to worry further about security [10].

2.1.3 Smartotum and the SIFIS-Home Framework

SIFIS-Home The *SIFIS-Home* Project was created to address these concerns by developing a secure-by-design and consistent software framework across all stack levels [1]. From a design standpoint the SIFIS-Home Framework is organized into dedicated frameworks (*i.e.* Smart Device, NSSD (Not-So-Smart Devices), Application, Cloud, Development Tools), each comprising the set of software components executed on a specific platform [11].

The overall architecture follows a microservices approach based on Docker containers: most of the components can expose a RESTful API when needed for integration. Instead of relying on the cloud as the single aggregation point of Smart Home information, SIFIS-Home relies on a distributed data plane deployed inside the home network, enabling components to exchange information without a central broker. This layer is implemented through a DHT (Distributed Hash Table) that supports both volatile messages (delivered to running applications) and persistent messages stored locally to preserve critical data such as settings and policies across reboots. When remote access is required, SIFIS-Home includes a bridging component that can exchange messages and configurations between the DHT and a cloud-side platform. This design makes it possible to keep the Smart Home state and policies primarily within the domestic environment, while still enabling remote interaction when needed [11].

Smartotum Smartotum originated as a technology transfer initiative that brings the SIFIS-Home vision into a deployable home automation system [12]. Consistently with the SIFIS-Home approach, Smartotum adopts a local-first architecture where the core intelligence resides inside the household and does not rely on cloud services. It also follows a distributed approach based on multiple controllers operating in the home and connected via Wi-Fi or Bluetooth to manage domestic services (heating, lighting, alarm systems and more), with a focus on ease of installation.[12]

While SIFIS-Home and Smartotum provide a secure-by-design technical foundation, they do not by themselves define how legal obligations and user rights should be enforced in practice. For this reason it has become evident that a comprehensive regulatory framework capable of imposing obligations on organizations that collect and process personal data is necessary to guarantee the protection of user privacy.

2.1.4 General Data Protection Regulation

In April 2016, the European Union adopted the GDPR (General Data Protection Regulation), which establishes a legal framework for the protection of personal data within the EU and imposes strict obligations on entities that collect, process, or store such data. Regarding the sector of autonomous domotic systems we are discussing, it also provides a fundamental legal framework for assessing privacy risks in IoT environments. It is then important to start addressing the GDPR by considering the definition of *personal data* it provides in Art. 4 [13]:

> Any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person;

This definition introduced the concept of *Data Subject*, who from now on will be considered as the person whose data is being collected and processed. In contrast to the Data Subject, we now give the definitions of the people responsible for the Data Subject's personal information, *i.e.* the *Data Controller* and the *Data Processor*, which are also reported in Art. 4 [13]:

> 'controller' means the natural or legal person, public authority, agency or other body which, alone or jointly with others, determines the purposes and means of the processing of personal data; [...] 'processor' means a natural or legal person, public authority, agency or other body which processes personal data on behalf of the controller;

The last figure we are going to analyze and consider in our project is the *Data Protection Officer*: there is no explicit definition as for the other ones in Art. 4, but it is regarded as a designated expert within an organization responsible for overseeing and advising on compliance with the GDPR. The cases in which a Data Protection Officer shall be designated by a Data Controller are described in Art. 37 [14], whereas its core functions are described in Art. 39 [15] and include:

- Informing and advising the organization on data protection obligations.

- Monitoring GDPR compliance.
- Providing guidance on data protection impact assessments.
- Acting as a point of contact for supervisory authorities on issues relating to processing.

The GDPR provides an extensive and detailed set of rules each Data Controller should adhere to when creating services and products, but we can find an overview of them in Art. 5 [16], which lists six principles relating to processing of personal data.

1. **Lawfulness, fairness and transparency**: Personal data shall be processed lawfully, fairly and in a transparent manner in relation to the data subject.
2. **Purpose limitation**: Personal data shall be collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes. [...]
3. **Data minimisation**: Personal data shall be adequate, relevant and limited to what is necessary in relation to the purposes for which they are processed.
4. **Accuracy**: Personal data shall be accurate and, where necessary, kept up to date. [...]
5. **Storage limitation**: Personal data shall be kept in a form which permits identification of data subjects for no longer than is necessary for the purposes for which the personal data are processed. [...]
6. **Integrity and confidentiality**: Personal data shall be processed in a manner that ensures appropriate security of the personal data, including protection against unauthorised or unlawful processing and against accidental loss, destruction or damage, using appropriate technical or organisational measures.

Besides the principles we just outlined, the GDPR requires that each Data Subject mentioned in Art. 4 [13] is granted the following rights when using a product or a service that is collecting their data:

1. **Right of access** (Art. 15 [17]): The data subject shall have the right to obtain from the controller confirmation as to whether or not personal data concerning him or her are being processed, and [...] the controller shall provide a copy of the personal data undergoing processing. [...]
2. **Right to rectification** (Art. 16 [18]): The data subject shall have the right to obtain from the controller without undue delay the rectification of inaccurate personal data concerning him or her. [...]

3. **Right to erasure** (Art. 17 [19]): The data subject shall have the right to obtain from the controller the erasure of personal data concerning him or her without undue delay [...]

4. **Right to restriction** (Art. 18 [20]): The data subject shall have the right to obtain from the controller restriction of processing [...]

5. **Right to portability** (Art. 20 [21]): The data subject shall have the right to receive the personal data concerning him or her, which he or she has provided to a controller, in a structured, commonly used and machine-readable format [...]

6. **Right to object** (Art. 21 [22]): The data subject shall have the right to object, on grounds relating to his or her particular situation, at any time to processing of personal data concerning him or her [...]

In order to ensure the respect of these pivotal rights, the GDPR mandates in Art. 25 [23] that Data Controllers take all the possible measures to guarantee that their products and services follow the principles of *privacy by design* and *privacy by default*. These principles require that data protection measures are embedded into the architecture of systems from the earliest stages of development, rather than being treated as an afterthought. In particular, services must be configured so that, by default, only the personal data strictly necessary for each specific purpose are processed, thus minimizing exposure and reducing privacy risks.

2.1.5 GDPR Documents

As discussed above, the GDPR requires both the protection of Data Subject rights and the adoption of principles such as transparency and privacy by design. In our software, these requirements are reflected in practice by the implementation of two important documents: the *Privacy Notice* and the *Privacy Impact Assessment*.

Privacy Notice The Privacy Notice is used by Data Controllers to comply with the transparency obligations established in Art. 12, 13 and 14. These provisions implement the principle of transparency by requiring Data Controllers to provide clear and accessible information to Data Subjects regarding the processing of their personal data [24]. A Privacy Notice is therefore a document that must be accessible to Data Subjects and that allows them to understand who is processing their data, the purposes of the processing, the legal basis relied upon, the retention period, possible recipients of the data, and the rights available to them granted by the GDPR.

Privacy Impact Assessment The other important document, addressed by the GDPR in Art. 35 [25], is the PIA (Privacy Impact Assessment). It is stated that this document shall be carried out when processing, in particular using new technologies, is likely to result in a high risk to the rights and freedoms of Data Subjects, taking into account the context and purpose of such processing. Art. 35 requires a PIA to be carried out, in particular, in case of:

1. a systematic evaluation of data related to Data Subjects which is based on automated processing, including profiling, and on which decisions are based that produce significant effects on such Data Subjects.
2. processing on a large scale of special categories of data, or of personal data relating to criminal convictions and offenses.
3. a systematic monitoring of a publicly accessible area on a large scales.

This document is supposed to be carried out by a Data Controller, but he or she should seek the advice of a Data Protection Officer, if designated. Moreover, Art. 35 requires that the PIA contains, at least:

1. A systematic description of the processing and its purposes.
2. An assessment of the necessity and proportionality of the processing in relation to the purposes.
3. An assessment of risks to individuals' rights and freedoms.
4. Measures to mitigate risks, including safeguards and security controls.

2.2 Legacy System Analysis

2.2.1 Overview of the Existing Privacy Dashboard

The software we designed and developed during this project was based on the requirements and functional goals of a previous Privacy Dashboard prototype, therefore the first step was to analyze its functionalities and address its limitations.

Privacy Dashboard was initially developed by Filippo Peron as the project for his Master's Degree thesis in Computer Engineering in Politecnico di Torino during *a.y.* 2022/2023, and he reported the outcome of his work in his thesis paper [26]. He carried out his project using Vaadin, an open source framework for the development of web applications in Java, which in turn adopts Spring Security for what concerns authentication and authorization of users. Regarding authorization, both the features and the user interfaces of the existing Privacy Dashboard are differentiated according to the role of the logged in user (*i.e.* one of Data Subjects, Data Controllers, and Data Protection Officers), as shown in the following pages.

- **Applications page**: The core of the Privacy Dashboard, as it contains a list of installed or managed applications, each with different details and functionalities, depending on the user's role. Data Subjects can see the description of each installed application, together with privacy related information, such as the evaluation received in a questionnaire, a link to the associated Privacy Notice and the list of provided consents, with the possibility of withdrawing each one. On the other hand, Data Controllers and Data Protection Officers do not have the list of provided consents, as that is a feature reserved for Data Subjects, but have the exclusive possibility of completing the questionnaire and Privacy Notice for each application. These different role-based interfaces of the *Applications* page are shown respectively in Figure 2.2 and Figure 2.3.

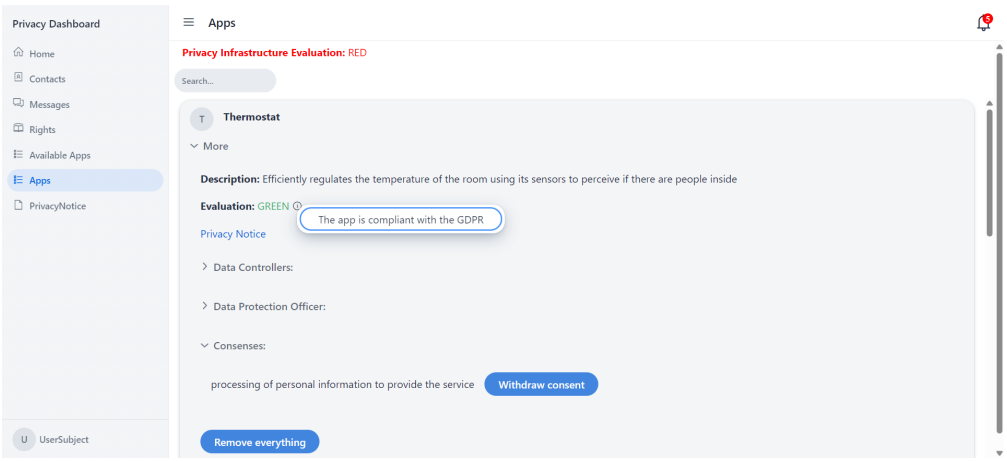


Figure 2.2: Applications Page for Data Subjects.

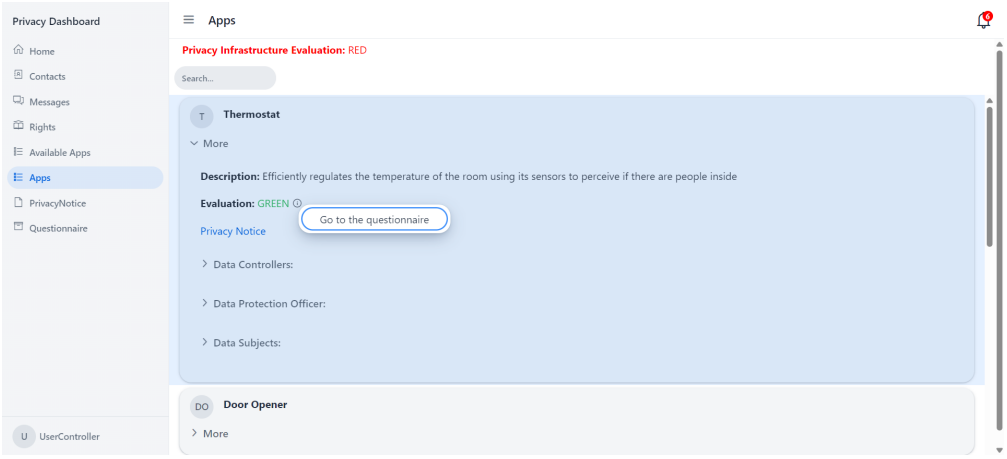


Figure 2.3: Applications Page for Data Controllers and Data Protection Officers.

- **Contacts page**: This page is shared across all roles. Each user can see the list of their contacts and, by selecting one, start a direct message exchange with that contact. For each contact, the user can also see which applications they have in common. A formal definition of *contact* in this context will be provided in Chapter 3.
- **Rights page**: This page allows Data Subjects to easily exercise their GDPR rights, as described in section 2.1.4. Once a type of right request is selected, a form is opened to allow the Data Subject to choose which application they are referring to and add some additional information before submitting it, as in Figure 2.4. The Data Controllers and Data Protection Officers can instead see a list of pending requests and, for each of them, they can mark it as handled and add an additional response for the Data Subject who submitted it, as in Figure 2.5.

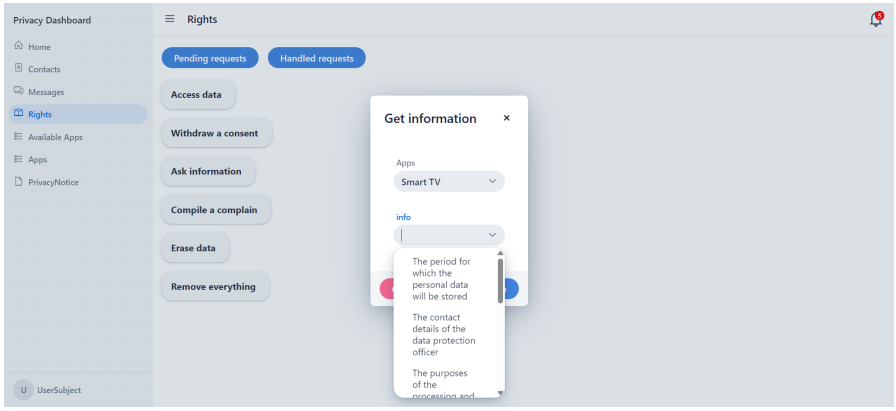


Figure 2.4: Rights Page for Data Subjects.

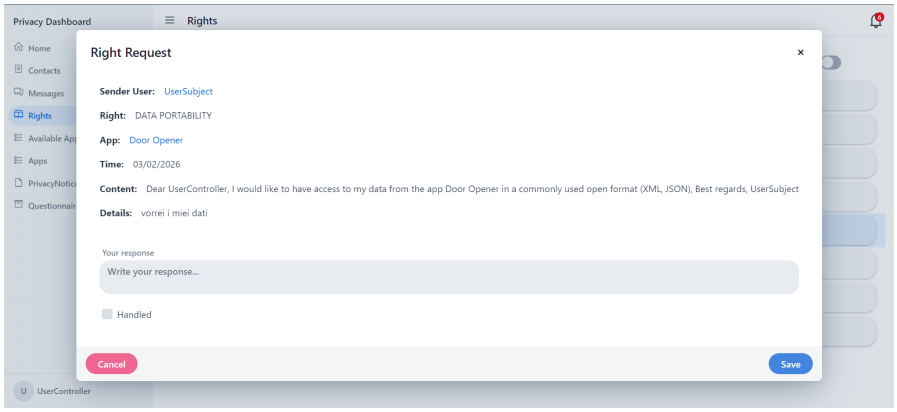


Figure 2.5: Rights Page for Data Controllers and Data Protection Officers.

- **Privacy Notice page**: In this page Data Controllers and Data Protection Officers have the possibility of writing or editing a Privacy Notice for each application, and they can choose whether to do so from scratch or from a provided template, as in Figure 2.6, whereas Data Subjects can only read them.
- **Questionnaire page**: In this page Data Controllers and Data Protection Officers have the possibility of carrying out or editing a Questionnaire for each application, and they can do so from a provided template, as in Figure 2.7, whereas Data Subjects do not have access to this page. More details on Privacy Notices and Questionnaires and how they are related to compliance with GDPR will be provided in Chapter 3.

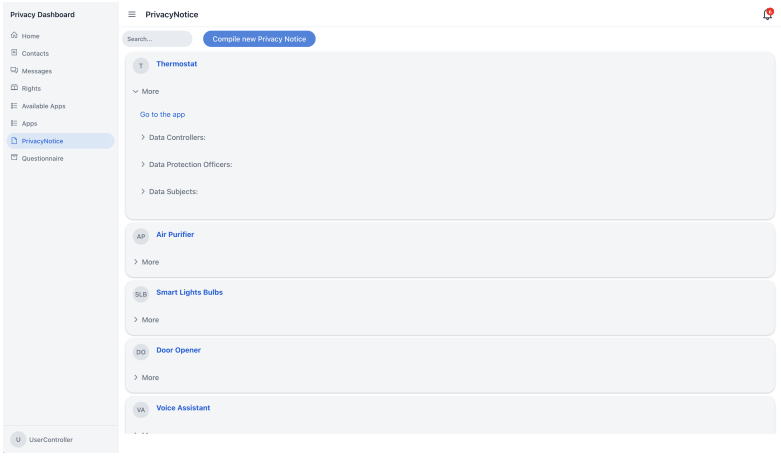


Figure 2.6: Privacy Notice Page for Data Controllers and Data Protection Officers.

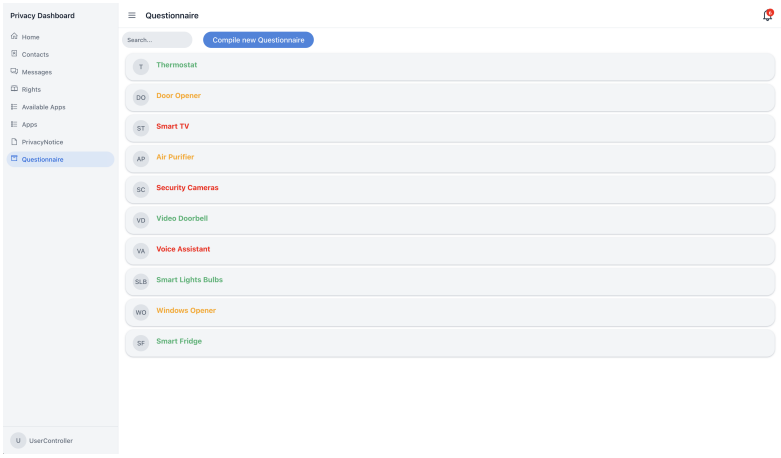


Figure 2.7: Questionnaire Page for Data Controllers and Data Protection Officers.

2.2.2 Limitations of the Existing Privacy Dashboard

One of the main limitations that the existing Privacy Dashboard has is the absence of interaction with a real domotic system, which means that, for a Data Subject, the association between an installed application and the owned Smart Home into which such application is installed is missing. Indeed, the existing backend did not store an entity representing the Smart Home, and thus there was not a way to represent a persistent relationship between a user, an application, and a Smart Home. As a result, the dashboard could represent user-application associations only at an abstract level, without linking them to a Smart Home instance or installation context. The web application was designed with this feature taken into account, but it had not been possible to implement and test it. Similarly, the feature of downloading files with the content of Privacy Notices and stored personal data was planned but not implemented. Finally, the creation and management of another important document related to GDPR compliance, the PIA (Privacy Impact Assessment), is missing.

These limitations were acknowledged by Filippo Peron in the conclusion of his thesis paper [26] and laid the foundations for the definition of our functional requirements, which will be presented in the following chapter.

Chapter 3

The New Privacy Dashboard

Building on top of the original Privacy Dashboard, we not only preserved its existing features, but also introduced new functionalities to address its limitations and to turn it into an all-in-one platform for managing privacy in smart home systems.

3.1 User Roles

Before analyzing the main features of the new Privacy Dashboard, it is important to introduce the key user roles on which the platform is based. As in the legacy Privacy Dashboard, the new system supports the main GDPR actors discussed in Section 2.1.4: Data Subjects, Data Controllers and Data Protection Officers. The Privacy Dashboard provides a convenient way both for Data Subjects to manage their privacy within the smart home and for Data Controllers and Data Protection Officers to manage the GDPR aspects of the applications under their responsibility, such as available consents, GDPR documents and right requests. These roles interact through a single platform, for this reason, they share some common functionalities, while others remain accessible only to specific ones.

In the new Privacy Dashboard, applications can originate from different sources, such as applications installed in a user's Smart Home. For this reason, it is important to make a distinction between the primary Data Controller of an application and the other Data Controllers delegated to manage it. The primary Data Controller, referred to as the owner of an application, can manage the assignment of other Data Controllers and the appointment of a DPO, whereas delegated Data Controllers cannot. The functionalities available to the different roles are summarized in Table 3.2.

Functionality	Data Subject	DC App Owner	DC App Manager	DPO
Homes	View	No	No	No
Smartotum Sync	Yes	No	No	No
Installed Apps	View	No	No	No
Managed Apps	No	View	View	View
Local Apps	No	View/Edit	View	View
Assignments	View	View/Edit	View	View
Privacy Notice	View	View/Edit	View/Edit	View/Edit
Questionnaire	View Evaluation	View/Edit	View/Edit	View/Edit
PIA	No	View/Edit	View/Edit	View/Edit
Consents	Give/Withdraw	Manage Catalog	Manage Catalog	Manage Catalog
Right Requests	Submit/Track	View/Handle	View/Handle	View/Handle
Notification History	View	View	View	View
Policy Translation	View/Edit	No	No	No
Profile Update	No	Password	Password	Email / Password
Delete Profile	Yes	Yes	Yes	Yes
Log Out	Yes	Yes	Yes	Yes

Table 3.2: Main frontend functionalities available to each role.

3.2 Main Features and User Interface

3.2.1 Smartotum Integration

The first focus was the integration of the Smartotum Home Automation System, described in 2.1.3, which enables the Privacy Dashboard to retrieve and synchronize information about users' Smart Homes.

Homes Each Data Subject may have multiple homes, so the platform must retrieve basic information such as name, address, and country to allow users to select the correct Smart Home to manage. An easy way for the user to select a specific Smart Home is therefore necessary as shown in Figure 3.1.

Rooms, Devices and Rules When a Smart Home is selected, more specific information is retrieved, such as rooms, devices, and rules, which will play a central role in the policy translation feature presented later in Section 3.2.3.

Installed Applications More information retrieved is about the Smart Home's installed applications, including each application's title and description, the assigned Data Controllers and associated consents. Instead, information about Data Protection Officers, Privacy Notices and Questionnaires is managed directly by the platform and has no corresponding entities in the Smartotum system.

A comprehensive view of the installed applications of the selected Data Subject's Smart Home can be seen in the Installed Apps page, represented in Figure 3.2.

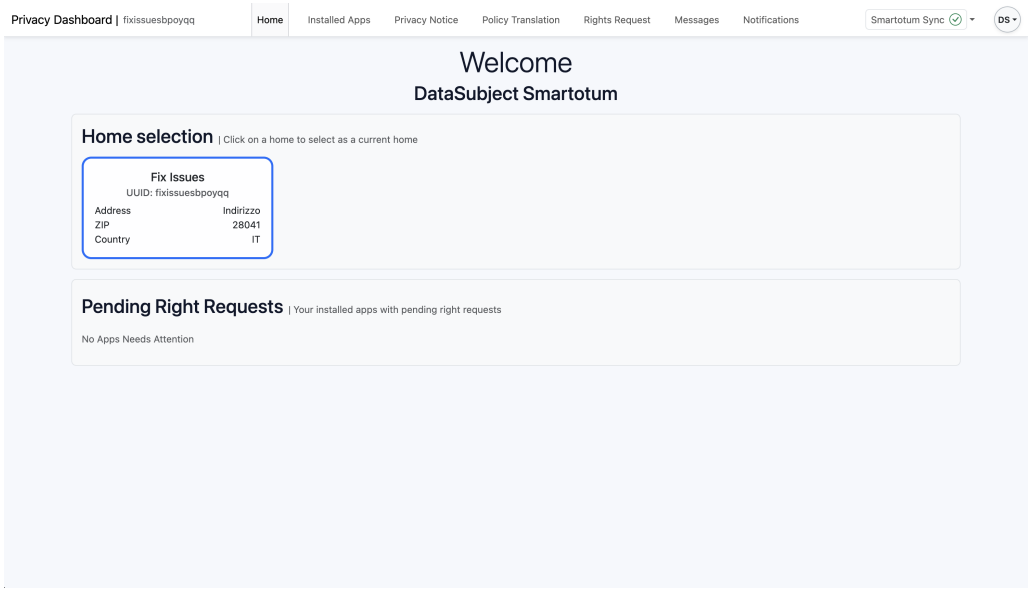


Figure 3.1: Home page for Data Subjects.

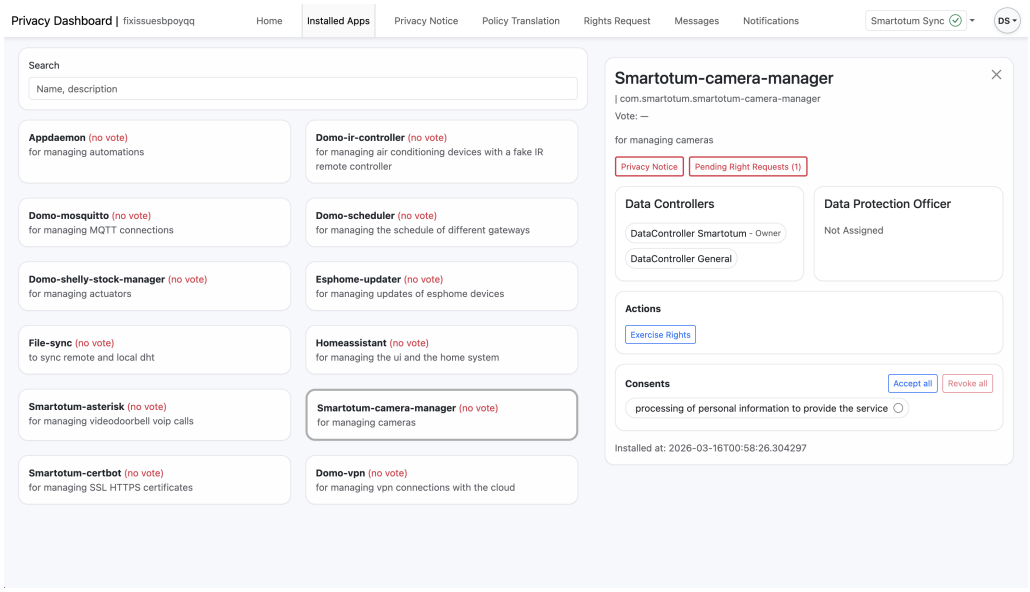


Figure 3.2: Installed Applications page for Data Subjects.

3.2.2 Automatic Enforcement of Consent Changes

Within the Application details panel shown in Figure 3.2, the dedicated Consent Management section allows the Data Subject not only to inspect the consents associated with the application, but also to grant or withdraw them individually or all at once.

From the user's perspective, changing the consent status is not limited to updating the information shown in the dashboard, but also affects the underlying Smart Home behavior for applications whose consents are associated with Smartotum actions. When an application is first installed, all its available consents are marked as not granted by default. For example, for a camera manager application, this could mean that video recording is initially not allowed and the corresponding deny rule is present in the Policy Translation page, as shown in Figure 3.4. When the consent is granted, the deny rule disappears, thus allowing video recording. If the user later decides to withdraw it again through the interface shown in Figure 3.3, the rule reappears in the Policy Translation page.

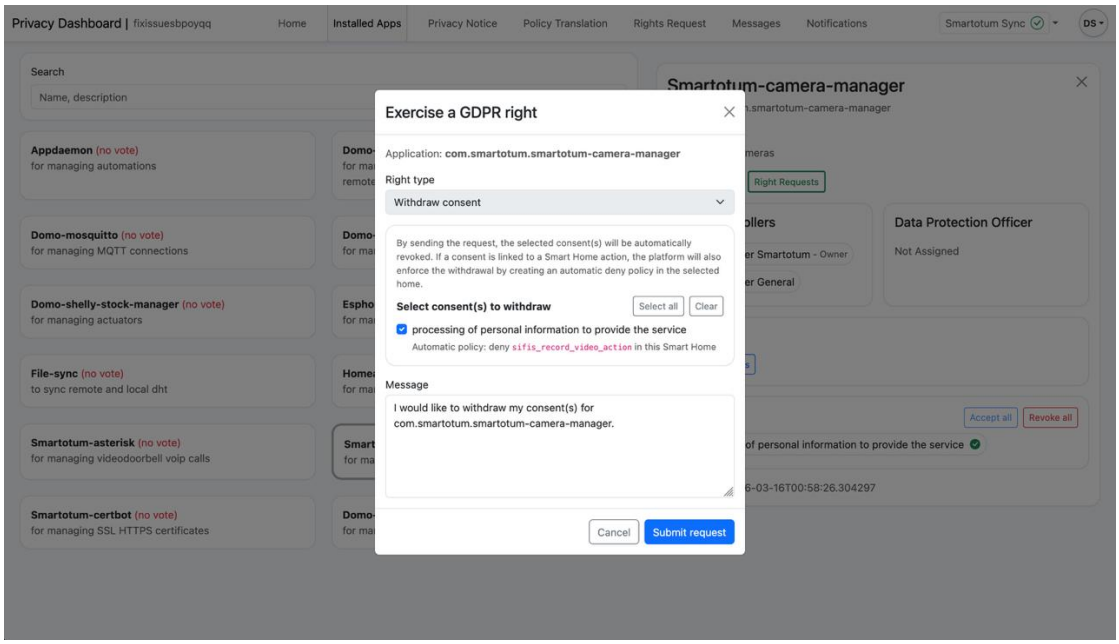


Figure 3.3: Example of a withdraw consent.

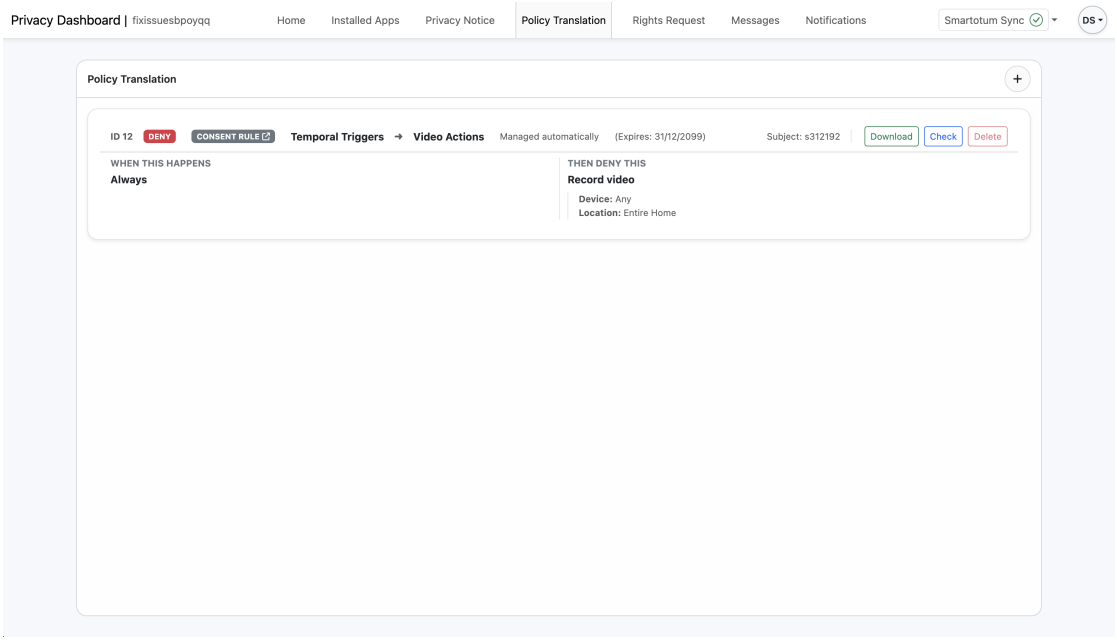


Figure 3.4: Example of a rule denying permission to record video.

3.2.3 Policy Translation Point

Another important addition to the Privacy Dashboard is the PTP (Policy Translation Point), which enables Data Subjects to define high level privacy decisions, such as allowing or denying specific actions within a given time window and location, as shown in Figure 3.5. Specifically, the user can choose which action to control (light or video actions), where the rule applies (a device, a room or the whole home), the time frame in which it applies (from/to a specific time on a specific day), the rule effect (deny or permit), and an expiration date after which the rule expires automatically. These high level decisions are then converted by the system into rules that can be enforced by devices available in the Smartotum environment.

In this Policy Translation page, the Data Subject can also access a dedicated dashboard that serves as the central management interface, showing the list of high-level policies he previously created, as in Figure 3.6. This dashboard provides a transparent overview by displaying all of the trigger and action details, alongside action buttons like "Download", "Check", and "Delete" to allow users to quickly locate and manage specific rules, as well as the "+" button to add new ones. In particular, the "Check" button initiates a conflict detection process designed to verify whether a specific rule introduces logical inconsistencies or redundancies with existing rules. Once submitted, the system processes the rule and returns a comprehensive *Policy Verification Report* presented in a modal window.

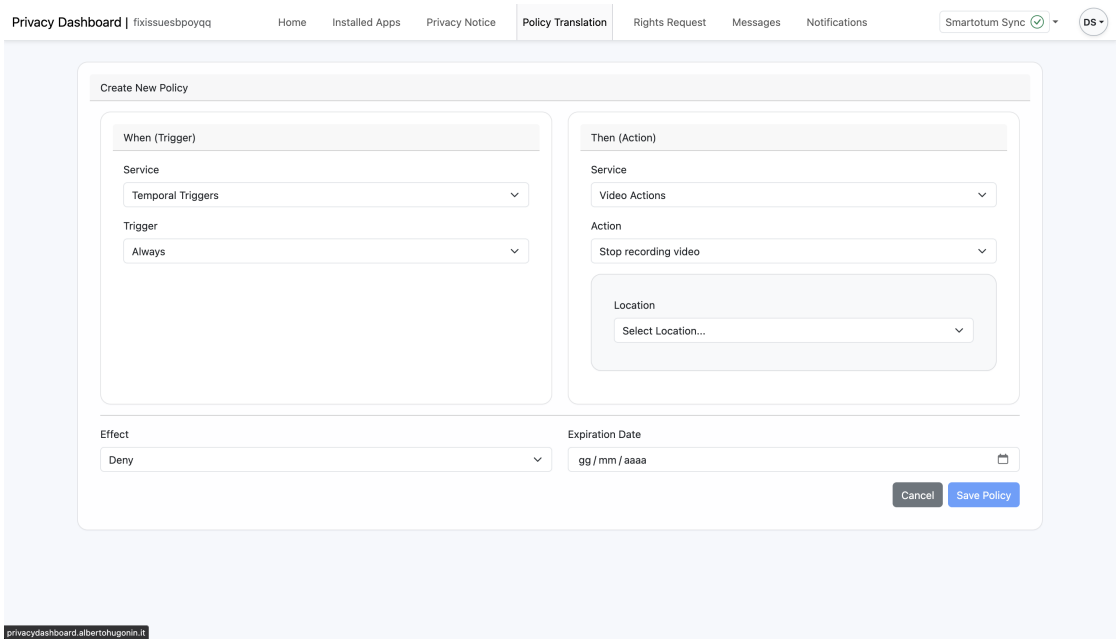


Figure 3.5: Example of privacy policy rule.

As illustrated in Figure 3.7, the feedback is categorized by conflict type using distinct visual color-coding to convey severity:

- **Inconsistent Rules (Yellow)**: Rules with overlapping conditions but conflicting effects (e.g., one rule permits an action while another denies the exact.
- **Redundant Rules (Blue)**: Rules that perform the exact same logic and yield the same effect. Although not logically dangerous, highlighting redundancies helps users maintain a clean and efficient policy set.

Each identified issue presents the full context of the involved rules, including the trigger, action, device name, and room location, alongside inline "Delete" buttons. This empowers the user to resolve the conflict immediately from the report interface without needing to manually search for the problematic rules in the main dashboard.

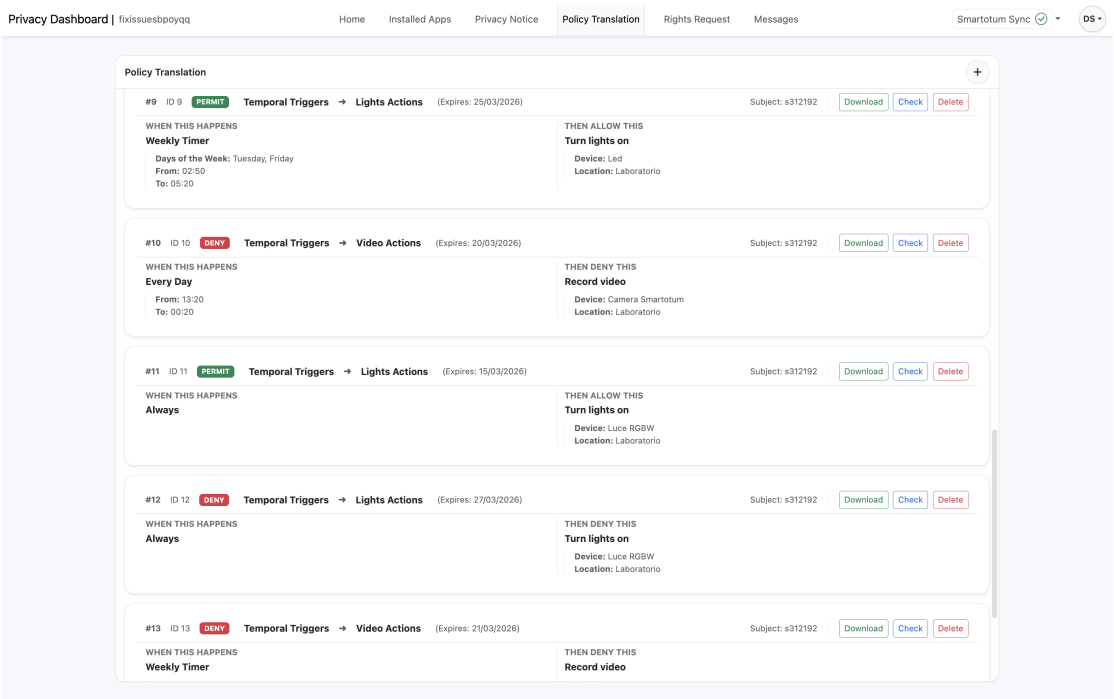


Figure 3.6: Overview of created high-level policies with action buttons.

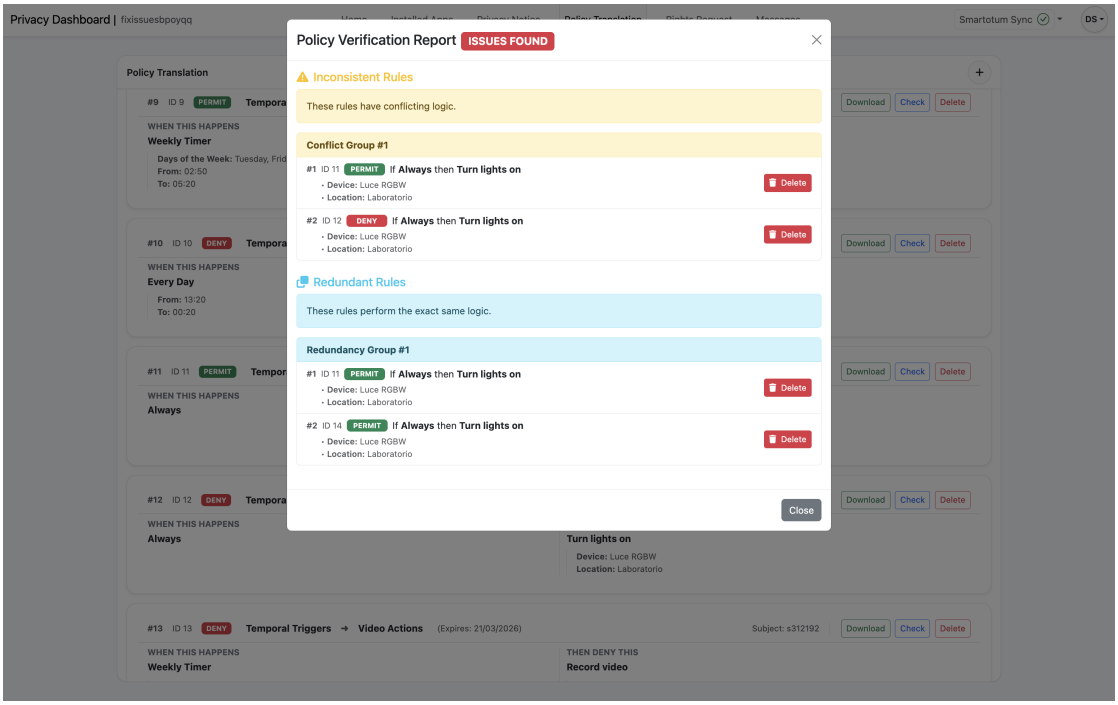


Figure 3.7: Policy verification report displaying categorized conflicts.

3.2.4 Marketplace API

While Smartotum does not currently provide an external interface for installing applications in a smart home, the Privacy Dashboard should remain open to future integrations with home automation systems. For this reason, the platform needs an interface that allows an external service to add applications so that they can be managed and, if supported, deployed to a smart home.

The Marketplace API (Application Programming Interface) within the Privacy Dashboard allows an authorized entity to add a new application and assign it to a Data Controller and their delegates by providing a valid application id, name and description. Once an application has been registered, the Privacy Dashboard becomes the source of truth for GDPR documents, role management, and available consents, while the Marketplace can only update the application title and description.

3.2.5 Manual Creation of Apps

An application can be created manually by a Data Controller, who will then be set as its owner, using the modal shown in Figure 3.8, with the following fields:

- **Application ID**: for manually created applications, the ID always has the prefix `com.privacydashboard` followed by a descriptive suffix.
- **Application name**: the name displayed throughout the dashboard.
- **Available consents**: the list of consents users can accept for the application.

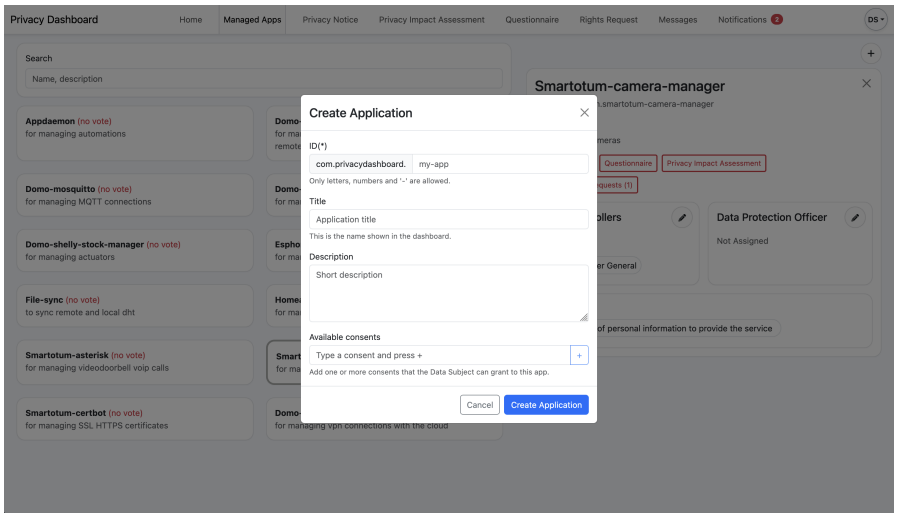


Figure 3.8: Create Application modal for Data Controllers.

3.2.6 Available Consents Management

Available consents can also be added and removed by the owner of the application following its creation, as shown in Figure 3.9.

Because Smartotum is the source of truth for applications installed in a Data Subject's home, the Privacy Dashboard treats their available consents as read-only to avoid inconsistencies. For this reason, adding or removing available consents is enabled only for manually created and Marketplace applications.

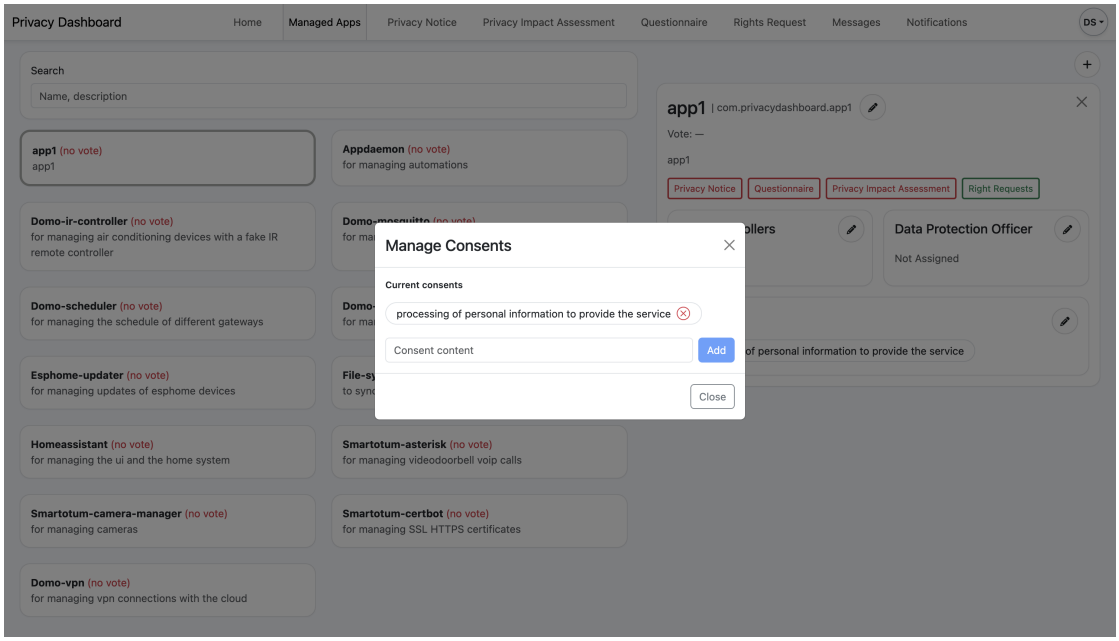


Figure 3.9: Manage Available Consents for Data Controllers.

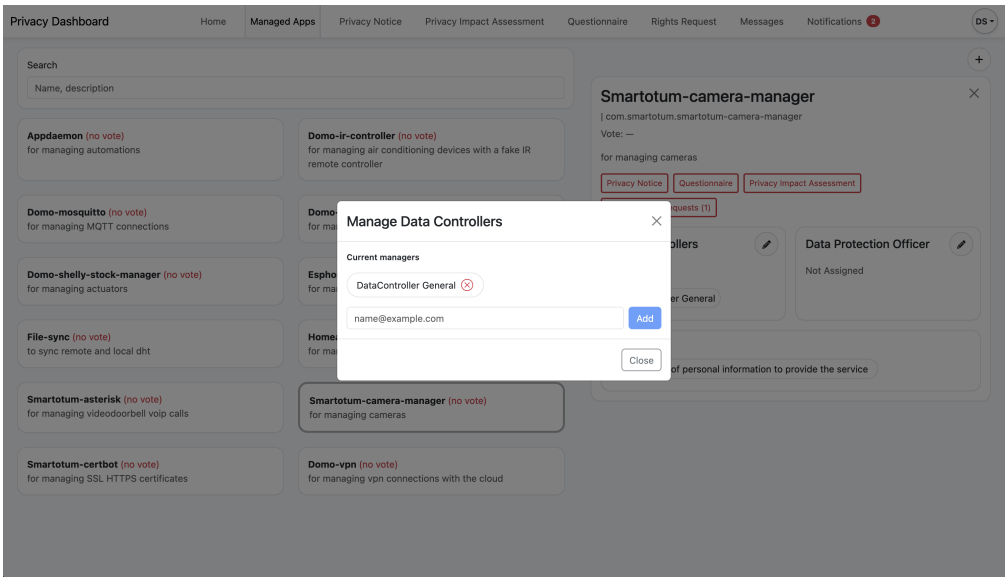
3.2.7 Role Assignment

By supporting different sources of applications, a role assignment process is needed to ensure that every application is linked to the correct Data Controller and, when applicable, to a Data Protection Officer, so that the Privacy Dashboard can act as an all-in-one platform for managing privacy in Smart Home systems. This role management functionality is shown in Figure 3.10.

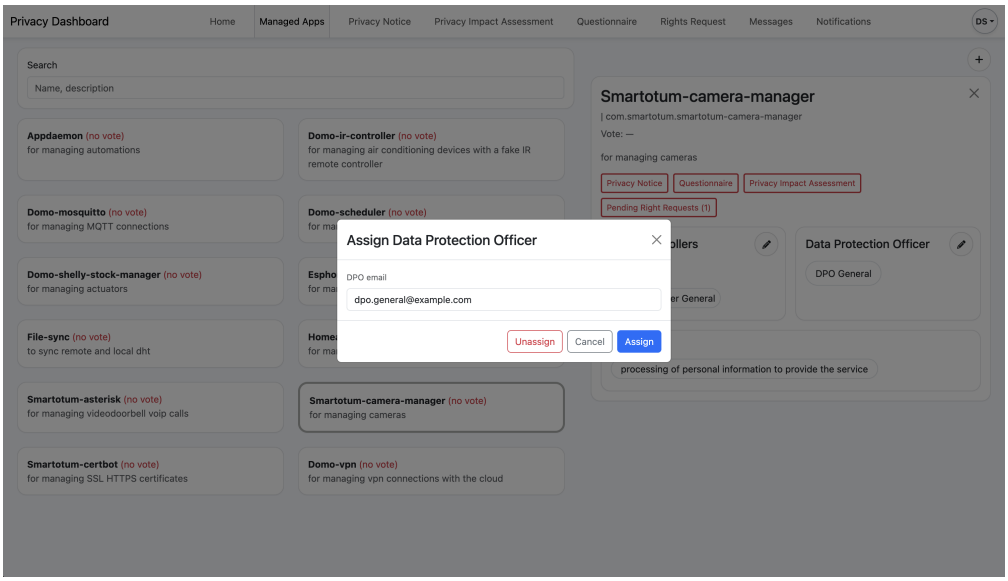
Manage Data Controllers The Primary Data Controller of an application, referred to as "the owner" for simplicity, can appoint additional Data Controllers as managers. Managers can perform the same operations as the owner (creating or editing GDPR Documents, editing available consents, and responding to rights

requests), but they cannot modify the managers list and assign a Data Protection Officer.

Assign Data Protection Officer The owner of an application can also appoint, remove, and replace a Data Protection Officer when needed.



(a) Manage Data Controllers



(b) Assign Data Protection Officer

Figure 3.10: Role Management.

3.2.8 Privacy Impact Assessment

As described in 2.1.5, the PIA (Privacy Impact Assessment) is an important document for assessing risks related to personal data processing. For this reason, building on the limitations discussed in 2.2.2, the Privacy Dashboard provides a dedicated page for its creation and management, as shown in Figure 3.11. On the left, applications can be selected or filtered based on their name, description, and the status of the PIA ("Available" or "Missing"). On the right, the document can be viewed when available and, using the toolbar, Data Controllers and Data Protection Officers can create, edit, delete, or download it as a PDF file.

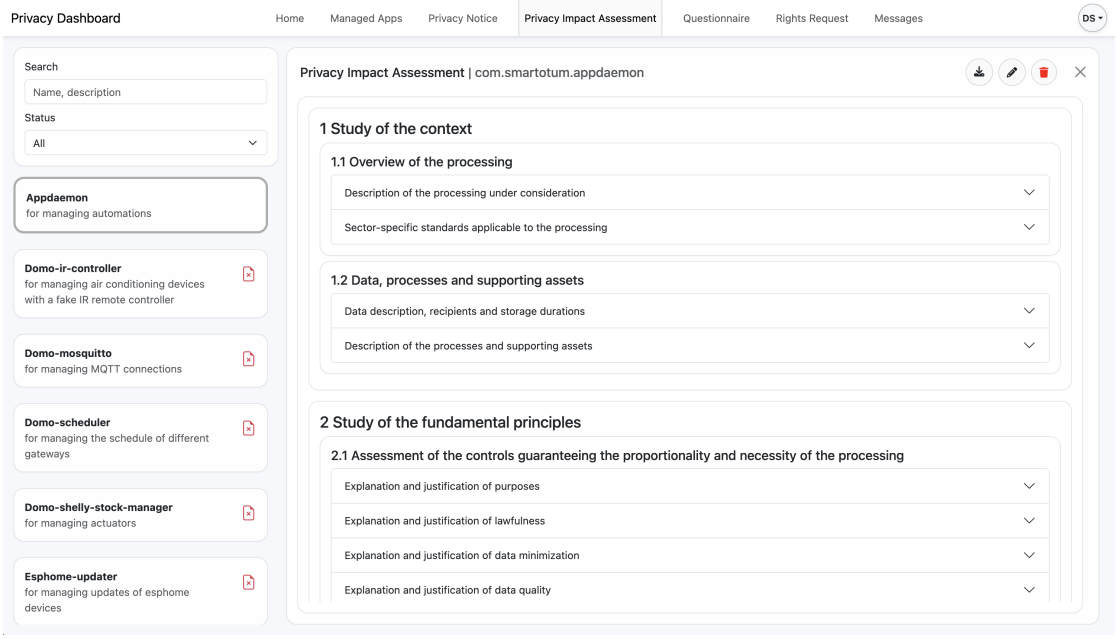


Figure 3.11: Privacy Impact Assessment page for Data Controllers and DPO.

3.2.9 Privacy Notice

In the context of GDPR, another document related to user's privacy protection and defined in 2.1.5 is the Privacy Notice. Similarly to the page devoted to the PIA, Privacy Dashboard provides a dedicated page for the creation and management of Privacy Notices, as shown in Figure 3.12. On the left, applications can be selected or filtered as in the PIA page. On the right, the Privacy Notice can be viewed when available, and, using the toolbar, Data Controllers and Data Protection Officers can create, edit, delete, or download it as a PDF file, whereas Data Subjects have only permissions to view and download. A new feature is the ability for a Data

Controller or Data Protection Officer to apply a Privacy Notice to an existing application without manually copying its contents, as in Figure 3.13.

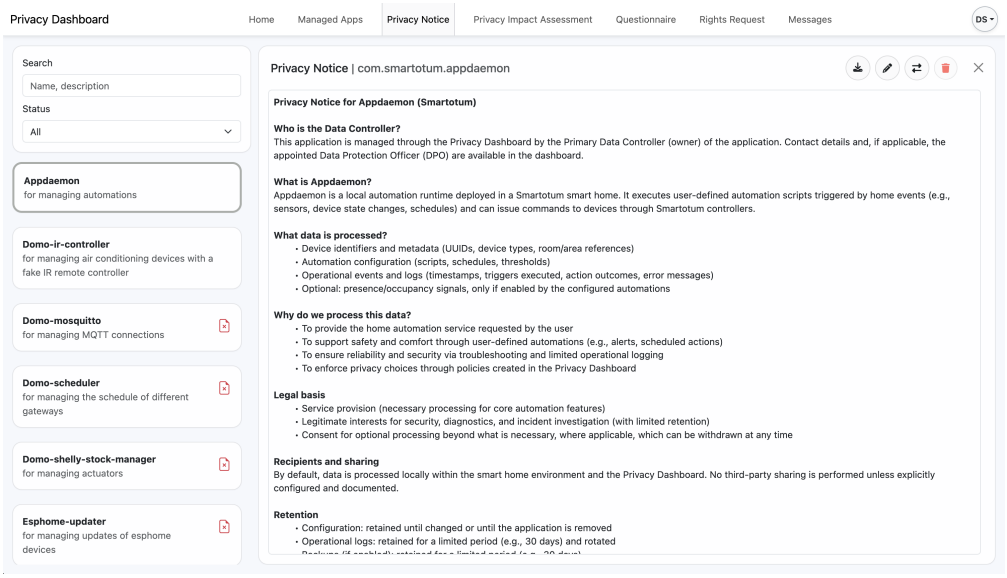


Figure 3.12: Privacy Notice page for Data Controllers and DPO.

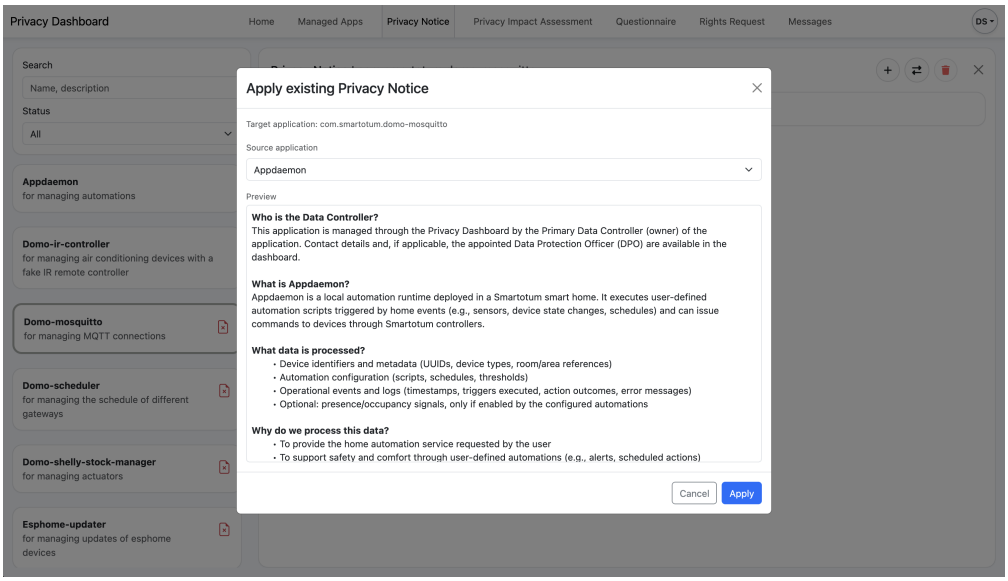


Figure 3.13: Apply Existing Privacy Notice.

3.2.10 Questionnaire

The last document we feature in the context of GDPR is the Questionnaire, which Data Controllers and Data Protection Officers can carry out for any application. It is made up of a set of 30 multiple choice questions, where each given answer can receive a color coded evaluation, which represents how much the selected option is compliant to GDPR. The application itself can then receive an evaluation, as a function of all the evaluations received by the 30 answers. Figure 3.14 shows our user interface to manage such document. On the left, applications can be selected or filtered as in the PIA and Privacy Notice pages. On the right, the completed Questionnaire can be viewed when available, together with the overall evaluation of the corresponding application and a summary that states how many answers received each possible evaluation. Using the toolbar, Data Controllers and Data Protection Officers can create, edit, delete, or download a Questionnaire as a PDF file.

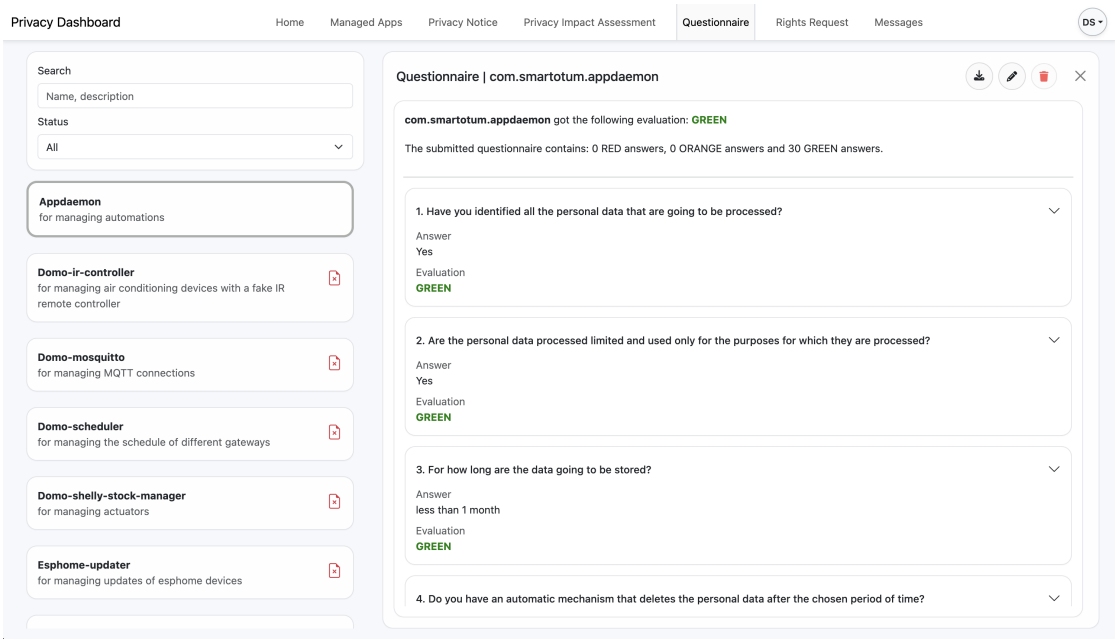


Figure 3.14: Questionnaire page for Data Controllers and DPO.

3.2.11 Messages and Contacts

The messages tab brings users to a modern messaging interface where they can open direct chats with any of their contacts. In the context of this project, a *contact* of a user is defined as such:

- for a Data Controller or Data Protection Officer, it is any user of any role who has at least one managed or installed application in common with the given Data Controller or Data Protection Officer.
- for a Data Subject, it is any user of role Data Controller or Data Protection Officer who has at least one managed application in common with the given Data Subject.

Figure 3.15 gives an example of a chat in this interface, which is split in two parts. On the left, a list of past chats is shown, where the user can see, for each of them, the corresponding receiver, the content, and date and time of the last exchanged message. On the bottom right of the list, there is a button to open a modal that lists every contact the user has, including those with whom he had never started a chat. On the right, there is the opened chat that the user selected from the list on the left, with the name and email of the receiver on the top. The messages exchanged in this example highlight some features of this chat interface, such as the possibility of sending multi-line messages and the fact that messages sent by the same user at the same date and time are automatically wrapped together.

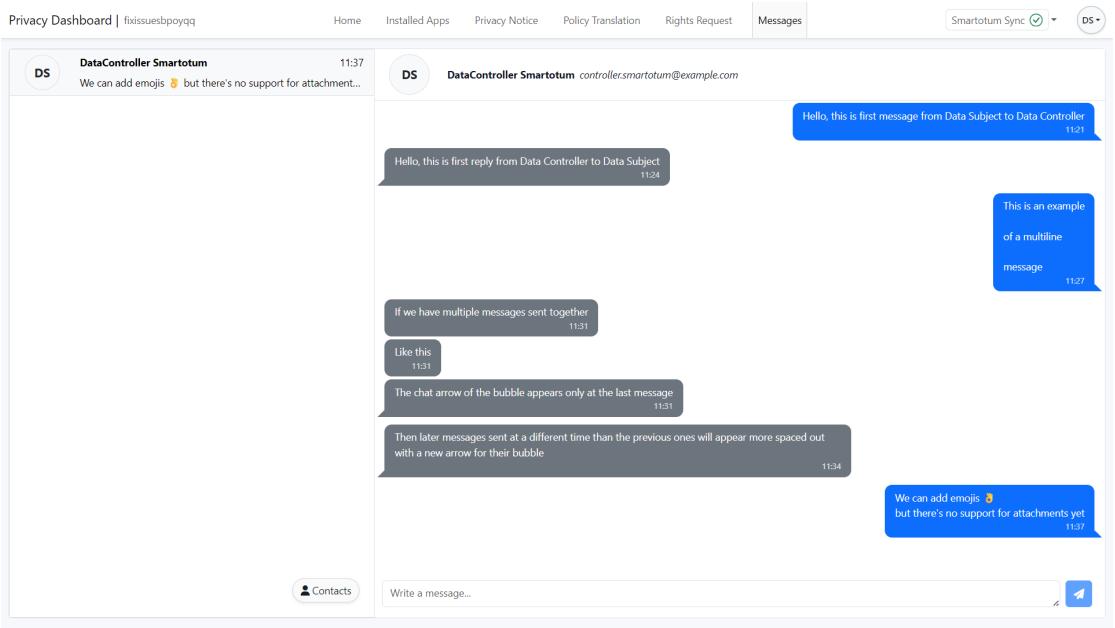


Figure 3.15: Messages and contacts page.

3.2.12 Notification History

Another important feature added to the Privacy Dashboard is the Notification History. This page allows every user role to keep track of the main events that occur in the platform.

At the moment, notifications are used only for events linked to consent acceptance and rights requests. For example, when a Data Subject grants a consent for an application, the associated Data Controllers and Data Protection Officer receive a notification for accounting purposes. Similarly, a notification is also received by a Data Controller and Data Protection Officer when a rights request is sent by a Data Subject. The Data Subject can also receive updates about the handling of their rights requests.

Notification History Page These events are collected in a dedicated page, where the user can inspect past notifications and mark them as read. An unread counter is also shown in the navigation bar, so that newly received events can be noticed immediately. By clicking on a notification, the corresponding page, such as the rights request page, is opened and the notification is marked as read. To make accounting easier, especially when the number of notifications grows, it also provides some filters, such as filtering by read status, type, right-request type, and date. An example of this page is shown in Figure 3.16.

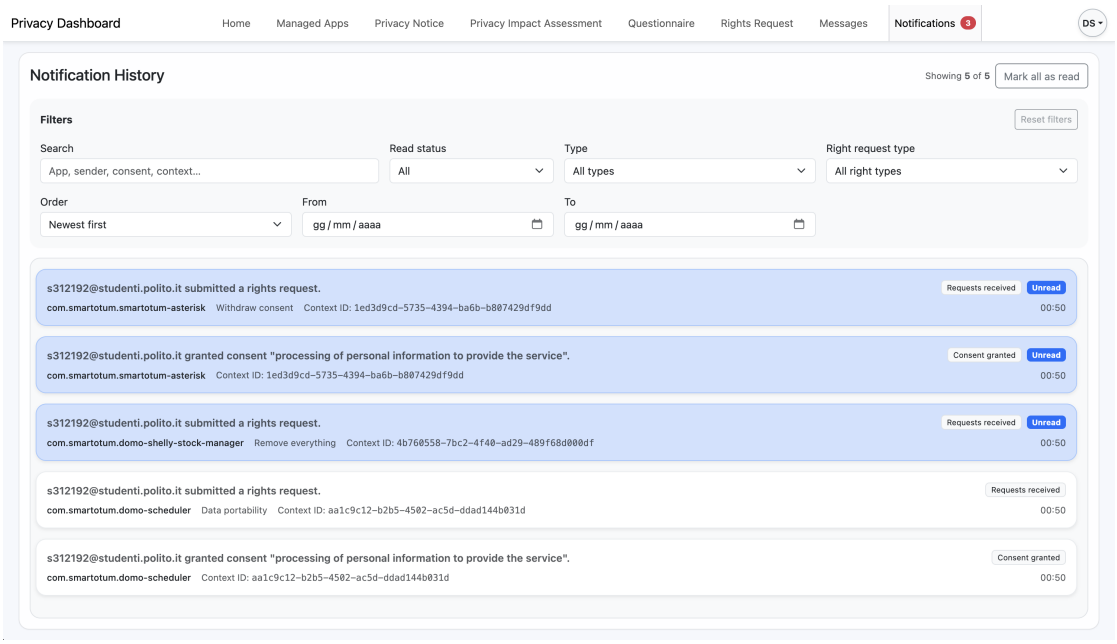


Figure 3.16: Notification History page

Audit Preservation An important function of the Notification History page is also to support audit and accountability. For this reason, notifications are preserved even if the related users, applications, or rights requests are later removed from the platform. When rights requests involve the same Data Subject, application, and Smart Home, they may appear very similar, because the system deliberately hides information about the user's Smart Home. For this reason, notifications also show a context identifier that is used by the system to correlate the user, Smart Home, and application, while hiding this information from the Data Controller and the Data Protection Officer.

3.3 End-to-End Usage Scenario

In order to better visualize how the different interfaces described so far can be used together in a unified flow, we can consider the following usage scenario, where a Data Controller creates a new application, sets up some available consents for Data Subjects to accept, and carries out and downloads the three corresponding GDPR documents.

1. After logging in with his credentials to access Privacy Dashboard as a Data Controller, the user navigates to the tab where he can see his currently managed applications.

2. From there,the Data Controller clicks on the button that opens the modal to create a new application, as in Figure 3.8. He fills out the fields of the creation form, so that he can click the button to save the new application, of which he will be automatically set as owner. Being the owner allows him to also add and remove available consent for that application at a later time, as in Figure 3.9.

3. The Data Controller now begins with the creation of the corresponding GDPR documents, starting from the Privacy Notice. He goes to the dedicated page, as in Figure 3.12, where he can create a new one either from scratch, from the template, or by associating an existing one to the application. Once done, he can choose to download this Privacy Notice as a PDF file by clicking on the dedicated button.

4. The Data Controller then moves to the page of the PIA, as in Figure 3.11. Upon clicking on the button to create a new PIA, he will be prompted to fill out all of the corresponding tables. Once submitted, the newly created PIA can also be downloaded as a PDF file.

5. Lastly, the Data Controller navigates to the page to fill out the Questionnaire for such application, as in Figure 3.14. Once he has answered to all of the 30 multiple choice questions, he can see and download as a PDF file the result of the submitted Questionnaire.

6. Since the Data Controller was set as the owner of the application he created, he can also choose to appoint additional Data Controllers and/or a Data Protection Officer, as in Figure 3.10. These newly assigned users can then be directly contacted through the messages page, as in Figure 3.15.

3.4 Functional Requirements

Below we present the functional requirements defined in the requirements document during the initial analysis of the Privacy Dashboard and subsequently reviewed and updated.

ID	Name
FR1	**Manage Accounts**
FR1.1	Enable the Unlogged User to create a new account choosing a role which is one of the following: Data Subject, Data Controller, Data Protection Officer.
FR1.2	Enable the Unlogged User to log in and access his personal dashboard.
FR1.2.1	Enable Data Subject to see a list of all his Smart Homes right after logging in.
FR1.2.2	Enable Data Subject to choose a Smart Home for which he wants to manage applications.
FR1.2.3	Enable the Smartotum system to return to the Privacy Dashboard the list of Smartotum applications of the Data Subject installed in the chosen Smart Home.
FR1.3	Enable Data Controller and Data Protection Officer to manage their account from the dashboard. Data Controllers can change their password; Data Protection Officers can change both email and password.
FR1.4	Enable the user to log out.
FR2	**Manage Contacts**
FR2.1	Enable the user to see a list of available contacts.
FR2.1.1	Enable Data Controller and DPO to view a list of all the users who share at least one application with them and send them updates, notices, and other important information.
FR2.1.2	Enable Data Subject to view all of the Data Controllers and DPO associated with at least one application installed by that Data Subject.
FR2.2	Enable the user to see each contact's details, including name, email if any, role, and the applications shared with that contact.
FR2.3	Enable the user to select a contact to send and receive messages to and from that contact.
FR3	**Manage Messages**
FR3.1	Enable the user to send and receive messages to and from any contact with a dedicated interface.

ID	Name
FR3.2	Enable the user to see a list of contacts to which he has sent or received messages to resume a conversation.
FR3.3	Enable the user to search for available contacts by name, email, role, or shared app name within the messaging interface when starting a new chat.
FR3.4	Enable the user to start a new conversation by selecting a contact from the list of available contacts.
FR4	**Manage Rights**
FR4.1	Enable Data Controller and DPO to see a list of all the requests received from different data subjects regarding different applications, and to distinguish different Data Subject, application, and Smart Home combinations without seeing the concrete Smart Home information.
FR4.2	Enable Data Controller and DPO to access details of each request in the list, including the type of request, the application it is related to, the date it was received, the current status of the request, the data subject's name, email, the request identifier, and the specific right being exercised.
FR4.3	Enable Data Controller and DPO to quickly see which requests are pending and which have been handled.
FR4.4	Enable Data Controller and DPO to answer requests from Data Subjects.
FR4.5	Enable Data Controller and DPO to change the status of requests from Data Subjects.
FR4.6	Enable Data Subject to see a list of all the GDPR rights supported by the dashboard that he can exercise, including access, additional information, portability, erasure, restriction of processing, complaint, removal of all personal data, and consent withdrawal.
FR4.7	Enable Data Subject to submit a request to a Data Controller or DPO from the selected Smart Home by choosing the specific right he wants to exercise and the application it is related to and by eventually providing additional information.
FR4.8	Enable Data Subject to monitor the status of the requests he has submitted from the selected Smart Home.
FR4.9	When a Data Subject submits a consent-withdrawal request, automatically reconfigure the corresponding IoT device immediately by interacting with the Smartotum system through dedicated APIs.
FR5	**Manage Applications**
FR5.1	Enable Data Controller and DPO to see a list of all the Smartotum applications for which he is responsible, retrieved from the Smartotum system.

ID	Name
FR5.2	Enable Data Controller and DPO to access details of each application in the list, including the application's name, description, evaluation, Privacy Notice, other joint data controllers or DPO if any.
FR5.3	Enable Data Subject to see a list of all the Smartotum applications installed in the selected Smart Home which are currently processing their data, by synchronizing with the Smartotum system.
FR5.4	Enable Data Subject to access details of each application in the list, including the application's name, description, evaluation, Privacy Notice, the list of all data controllers and DPO involved in the processing of their data, the list of consents he has provided.
FR5.5	Enable Data Subject to give consents to a specific application in the selected Smart Home by choosing from the application's predefined list of available consents.
FR5.6	Enable Data Subject to define and apply a new high-level policy in the selected Smart Home, targeting a specific device, a room, or the entire home.
FR5.7	Enable Data Subject to quickly and easily request the withdrawal of selected consents from a specific app in the selected Smart Home.
FR5.8	Enable Data Subject to quickly and easily request the withdrawal of all their given consents from a specific app in the selected Smart Home.
FR5.9	Enable the Data Subject to quickly and easily request the removal of all their personal data from a specific app.
FR5.10	Enable the Marketplace system to automatically add an application to the list of Smartotum applications of a Data Controller when the Data Controller installs it from the Marketplace system.
FR5.11	When a Data Subject gives consents or submits a consent-withdrawal request, automatically reconfigure the corresponding IoT device to align with the updated privacy consent by interacting with the Smartotum system through dedicated APIs.
FR5.12	Enable the system to automatically enforce deny rules in Smartotum when the current consent state requires that an application action must not be allowed, including cases in which a consent is not given or is later withdrawn.
FR6	**Manage Privacy Notices**
FR6.1	Enable the Data Controller and DPO to see a list of applications for which he is responsible along with their associated privacy notices, if any.
FR6.2	Enable Data Controller and DPO to edit an existing privacy notice associated to an application.

ID	Name
FR6.3	Enable Data Controller and DPO to write from scratch a privacy notice to be associated to an application.
FR6.4	Enable Data Controller and DPO to write, starting from a provided template, a privacy notice to be associated to an application.
FR6.5	Enable Data Controller and DPO to associate to an application an already existing privacy notice.
FR6.6	Enable the user to see a list of applications installed in the selected Smart Home which are currently processing their data along with their associated privacy notices.
FR6.7	Enable the user to download as a PDF file the privacy notice associated to each application in the list.
FR7	**Manage Questionnaires**
FR7.1	Enable Data Controller and DPO to see a list of all the applications for which he is responsible, which are color-coded based on the results of their respective questionnaires which verify compliance with the GDPR regulations.
FR7.2	Enable Data Controller and DPO to carry out a questionnaire for a specific application.
FR7.3	Enable Data Controller and DPO to edit answers previously submitted to a questionnaire for a specific application.
FR7.4	Enable Data Controller and DPO to see a summary highlighting the critical aspects of a specific application given the answers submitted to the corresponding questionnaire.
FR7.5	Enable Data Controller and DPO to download as a PDF file the carried out questionnaire for a specific application.
FR8	**Manage Privacy Impact Assessments**
FR8.1	Enable the Data Controller and DPO to see a list of applications for which he is responsible along with their associated privacy impact assessments, if any.
FR8.2	Enable Data Controller and DPO to carry out a privacy impact assessment for a specific application.
FR8.3	Enable Data Controller and DPO to edit an existing privacy impact assessment associated to an application.
FR8.4	Enable Data Controller and DPO to download as a PDF file the carried out privacy impact assessment for a specific application.
FR9	**Manage Notifications**

ID	Name
FR9.1	Enable the system to store and show a notification history for accounting and accountability when consent or rights-request actions are performed.
FR9.1.1	Giving a consent for an application: notification to all the data controllers and DPO bound to the application.
FR9.1.2	Submitting a rights request: notification to the related data controllers and DPO.
FR9.1.3	Answering a rights request: notification to the requester.
FR9.1.4	Updating the status of a rights request: notification to the requester.
FR9.2	Enable the system to preserve the information associated with each notification so that the notification history remains understandable for accounting and accountability purposes even if the related users, applications, or requests are later removed.
FR10	**Smartotum Synchronization**
FR10.1	Enable Privacy Dashboard to synchronize Smart Homes and Smartotum applications for a Data Subject at login and through explicit refresh actions, so that the dashboard can realign its local data with the current state of the Smartotum system.

Below we present a table that maps each functional requirement defined above to the corresponding functionality available in the frontend and to the corresponding endpoints it calls in the backend.

ID	Dashboard functionality	Main backend endpoints used
FR1.1	User registration (local account for DC/DPO, Smartotum-linked registration for DS)	`POST/auth/register`
FR1.2	User login and session restore	`POST/auth/login` `GET/auth/current`
FR1.2.1	Data Subject home list shown after login	`GET/homes`
FR1.2.2	Data Subject selects the working Smart Home in the dashboard	No dedicated endpoint. Client-side home selection; the selected `smart_home_uuid` is reused by subsequent scoped requests.
FR1.2.3	Synchronization and retrieval of Smartotum applications for the selected Smart Home	`POST/applications/refresh` `GET/applications/home/<smart_home_uuid>`

ID	Dashboard functionality	Main backend endpoints used
FR1.3	Profile management for DC/DPO from the user menu	`PATCH/auth/current`
FR1.4	Logout	`POST/auth/logout`
FR2.1	Available contacts list	`GET/users/contacts`
FR2.1.1	DC/DPO contact discovery and messaging toward users who share at least one application	`GET/users/contacts` `POST/messages/send`
FR2.1.2	DS visibility of Data Controllers and DPO associated with installed applications	`GET/users/contacts`
FR2.2	Contact details with shared applications in the contacts modal	`GET/users/contacts`
FR2.3	Selecting a contact and opening the conversation	`GET/messages/history/<user_uuid>` `POST/messages/send`
FR3.1	Send and receive messages in the chat interface	`GET/messages/history/<user_uuid>` `POST/messages/send`
FR3.2	Conversation list ordered by latest exchanged message	`GET/users/contacts/messaged`
FR3.3	Search available contacts by name, email, role and shared app name when starting a new chat	`GET/users/contacts` Filtering is currently performed client-side after fetch.
FR3.4	Start a new conversation from the contacts modal	`GET/users/contacts` `POST/messages/send`
FR4.1	Manager inbox of rights requests, distinguishable across requester, application, and Smart Home combinations without exposing concrete Smart Home information	`GET/requests`
FR4.2	Rights request detail panel (requester name/email, app, dates, status, right type, request identifier)	`GET/requests` `GET/requests/application/<application_uuid>/user/<user_uuid>`

ID	Dashboard functionality	Main backend endpoints used

FR4.3	Filtering requests by pending/handled state	`GET/requests?status=PENDING`
		`GET/requests?status=HANDLED`
		`GET/requests/home/<smart_home_uuid>?status=...`

FR4.4	Manager answer to a rights request	`PUT/requests/handle/<request_uuid>`
FR4.5	Manager status update of a rights request	`PUT/requests/handle/<request_uuid>`
FR4.6	DS list of supported GDPR rights in the submit-right modal (access, additional information, portability, erasure, restriction of processing, complaint, removal of all personal data, consent withdrawal)	No dedicated endpoint. Rights are currently exposed by frontend enums and submit forms.
FR4.7	DS submission of a rights request for the selected app and home	`POST/requests/submit`
FR4.8	DS monitoring of submitted requests for the current Smart Home	`GET/requests/home/<smart_home_uuid>`
FR4.9	Immediate IoT reconfiguration when a consent-withdrawal request is submitted	`POST/requests/submit` Smartotum enforcement is triggered server-side during request submission.
FR5.1	Managed-applications list for Data Controller / DPO	`GET/applications/managed`

ID	Dashboard functionality	Main backend endpoints used
FR5.2	Managed-application details: metadata, evaluation, Privacy Notice, joint data controllers and DPO	GET/applications/managed PATCH/applications/managed/<application_uuid> PUT/applications/managed/<application_uuid>/dpo/<user_email> DELETE/applications/managed/<application_uuid>/dpo PUT/applications/managed/<application_uuid>/managers/<user_email> DELETE/applications/managed/<application_uuid>/managers/<user_email>
FR5.3	Installed-applications list for the selected Smart Home	GET/applications/home/<smart_home_uuid> POST/applications/refresh
FR5.4	DS application details (controllers, DPO, evaluation, Privacy Notice, consents)	GET/applications/home/<smart_home_uuid> GET/privacy-notices/installed-application/<application_uuid> GET/consents/home/<smart_home_uuid>/application/<application_uuid>
FR5.5	DS consent giving for a specific application	GET/consents/home/<smart_home_uuid>/application/<application_uuid> PUT/consents/home/<smart_home_uuid>/application/<application_uuid>
FR5.6	High-level policy definition / policy translation for the selected Smart Home, targeting an entire home, a room or a specific device	GET/sifis/homes/<home_uuid>/rules POST/sifis/homes/<home_uuid>/rules DELETE/sifis/homes/<home_uuid>/rules/<rule_id> POST/sifis/homes/<home_uuid>/rules/<rule_id>/download GET/sifis/homes/<home_uuid>/rules/<rule_id>/check
FR5.7	DS request of withdrawal of selected consents	POST/requests/submit
FR5.8	DS request of withdrawal of all given consents for the selected app	POST/requests/submit

ID	Dashboard functionality	Main backend endpoints used
FR5.9	DS request of removal of all personal data from an application	`POST/requests/submit`
FR5.10	Marketplace-side registration and update of managed applications	`POST/integrations/marketplace/apps` `PATCH/integrations/marketplace/apps/<application_id>`
FR5.11	Automatic IoT reconfiguration after consent updates or consent-withdrawal requests	`PUT/consents/home/<smart_home_uuid>/application/<application_uuid>` `POST/requests/submit`

FR5.12	Automatic deny-rule enforcement in Smartotum when the current consent state requires that an application action must not be allowed	`POST/applications/refresh`
		`PUT/consents/home/<smart_home_uuid>/application/<application_uuid>`
		`POST/requests/submit`
		Initial not-given enforcement is materialized during Smartotum application refresh.

FR6.1	Managed-applications list with Privacy Notice availability	`GET/applications/managed` `GET/privacy-notices/managed-application/<application_uuid>`
FR6.2	Edit existing Privacy Notice for a managed app	`PUT/privacy-notices/managed-application/<application_uuid>`
FR6.3	Write Privacy Notice from scratch	`PUT/privacy-notices/managed-application/<application_uuid>`
FR6.4	Write Privacy Notice from template	`GET/privacy-notices/template` `PUT/privacy-notices/managed-application/<application_uuid>`
FR6.5	Associate an existing Privacy Notice to a managed app	`PUT/privacy-notices/managed-application/<application_uuid>/privacy-notice/<privacy_notice_uuid>`
FR6.6	DS view of installed applications together with their Privacy Notices	`GET/applications/home/<smart_home_uuid>` `GET/privacy-notices/installed-application/<application_uuid>`

ID	Dashboard functionality	Main backend endpoints used
FR6.7	Privacy Notice PDF download	`GET/privacy-notices/installed-application/<application_uuid>` PDF generation is currently performed client-side from the loaded content.
FR7.1	Managed-applications list color-coded by questionnaire outcome	`GET/applications/managed`
FR7.2	Carry out questionnaire for a managed app	`GET/questionnaires/template` `PUT/questionnaires/managed-application/<application_uuid>`
FR7.3	Edit questionnaire answers for a managed app	`GET/questionnaires/managed-application/<application_uuid>` `PUT/questionnaires/managed-application/<application_uuid>`
FR7.4	Questionnaire summary and critical aspects view	`GET/questionnaires/managed-application/<application_uuid>`
FR7.5	Questionnaire PDF download	`GET/questionnaires/managed-application/<application_uuid>` PDF generation is currently performed client-side from the loaded questionnaire.
FR8.1	Managed-applications list with Privacy Impact Assessment availability	`GET/applications/managed` `GET/privacy-impact-assessments/managed/<application_uuid>`
FR8.2	Carry out Privacy Impact Assessment	`GET/privacy-impact-assessments/template` `PUT/privacy-impact-assessments/managed/<application_uuid>`
FR8.3	Edit Privacy Impact Assessment	`GET/privacy-impact-assessments/managed/<application_uuid>` `PUT/privacy-impact-assessments/managed/<application_uuid>`

ID	Dashboard functionality	Main backend endpoints used
FR8.4	Privacy Impact Assessment PDF download	`GET/privacy-impact-assessments/managed/<application_uuid>` PDF generation is currently performed client-side from the loaded PIA.
FR9.1	Notification history for accounting and accountability on consent and rights-request actions	`GET/notifications` `GET/notifications/unread-count` `PUT/notifications/<notification_id>/read` `PUT/notifications/read-all`
FR9.1.1	Consent-granted notification to controllers and DPO bound to the application	`PUT/consents/home/<smart_home_uuid>/application/<application_uuid>` `GET/notifications` `GET/notifications/unread-count`
FR9.1.2	Rights-request submission notification to the related controllers and DPO	`POST/requests/submit` `GET/notifications` `GET/notifications/unread-count`
FR9.1.3	Rights-request answer notification to the requester	`PUT/requests/handle/<request_uuid>` `GET/notifications` `GET/notifications/unread-count`
FR9.1.4	Rights-request status-update notification to the requester	`PUT/requests/handle/<request_uuid>` `GET/notifications` `GET/notifications/unread-count`
FR9.2	Notification history remains understandable even if related users, applications or requests are later removed	No dedicated endpoint. Preservation is implemented server-side in persisted notification snapshots and surfaced via `GET/notifications`.
FR10.1	Smartotum synchronization in the login flow and through explicit refresh actions	`POST/auth/login` `POST/homes/refresh` `POST/applications/refresh` `GET/homes` `GET/applications/home/<smart_home_uuid>` Refresh endpoints are also invoked by the frontend login-triggered sync flow.

Chapter 4

System Architecture and Technologies

4.1 High-Level Architecture

Since our software operates over the Internet, it can be considered as a *network-based application*. There are many different architectural patterns that could be adopted to implement such type of application, and one of the most common is the *client–server architectural model*, which is also the one we chose. The main idea behind this distributed architecture is to have a clear separation of concerns between the two main components, the *client*, *i.e.* the service consumer, and the *server*, *i.e.* the service provider. The server component, offering a set of services, listens for requests upon those services. The client component, which desires a service to be performed, sends a request to the server via a connector. The server either rejects or performs the request and sends a response back to the client [27], as depicted in Figure 4.1.

The client is where the computation and rendering of the user interface happens, so it results in what the end user sees and interacts with in the browser, while the server has the main function of processing and storing data that goes to and comes from the client that is sending and receiving requests. Within this context, the client is often referred to as the *frontend*, whereas the server is often called the *backend*. The separation of concerns between client and server brings many advantages, where one of them is the possibility for frontend and backend to evolve independently: the former can have its portability improved to work across multiple platforms, the latter can have its scalability improved by simplifying its components [27]. The communication between client and server happens via the Internet and cannot be set up arbitrarily by the developer or the end user, but it has to follow a predefined message exchange protocol, the HTTP (Hypertext Transfer Protocol).

Fielding et al. formally defined HTTP as [28]:

> a family of stateless, application-level, request/response protocols that share a generic interface, extensible semantics, and self-descriptive messages to enable flexible interaction with network-based hypertext information systems.

All requests we mentioned earlier sent by the client are HTTP requests and receive from the server an HTTP response, and they have to follow a precise structure. HTTP requests contain different fields, among which we find: an URL (Uniform Resource Locator), used to identify the resource in the server we want to point to; a method, which describes the type of action we want the server to perform (*e.g.* read or write some desired data); an optional request body, for requests that write or update data stored in the server. Similarly, HTTP responses also have a specific structure, with different fields being, among the others: a status code, which informs the client on the outcome of its request (*e.g.* wether it has failed or completed successfully), and a response body, which contains the data requested by the client. An example of an HTTP request with the corresponding response is given in Figure 4.2. As mentioned in the definition of HTTP, one core property of this protocol is statelessness: each HTTP request is independent, since it must contain all of the information necessary to the server for understanding it, and cannot take advantage of any stored context on the server. This improves the properties of visibility, reliability, and scalability. However, the main disadvantage is that it may decrease network performance by increasing the repetitive data sent in a series of requests, since that data cannot be left on the server in a shared context [27].

Building upon the previously described client–server communication model, the following sections detail the architectural patterns and technologies that implement this interaction, describing how data is stored in the database, processed in the server, sent via HTTP and consumed by the client.

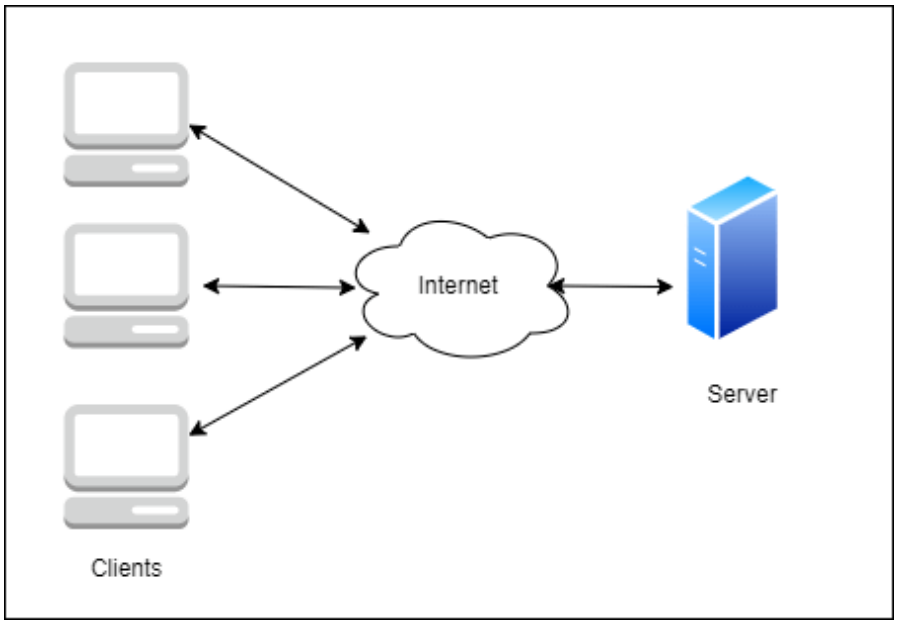


Figure 4.1: The Client-Server Architecture.

Figure 4.2: Example of an HTTP request and response.

4.2 Database Technologies

4.2.1 The Relational Data Model

Storing data so that they can persist across different connection sessions is a crucial part of any web application, and becomes possible with the adoption of a *database*. According to the logical structure that they adopt, databases can be classified into various categories, but the two most common ones are *relational* and *non-relational* databases. We now focus on the former, since the one we implemented falls into the category of relational databases.

Relational databases are a class of databases that manage data relying on the *relational model*, which was first proposed in 1970 by E. F. Codd [29]. He developed this data model by considering the term *relation* in a mathematical sense and by defining it as such:

> Given sets $S_1, S_2, \ldots, S_n$ (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples each of which has its first element from S_1, its second element from S_2, and so on.

In our context, each relation is represented by means of a table that has a set of columns and a set of rows. The relational model formalizes them with the following definitions:

- **Attribute**: a column of a table, with a name and a *domain*.
- **Domain**: value set that can be assumed by an attribute.
- **Degree**: number of attributes in the *heading*, that is the set of attributes in the table.
- **Tuple**: a row of a table.
- **Cardinality**: number of tuples in the *body*, that is the set of tuples in the table.

This model has some important characteristics, among which there are the facts that tuples in a relation are distinct and unordered. In order to ensure that there are no duplicate rows in a table, the relational model introduces the concept of *primary key*: one domain, or combination of domains, of a given relation that has values which uniquely identify each element (n-tuples) of that relation [29]. To provide an example, if we consider a table that stores data about some vehicles where each tuple would be an individual vehicle, then a primary key may be the attribute storing the license plates which uniquely identify each vehicle. Another core characteristic of the relational model is that cross-references between data in

different relations are represented by means of domain values, and this brings to the concept of *foreign key*: a domain, or combination of domains, of relation R that is not the primary key of R but its elements are values of the primary key of some relation S [29]. Figure 4.3 provides a visual example, where the domain of the column "TeacherID" of the "Courses" table is a foreign key since its elements are values of the primary key of the "Teachers" table.

Courses

Code	Name	TeacherID
M2170	Information systems	D101
M4880	Computer Networks	D102
F0410	Databases	D321

Teachers

ID	Name	Department	Phone#
D101	Green	Computer Engeneering	123456
D102	White	Telecommunications	636363
D321	Black	Computer Engeneering	414243

Figure 4.3: Example of cross-reference between two tables.

The relational model brings numerous advantages, and one of them is the possibility of enforcing strong integrity constraints, which can be classified into *intra-relational* constraints and *inter-relational* constraints, both of which contribute to preserving data consistency. Intra-relational constraints are defined on the attributes of a single table and can be, for example, a constraint on the domain, such as imposing that integer values of a column must be positive. Inter-relational constraints are instead defined on many relations at the same time and can be, for example, referential constraint, such as imposing that all values used in the foreign key column of a relation exist as primary keys in tuples of the referenced relation. Through these mechanisms, the relational model prevents invalid database states and enforces coherent relationships among stored data. Another significant advantage of the relational model described by E. F. Codd [29] is the possibility of applying normalization techniques, which aim to reduce structural redundancy. By decomposing relations according to functional dependencies and organizing data into well-defined normal forms, normalization ensures that each fact is represented in a single, appropriate location within the schema, and contributes to a clearer logical organization of the database structure.

4.2.2 PostgreSQL

While the relational model defines the logical structure and integrity constraints governing data organization, these principles must be implemented and enforced by a DBMS (Database Management System). As the name suggests, a DBMS is a software system that enables users to define, create, maintain and control access to the database [30]. A DBMS allows user to perform various operations, starting from the creation of the database *schema*, *i.e.* the definition of the tables the user wants to create, specifying attributes, data types and constraints for each of them. It then allows to access such data, by enabling users to insert, retrieve, update and delete rows from the tables - this set of basic operations is often referred to as *CRUD (Create, Read, Update, Delete)*. In addition to a system that provides CRUD operations, Connolly and Begg listed other important systems that every DBMS shall include [30]:

- **Security system**: prevents unauthorized users accessing the database.
- **Integrity system**: maintains the consistency of stored data.
- **Concurrency control system**: allows shared access of the database.
- **Recovery control system**: restores the database to a previous consistent state following a hardware or software failure.
- **User-accessible catalog**: contains descriptions of the data in the database.

The systems described above collectively enable a DBMS to provide reliable transactional behavior. A *transaction* can be defined as a sequence of operations executed as a single logical unit of work and is characterized by the ACID (Atomicity, Consistency, Isolation, Durability) properties, which Gray and Reuter defined as follows [31]:

- **Atomicity**: A transaction's changes to the state are atomic - either all happen or none happen.
- **Consistency**: A transaction is a correct transformation of the state. The actions taken as as group do not violate any of the integrity constraints associated with the state.
- **Isolation**: Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.
- **Durability**: Once a transaction completes successfully (*i.e. commits*), its changes to the state survive failures.

In the context of our project, it was then important to choose a DBMS that would fully support ACID properties, and that is the reason why we adopted *PostgreSQL*, a free and open-source relational DBMS. First designed at the University of California, Berkeley in 1986 [32], PostgreSQL has now become one of the world's most used DBMS, according to Stack Overflow's 2025 Developer Survey [33]. It was given that name to emphasize its full support to SQL (Structured Query Language), which is the most widespread language to define and manage tables, perform CRUD operations and more. Moreover, PostgreSQL is fully compliant to ACID properties through its MVCC (Multi-Version Concurrency Control) mechanism, so it enables high levels of concurrent access without compromising consistency. Its extensible architecture allows for the definition of custom data types, operators, and indexing methods, making it suitable for a wide range of applications, including our *Privacy Dashboard*.

4.3 Backend Technologies

4.3.1 The Controller-Service-Repository Pattern

Even though PostgreSQL provides the mechanisms necessary for persistent data storage and reliable transaction management, it represents only the bottom layer of the backend infrastructure. A complete backend system must also define how HTTP requests are handled, how application logic is structured, and how interactions with the DBMS are organized. For this purpose, architectural patterns are adopted to ensure a more structured and consistent design and development process that leads to a more maintainable result. There are various architectures that may be implemented to design and structure a backend system, and the two most common are *monolithic* and *microservices*. Monolithic architecture refers to the traditional approach of building software systems as a single, unified application, so all components are packaged and deployed as a single executable unit, typically sharing the same runtime environment; on the other hand, microservices architecture refers to the more modern approach of structuring applications as a collection of small, autonomous services, each responsible for a distinct business capability, so components are free to evolve, deploy, and scale independently [34]. Each architecture presents advantages and drawbacks: monolithic applications are more straightforward to design, develop, test and deploy for smaller scale projects, by not having to orchestrate the communication among different services, but start to show limitations as projects scale in size and complexity; microservices applications instead promote modularity, scalability, and maintainability [34], which is ideal for larger scale projects, but may introduce significant overhead and complexity due to the need to manage multiple services at once, and of setting up proper synchronization and communication among them. Given these considerations and

taking into account the relatively small scale of our team and project, we opted for a backend that follows the monolithic architecture.

Although our backend is deployed as a single monolithic application, its internal structure is organized according to well-defined design principles that foster separation of concerns and modularity. In particular, the system follows a layered organization based on the *Controller–Service–Repository* pattern, which clearly distinguishes request handling, business logic, and data access responsibilities. Separation of concerns, which is the core motivation behind this pattern, is achieved by delegating different scopes and responsibilities to each layer and comes with several advantages, such as allowing them to be tested independently. This pattern defines a clear and well-structured flow of control and data that is received with an HTTP request, is carried through the three layers, and is sent back via a corresponding HTTP response.

Controller When a client sends an HTTP request to the backend, it is handled in the controller layer. Here a function is called based on the URL of the request, and, if input data was given, validation is performed, as to immediately return an error to the client without going further in the data flow if this validation fails (*e.g.* if a string was given when a number was expected, or if a negative number was given when a positive value was expected). If all validation succeeds, the controller calls a service method to perform some business logic, and then receives from it some result data. Finally, the controller uses this data received from the service method to build an HTTP response to send to the client.

Service When called from the controller layer, functions in the service layer perform the business logic of the application, and that may include, for example, executing some more advanced validation on the input data, or transforming the input data in data understandable by the DBMS or vice versa. If the user is requesting an operation that requires accessing the database, functions in this layer will call methods of the repository layer, and, once they receive a returned value, they will handle it and send it back to the controller method that called them.

Repository This is where the interaction between the backend and the DBMS happens. When called from the service layer, functions in the repository layer perform operations which, for example, may consist in executing some SQL statements, and this allows to perform any operation that is provided by the selected DBMS. When data is received back from the DBMS, functions in the repository layer return it to the service method that called them to be further processed.

4.3.2 Flask

While the previous subsection described the internal layering of our backend, it is also necessary to define how the backend exposes its functionality to clients. In web applications, this is commonly achieved through the implementation of a *RESTful API*. Introduced by Fielding in 2000, REST (Representational State Transfer) is defined as an architectural style that provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems [27]. In practice, REST defines a set of constraints that shape the interaction between clients and servers. In our case, these constraints are realized through a collection of HTTP endpoints, each representing a specific resource of the system and accessible via a unique URL. Clients, as mentioned in Section 4.1, interact with these resources by sending HTTP requests using standard methods such as *GET*, *POST*, *PUT* and *DELETE*, which correspond to retrieval, creation, modification and deletion operations, which in turn can be mapped by the backend to CRUD operations to execute on the underlying DBMS. The set of rules that precisely define all of the requests that a backend is able to receive is denoted as an API (Application Programming Interface), thus we can now refer to HTTP requests and HTTP responses as *API requests* and *API responses*. According to REST principles, resources are not transmitted directly; instead, clients and servers exchange *representations* of those resources. Although early web systems primarily exchanged documents in HTML (HyperText Markup Language) format, used to instruct browsers on how to render a particular web page, modern web applications frequently adopt machine-readable formats to enable programmatic interaction between frontend and backend components. In our web application in particular, resource representations are encoded using JSON (JavaScript Object Notation), a data format based on the data types of the JavaScript programming language, *i.e.* collections of key–value pairs and ordered lists. Given that JSON is a language which can be easily understood by both developers and machines, it has become the most popular format to send API requests and responses over the HTTP protocol [35]. We could see an example of a JSON object contained in an API response in Figure 4.2, where we received an object with two key-value pairs.

There are many frameworks that enable the implementation of a backend as a RESTful API that returns JSON objects, and among them we chose *Flask*, a lightweight Python web framework built on the WSGI (Web Server Gateway Interface) specification. It is designed to facilitate the creation of a basic working system, with the ability to scale it up to a complex application, thanks to its support to a wide range of extensions and libraries. According to Stack Overflow's 2025 Developer Survey [36], Flask has become one of the most popular Python web

application frameworks. As a "microframework" [37], Flask provides the essential components required to handle HTTP requests with native JSON support, define routing mechanisms, and generate responses, while leaving architectural decisions and application structure to the developer. This minimalist design aligns well with the Controller–Service–Repository pattern described in 4.3.1, as it allows a clear separation between request handling, business logic, and DBMS access without imposing a rigid framework structure.

Given all these reasons, Flask was considered an optimal solution for the scope of our project.

4.4 Frontend Technologies

4.4.1 Single Page Application Architecture

Having analyzed the main technologies behind our backend, we can now describe the outermost layer of client-server architecture: the *frontend*. As introduced in 4.1, the main role of the client is to send and receive API requests and responses to the server and, based on the data obtained, render a page on the web browser that the end user can interact with. As we have seen for the backend, there exist different architectural patterns to design and develop a frontend system too, where the two most common are *MPA (Multi-Page Application)* and *SPA (Single Page Application)*. In web applications developed following the traditional MPA architecture, the frontend is based on a multi-page interface model, in which on each navigation event requiring new content, the entire page is reloaded: the server is responsible for generating complete HTML documents, which are transmitted to the browser in each API response. This server-side rendering can lead to increased latency and reduced interactivity compared to more dynamic approaches. The SPA pattern was later developed to improve the user experience by focusing on not refreshing the entire web page at every API request: in this new model, the SPA interface is composed of individual components which can be updated and replaced independently, so that the entire page does not need to be reloaded on each user action. This, in turn, helps to increase the levels of interactivity and responsiveness [38], since most of the rendering logic is shifted from the backend to the frontend, so the user no longer has to wait for the backend to generate full HTML documents for each interaction. In order to dynamically update its components, the SPA does not need to receive HTML content, but rather some structured information that it can map to a web layout, and a fitting example of this required data is JSON objects: a RESTful API that returns JSON objects, such as the one we have described in the previous section, is then a type of backend that matches well with a SPA. Whenever the user asks for some new information, the SPA sends a API request to the RESTful backend, which in turn returns the desired data in a JSON

object. The SPA renders its individual components by using the received JSON data to dynamically generate and update the corresponding HTML representation in the browser.

Considering these advantages that a SPA brings for an enhanced user experience, and, most importantly, how our backend is structured, we chose to follow a SPA pattern rather than a MPA, in order to design a client architecture that would seamlessly interact with our RESTful API developed in Flask.

4.4.2 React

The implementation of a SPA is often carried out adopting a dedicated frontend framework that provides a structured set of tools and abstractions that facilitate the development of interactive user interfaces executed directly within the web browser. This client side execution is enabled by the fact that modern browsers embed a JavaScript engine, which is capable of running an application written in a programming language such as JavaScript or TypeScript. There exist many dedicated frontend frameworks useful to build such kind of client application, and the one we adopted for our project is *React*. Developed in 2013 by Meta (formerly Facebook), React is a free and open-source library, and according to Stack Overflow's 2025 Developer Survey [36] it is now one of the most widely-adopted frontend frameworks in the world. The main idea behind React is, as we introduced in the previous section, to break down the SPA interface into many individual reusable components, and this means that React uses a real, full featured programming language to render views which are easier to extend and maintain, compared to traditional HTML templates [39]. React enhances modularity by following a declarative programming pattern, since the developer describes, for each component, how it is structured and how it should behave on changes in the application state. Each component may maintain an internal state, representing dynamic data that determine how it is rendered in the browser. The structure of each component can also include other nested components, and, in order to allow changes in a parent component's state to propagate to its child components through updated properties (*props*), React allows downward state propagation, which results in a flow of updated data from higher to lower levels of the component hierarchy.

A core feature of React is its creation and usage of a virtual DOM (Document Object Model), which is a data structure parallel to the browser's DOM. The browser's DOM is a representation of an HTML document as a logical tree, where each node of the tree corresponds to an HTML object, and the entire tree represents the hierarchical structure of the entire HTML document. An example of the virtual DOM generated by React with its corresponding real DOM is shown in Figure 4.4. Whenever the state of a component changes, React automatically reflects this

change on the corresponding branches and nodes of its virtual DOM. It then runs an algorithm to compute the differences between its updated virtual DOM and the browser's DOM so that it can proceed to update the latter, modifying only the affected nodes and branches, rather than building the entire tree again from its root [40]. This is how React implements the SPA feature, introduced in the previous section, of not reloading the entire page at every user's event, but rather only making as little changes as possible to the rendered page to maximize efficiency and responsiveness. Figure 4.5 shows an example of this feature in action: the first list item element had a class "selected" which was then removed, and this is reflected in the "New Virtual DOM"; the Real DOM is then updated by only removing such class from the corresponding HTML element, rather than re-building the entire HTML tree from the root.

Given the advantages of modularity, reusability and efficiency described above, together with the support of a large ecosystem of libraries that extend its capabilities, we deemed React as a suitable choice for a framework to implement our SPA frontend.

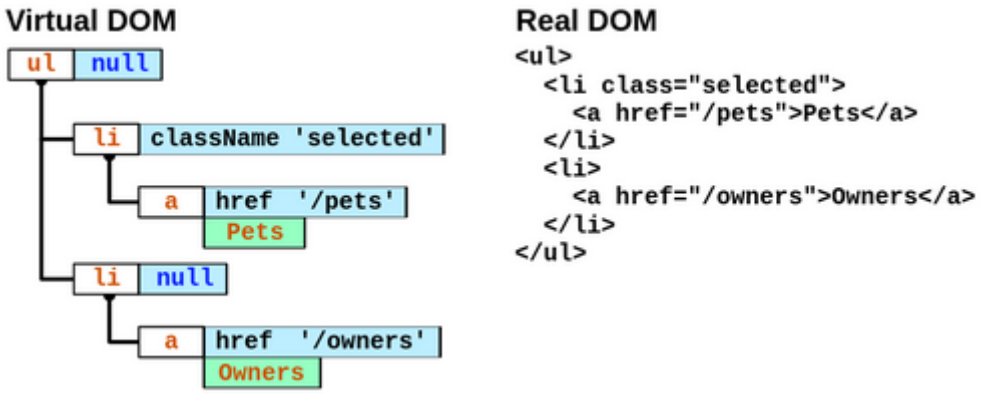


Figure 4.4: Example of virtual DOM compared to browser's DOM.

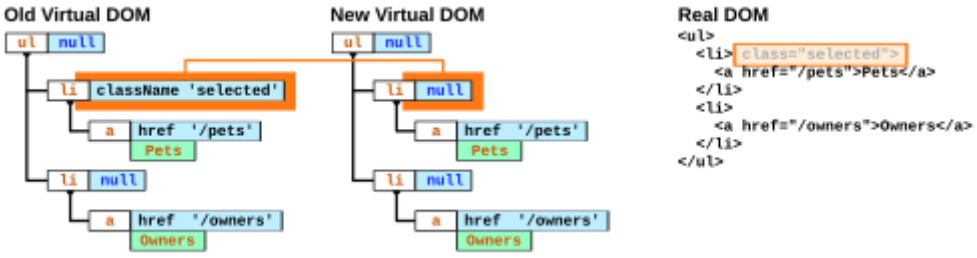


Figure 4.5: Example of update to virtual DOM reflected to browser's DOM.

Chapter 5

Implementation

5.1 Backend Implementation

5.1.1 ORM and Repository Layer with SQLAlchemy

Object–Relational Mapping

As we discussed in 4.2, data is persisted in our database following the relational model, so it is structured in tables with varying rows and columns. However, when dealing with data in our Flask backend we are not directly handing tables, but we are rather following an object-oriented pattern, as data is modeled using Python classes. The relational and object-oriented patterns have some fundamental differences, especially when managing relationships, and thus when accessing objects that reference other objects: with the object-oriented pattern you traverse objects via their relationships, whereas with the relational pattern you join the rows in tables via their relationships [41]; the presence of these differences between the two patterns is referred to as the *Object-Relational Impedance Mismatch* [42]. In order to solve this problem there exist technologies that automate the process of translating the logical representation of the objects we use in our backend into tables stored in the database, and vice versa. This process automation is denoted as *ORM (Object–Relational Mapping)*, whose purpose is also to create an abstraction layer that hides technical details of the mapping implementation, which includes the mapping of data types and relationships [43]. An ORM maps from objects to tables following these general rules:

- A class that represents an object corresponds to a table.
- An attribute of such class corresponds to a column in the table.
- A reference to another class corresponds to a relationship in the table, *i.e.* a foreign key column.

- An instance of such class corresponds to a row in the table.

The adoption of an ORM brings several advantages: having an abstraction over SQL simplifies the integration between the backend and the DBMS, improves readability, and most importantly enhances maintainability. Whenever changes are made in the logical data modeling in the backend, using an ORM allows the developer to avoid having to manually reflect this changes in the SQL statements that define the database schema, as such SQL statements would be automatically generated by the ORM itself, and this helps saving time and reducing the risk of errors. However, the adoption of an ORM comes with some drawbacks too: there may be cases in which not having control over the SQL statements is not ideal, and the abstraction introduced by an ORM may lead to performance overhead in certain scenarios, particularly when complex queries are generated automatically or when inefficient loading strategies are used.

ORM in SQLAlchemy

The most popular library that provides an ORM in Python is *SQLAlchemy*, which is divided into two different APIs, the *Core* and the *ORM*. The official documentation defines them as such [44]:

- SQLAlchemy **Core** is the foundational architecture for SQLAlchemy as a "database toolkit". It provides tools for managing connectivity to a database, interacting with database queries and results, and programmatic construction of SQL statements.

- SQLAlchemy **ORM** builds upon the Core to provide ORM capabilities. The ORM provides an additional configuration layer allowing user-defined Python classes to be mapped to database tables and other constructs, as well as an object persistence mechanism known as the *Session*. It then extends the Core-level SQL Expression Language to allow SQL queries to be composed and invoked in terms of user-defined objects.

The process of declaring a Python class that SQLAlchemy automatically translates into a table is denoted as *Declarative Table*, with such Python class being denoted as a *mapped class*. The mapped class is then any Python class we would like to create, which will have attributes that will be linked to the columns in a database table.

In our project, we distinguish mapped classes in two categories: *entities* and *relations*. An entity class corresponds to the logical representation of the object itself, *i.e.* an object that can exist independently; a relation class is instead used to denote a many-to-many relationship between two (or more) entities, so that such class can easily be mapped by SQLAlchemy into a table that represents the

corresponding many-to-many relationship between the corresponding two (or more) tables. In total, our backend contains 39 entity classes and 5 relation classes. Examples of mapped classes are given below, with simplified versions of two of our entities, the `User` and the `SmartHome` classes, and of the relation class that represents their many-to-many relationship, `UserSmartHome`. Listing 5.1 shows the `User` mapped class, where the `id` and `email` attributes will be mapped as table columns, whereas `owned_smart_homes` uses the `relationship()` construct that is used by SQLAlchemy to define how mapped objects are linked at the ORM level, enabling navigation between related entities.

```
class User(db.Model):
    __tablename__ = "users"
    id: Mapped[int] = mapped_column(primary_key=True)
    uuid: Mapped[UUID] = mapped_column(default=uuid4, unique=True)
    firstname: Mapped[str]
    lastname: Mapped[str]
    email: Mapped[str] = mapped_column(unique=True)
    password_hash: Mapped[str]
    role = mapped_column(Enum(UserRole, validate_strings=True),
   nullable=False)
    profile_picture: Mapped[str | None]
    owned_smart_homes: Mapped[List["UserSmartHome"]] =
   relationship(
        back_populates="user", cascade="all, delete-orphan",
   passive_deletes=True
    )
```

Listing 5.1: Simplified version of the User entity mapped class.

A very similar logic follows for Listing 5.2, where `owned_by_users` tells SQLAlchemy to link via a one-to-many relationship the `SmartHome` and the `UserSmartHome` class.

```
class SmartHome(db.Model):
    __tablename__ = "smart_homes"
    id: Mapped[int] = mapped_column(primary_key=True)
    uuid: Mapped[str] = mapped_column(unique=True)
    name: Mapped[str] = mapped_column(unique=True)
    address: Mapped[str]
    zip_code: Mapped[str | None] = mapped_column(nullable=True)
    country: Mapped[str | None] = mapped_column(nullable=True)
    owned_by_users: Mapped[List["UserSmartHome"]] = relationship(
        back_populates="smart_home", cascade="all, delete-orphan",
    passive_deletes=True
    )
```

Listing 5.2: Simplified version of the SmartHome entity mapped class.

Finally, Listing 5.3 shows the `UserSmartHome` relation class, that in SQLAlchemy is denoted as an *Association Object*, which works as follows: two individual `relationship()` constructs link first the parent side (`User`) to the mapped association class (`UserSmartHome`) via one-to-many (`User.owned_smart_homes`), and then the mapped association class (`UserSmartHome`) to the child (`SmartHome`) side via many-to-one (`UserSmartHome.smart_home`), to form a uni-directional association object relationship from parent, to association, to child (*i.e.* from `User`, to `UserSmartHome`, to `SmartHome`). For a bi-directional relationship, which is actually present in this example, four `relationship()` constructs are used in total to link the mapped association class to both parent and child in both directions. Although this approach introduces additional mapping constructs at the ORM level, it provides a flexible and explicit representation of many-to-many relationships, especially when the association itself may require additional attributes.

```
1  class UserSmartHome(db.Model):
2      __tablename__ = "user_smart_home_relations"
3
4      user_id: Mapped[int] = mapped_column(
5          ForeignKey("users.id", ondelete="CASCADE"), primary_key=
       True
6      )
7      smart_home_id: Mapped[int] = mapped_column(
8          ForeignKey("smart_homes.id", ondelete="CASCADE"),
       primary_key=True
9      )
10
11     user: Mapped["User"] = relationship(
12         back_populates="owned_smart_homes"
13     )
14     smart_home: Mapped["SmartHome"] = relationship(
15         back_populates="owned_by_users"
16     )
```

Listing 5.3: Simplified version of the UserSmartHome relation mapped class.

Repository in SQLAlchemy

Once the mapped classes have been modeled, we can easily access the tables generated in our DBMS, since SQLAlchemy also provides an abstraction for the SQL statements used to perform CRUD operations on tables, with a set of dedicated functions. The result of this is that the Repository layer of our backend, as described in 4.3.1, does not actually contain any SQL code, but rather only Python code, which indeed consists of these SQLAlchemy constructs that in turn are automatically translated in the corresponding SQL statements. Listing 5.4

shows an example of a repository method that allows to query the table storing our users while filtering on a given email. In the last line, `db.session` denotes the SQLAlchemy *Session*, which manages the transactional interaction with the database, and `scalar(statement)` executes the query within this context, returning the corresponding mapped object, if any. We can notice that, in such case, the object will be returned as an instance of the `User` class declared in Listing 5.1.

```
1 class UserRepository:
2     @staticmethod
3     def get_user_by_email(email: str) -> User | None:
4         statement = db.select(User).where(User.email == email)
5         return db.session.scalar(statement)
```

Listing 5.4: Example of a repository method.

5.1.2 The Controller Layer with flask-smorest and marshmallow

As described in 4.3.1, the controller layer of our Flask RESTful API backend is the one in charge of interacting with external systems through a set of structured API endpoints. In order to fully implement such type of backend layer that would satisfy our requirements, we had to resort to two external libraries, *flask-smorest* and *marshmallow*, since Flask does not natively provide all of the functionalities we needed.

The marshmallow library

Our RESTful API receives and returns JSON objects in its endpoints, but our business logic only handles data as Python classes. To automate the process of translating from JSON objects to Python classes, and vice versa, the marshmallow library is used, since it provides schemas that can be used, according to its official documentation, to [45]:

- Validate input data.
- Deserialize input data to app-level objects.
- Serialize app-level objects to primitive Python types. The serialized objects can then be rendered to standard formats such as JSON for use in an HTTP API.

An example of a marshmallow schema used for input data validation and deserialization is shown in Listing 5.5, where `ma` is an instance of `Marshmallow()` provided

by Flask-Marshmallow, a library that acts as a thin integration layer for Flask and marshmallow itself. This `MessageReqSchema` schema was used to verify the input of the API endpoint that allows users to send a message to another user: the validation here consists in ensuring that the given JSON request body contains a field `content`, which is a non-empty mandatory string, a field `attachment`, which is an optional string, and `receiver_uuid`, which is a mandatory UUID field, so it has a precise syntax to follow.

```
1 class MessageReqSchema(ma.Schema):
2     content = ma.Str(required=True, allow_none=False)
3     attachment = ma.Str(required=False)
4     receiver_uuid = ma.UUID(required=True)
```

Listing 5.5: Example of a marshmallow input schema.

The serialization process is implemented in a similar pattern, with the additional possibility, provided that the *flask-sqlalchemy* and *marshmallow-sqlalchemy* libraries are also installed, to automatically generate marshmallow schemas, to be used as JSON response bodies, directly from the SQLAlchemy mapped classes described in the previous section. The class `MessageResSchema` shown in Listing 5.6 is an example of an output schema, used to serialize Python objects into JSON objects given as a response of API endpoints that retrieve exchanged messages. It contains an attribute, `message`, that contains a nested object, `MessageSchema`, which is an automatic marshmallow schema created from the `Message` SQLAlchemy mapped class.

```
1  class MessageSchema(ma.SQLAlchemyAutoSchema):
2      class Meta:
3          model = Message
4          include_fk = False
5          include_relationships = False
6          load_instance = False
7          exclude = ("id",)
8
9  class MessageResSchema(ma.Schema):
10     message = ma.Nested(MessageSchema)
11     sender_uuid = ma.UUID()
12     receiver_uuid = ma.UUID()
```

Listing 5.6: Example of a marshmallow output schema.

The flask-smorest library

The marshmallow schemas we just addressed are used within the controller's functions, which are implemented adopting the flask-smorest library. One core feature of this library is that we can split functions, and thus API endpoints, in groups, according to the functionalities they relate to. Each of these groups is denoted as a *Blueprint*. For example, if we have many endpoints that deal with functionalities related to the `User` class, then these will all belong to the `User` Blueprint, defined in Listing 5.7:

```
user_bp = Blueprint(
    "Users API",
    "user_bp",
    url_prefix="/api/users",
    description="Endpoints for users",
)
```

Listing 5.7: Example of a flask-smorest Blueprint.

Then all of the `Message` endpoints will belong to the `Message` Blueprint, and so on. This grouping helps with separation of concerns and with an improved organization of the project structure, since we can put each Blueprint and all its belonging endpoints into a dedicated Python file. An example of an endpoint that belongs to the `Message` Blueprint is shown in Listing 5.8:

```
@message_bp.route("/send", methods=["POST"])
@message_bp.arguments(MessageReqSchema, location="json", as_kwargs=False)
@message_bp.response(201, MessageResSchema)
@logged_in_required()
def send_message(message, current_user):
    if message["receiver_uuid"] == current_user.uuid:
        raise WrongUserUuidException("Cannot send a message to yourself!")
    return MessageService.send_message(current_user.id, current_user.role, message)
```

Listing 5.8: Example of a flask-smorest endpoint.

This function defines the *POST* endpoint introduced earlier, accessible at the `/api/messages/send` URL, that allows users to send a message to another user:

- the `@message_bp.route()` decorator specifies the path to reach the endpoint and its HTTP method.
- the `@message_bp.arguments()` decorator receives as a parameter the marshmallow `MessageReqSchema` schema we declared in Listing 5.5, and it will perform

validation generating an HTTP error response without requiring additional control logic inside the controller function if the provided JSON request body does not comply with the provided schema.

- the `@message_bp.response()` decorator receives as a parameter, besides the HTTP status code it would return in case of success, the marshmallow `MessageResSchema` schema we declared in Listing 5.6 that specifies the format of the JSON response body returned in case of success.
- the `@logged_in_required()` custom decorator is used for authorization and blocks access to the endpoint if the user is not logged in.

Finally, if the condition to throw the `WrongUserUuidException` is not satisfied, the function calls the `MessageService.send_message()` method, moving the control flow to the service layer, described in 4.3.1, to then provide its returned value as a `MessageResSchema` in the JSON response body.

In total, the controller layer of our RESTful API contains 74 endpoints, each following the implementation pattern shown in Listing 5.8, which are grouped in 14 different Blueprints: `application_bp`, `auth_bp`, `consent_bp`, `dht_bp`, `homes_bp`, `marketplace_bp`, `message_bp`, `notification_bp`, `privacy_impact_assessment_bp`, `privacy_notice_bp`, `questionnaire_bp`, `request_bp`, `user_bp`, `sifis_bp`.

The adoption of flask-smorest, combined with marshmallow, provides a structured and declarative approach to implement the controller layer of our RESTful API. By integrating request validation, object serialization, automatic error handling, and endpoint organization through Blueprints, these libraries reduce boilerplate code and enforce a clear API contract between client and server, resulting in a more maintainable, consistent, and robust backend architecture.

5.1.3 Token-based Authentication with Flask-JWT-Extended

Session-based and token-based authentication

One essential feature of most web applications is the management of security, meant as ensuring that users can only interact with some predefined content and pages, preventing access to content belonging to any other user or to features of the software they are not allowed to use. This leads to the important concepts of *authentication* and *authorization*, which can be given the following definitions [46]:

- **Authentication** is the process of confirming an attribute claimed by an entity. In the vast majority of cases, it is confirmation of identity that the entity claims using credentials.
- **Authorization** is the process of granting permissions on specific actions to given entities.

It is then clear that authentication is a prerequisite for authorization, since it would not be reasonable to grant or deny access to someone or something that we do not know about [47]. We can categorize authentication methods in two broad categories, *session-based* and *token-based* authentication. In session-based authentication, the server creates, upon successful login, an object containing information useful to identify the logged in user, the *session*, and stores it in its memory or in the database, and sends back to the client a cookie containing only the id of such session. Whenever the client makes new requests, it sends this cookie back to the server, which then compares the received session id with the one it has stored and grants access to the requested resources, in case of success. However, this mechanism implies that there is information stored in the server, and that the response to an HTTP request would vary depending on it. This introduces server-side state, which reduces adherence to the REST stateless constraint we described in Chapter 4, that requires an HTTP request to always contain all of the information needed for authentication and requires the server to instead store nothing about it; token-based authentication is then what we used to comply with this principle. In this type of authentication, the server creates, upon successful login, a token that encodes claims that allow it to identify the authenticated user and sends it to the client. Whenever the client makes new requests, it sends this token back to the server, which then has to check its validity and grants access to the requested resources, in case of success. Hence this time no session storage is required at all and the constraint of statelessness is respected. The token exchange between client and server can happen mainly through two different ways: the token can be placed in an HTTP cookie and automatically sent by the browser on subsequent requests, or stored by the client and explicitly attached to requests in an `Authorization: Bearer <token>` header. Instead, two other less common ways are to send the token as either a query parameter or as part of the request body.

JSON Web Token

The specific token we are generating in the server is a *JWT (JSON Web Token)*, which is formally defined as [48]:

> a string representing a set of claims as a JSON object that is encoded in a JSON Web Signature or JSON Web Encryption, enabling the claims to be digitally signed or MACed and/or encrypted.

The JWT is then a string made up of a set of encoded values, and it has to follow a specific syntax, `<header>.<payload>.<signature>`, where:

- `<header>` contains metadata about the token type.
- `<payload>` contains details about the permissions granted, where its content is made of both registered and custom key/value pairs.

- `<signature>` contains a digital signature of the two parts above: it can be based on a asymmetric algorithm or on a secret, shared between the signer and the verifier.

Figure 5.1 shows an example of a JWT with the corresponding JSON object it encodes, where the first part in red is the `<header>`, the second part in purple is the `<payload>` and the last part in blue is the `<signature>`.

Figure 5.1: Example of a JSON Web Token.

One of the claims JWTs typically include in the `<payload>`, although not present in this example, is the expiration time `<exp>`, which contains the Epoch-time after which the JWT expires: since stealing a JWT would allow an attacker to impersonate a user, it is important to reduce this risk by having that a token is not valid forever, once issued, but it only has a validity for a limited amount of time.

JWT in Flask-JWT-Extended

Flask does not natively support the creation and management of JWTs to handle authentication and authorization, which is why we utilized the *Flask-JWT-Extended* library. This library is particularly useful since, besides the creation and verification of JWTs described so far, it also provides [49]: adding custom claims to JWTs, which can then be subject to a custom validation, automatic user loading, refresh tokens, token revoking and blocklisting, and finally CSRF (Cross-Site Request Forgery) protection. Flask-JWT-Extended integrates smoothly with flask-smorest, since it is possible to add authentication and authorization to any route by simply adding one decorator to such function. For example, the native `@jwt_required()` will block requests without a valid JWT present, returning the corresponding error in the HTTP status code and response. We can also define our own custom decorators, so that we can authorize, for example, access to a specific route only if the currently logged in user is of a specific custom role. An example of such custom decorator is shown in Listing 5.9, where the route to remove a privacy notice from a specific application is accessible only to users who are either Data Controllers or Data Protection Officers, as required by `@dc_or_dpo_required()`, which we defined as in Listing 5.10.

```
@privacy_notice_bp.route("/managed-application/<application_uuid>", methods=["DELETE"])
@privacy_notice_bp.arguments(ApplicationUuidReqSchema, location="path", as_kwargs=True)
@dc_or_dpo_required()
def remove_privacy_notice_by_managed_application_uuid(
    application_uuid: UUID, current_user
):
    PrivacyNoticeService.remove_privacy_notice_by_dc_or_dpo_id_and_application_uuid(
        current_user.id, application_uuid
    )
    return "", 204
```

Listing 5.9: Example of usage of a Flask-JWT-Extended decorator.

```
def dc_or_dpo_required():
    def wrapper(fn):
        @wraps(fn)
        def decorator(*args, **kwargs):
            user = get_current_user_from_claims()
            if user.role in (
                UserRole.DATA_CONTROLLER,
                UserRole.DATA_PROTECTION_OFFICER,
            ):
                return fn(*args, **kwargs, current_user=user)
            raise WrongUserRoleException("User is not a Data Controller or DPO!")
        return decorator
    return wrapper
```

Listing 5.10: Example of implementation of a Flask-JWT-Extended decorator.

Flask-JWT-Extended allows to configure in which way tokens are exchanged between client and server: in the example of Listing 5.10, and in other custom decorators as well, we are extracting a JWT sent in a cookie, since we have set our backend to accept JWTs in cookies with the `JWT_TOKEN_LOCATION` configuration. Other than the location, Flask-JWT-Extended provides also different configuration settings to ensure security when using cookies, like `JWT_COOKIE_CSRF_PROTECT`, `JWT_COOKIE_SECURE`, and `JWT_COOKIE_SAMESITE`.

Using a dedicated library for JWT creation and management helps to enforce security, since we are using functions that allow us to follow standardized best practices, which is safer than creating our own authentication and authorization system from the ground up. Moreover, defining custom decorators in a dedicated part of the project and then using them in the same way across all routes that require them helps to improve the maintainability and scalability of our backend.

5.2 End-to-end Testing with Postman

The fact that most of the endpoints of our RESTful API backend receive user input and perform actions as a function of it requires us to ensure that the backend is able to handle valid inputs, invalid inputs, and edge cases, responding correctly without causing service interruptions: no matter what is sent from the client as a request, the service should be able to respond in a graceful manner by taking care of any possible runtime fault [50]. This requirement of verifying software correctness and reliability leads to the concept of automated software testing, a core aspect of software engineering which Dustin et al. formally defined as [51]:

> The management and performance of test activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool.

In the context of a RESTful API backend there exist several kind of automated tests that can be performed, such as *unit*, *integration*, and *end-to-end* testing. Given the scope and time constraints of the project, the testing effort was focused on end-to-end testing, as it best supports the validation of complete API workflows and interactions between the different layers of the system.

Performing end-to-end testing means to systematically invoke API endpoints using realistic input scenarios that reproduce typical user interactions, both for correct cases and for cases that are supposed to give an error. After calling each endpoint, the response is analyzed to compare the actually received result with the result that was expected to be received, for that specific endpoint with that specific input. If the received result equals the expected result, then the test is said to pass, otherwise it is said to fail. The percentage of passing tests over the total amount of tests performed is then a good indicator for the correctness of the backend system. More formally, Leotta et al. define end-to-end testing as [52]:

> A type of black box testing based on the concept of test scenario, that is a sequence of steps/actions performed on the web application (*e.g.*, insert username, insert password, click the login button). One or more test cases can be derived from a single test scenario by specifying the actual data to use in each step (*e.g.* username=John.Doe) and the expected results (*i.e.* defining the assertions). The execution of each test case can be automated by implementing a test script following any of the existing approaches.

The goal of end-to-end web testing is exercising the web application under test as a whole to detect as many failures as possible, where a failure can be considered as a deviation from the expected behavior [52]. As the name suggests, this type of testing is particularly useful for a layered architecture like ours, since sending a

specific HTTP request to analyze its response means that, in most cases, data has gone through all our architectural layers, and a successful test would mean that all layers have worked as expected.

Postman In order to automate the end-to-end testing of our backend, we used *Postman*, an API development and testing platform that provides an user interface to send HTTP requests, analyze responses, and automate tests. Within Postman it is possible to save a set of HTTP requests, grouped by functionality, as a *collection*, which is then exportable as a single JSON file, so that it can be easily uploaded to a remote repository as any other file of the project. Each request can be given any input, such as a request body, exactly as it would be given from a frontend that calls such endpoint. Then testing can be performed on the response by writing custom JavaScript code that executes automatically upon the reception of such response. Postman provides functions to test for the response status code, message, and response body, which can be tested against a desired structure and/or content. Listing 5.11 shows an example of a Postman script to test the response obtained from one of our endpoints: it checks that the response status code is `200`, *i.e.* a successful case; then it checks that the response body has the expected structure, an array of exactly two objects; finally, each of these objects is expected to have all the specified keys, and, in particular, key `role` is expected to always have value `"DATA_SUBJECT"`.

```
1  pm.test("Response status code is 200", function () {
2      pm.expect(pm.response.code).to.eql(200);
3  });
4
5  pm.test("Response is an array of two objects", function () {
6      const responseBody = pm.response.json();
7      pm.expect(responseBody).to.be.an('array').that.has.lengthOf(2)
       ;
8      responseBody.forEach(function(user) {
9          pm.expect(user).to.have.all.keys("uuid", "firstname", "
       lastname", "email", "role", "profile_picture");
10         pm.expect(user).to.not.have.property("password_hash");
11         pm.expect(user.role).to.eql("DATA_SUBJECT");
12     });
13 });
```

Listing 5.11: Example of a Postman end-to-end test.

A key feature that further improves automation is the fact that a collection of endpoints can be set to run in a specific sequence, so as to simulate an entire user's flow of requests that would naturally occur during his usage of the system. The core functionality behind this is that, upon reception of a response from a first endpoint, any value obtained in the response body can be saved to a Postman environment variable, and then, right before calling the following endpoint, this variable can be read to be set as a specific input of this second endpoint. More generally, this allows to have a sequence of requests chained one after the other in which the result of a request is used to compose the input of the following one. Listing 5.12 and Listing 5.13 show an example of usage of a Postman environment variable within two consecutive endpoints. In particular, Listing 5.12 is executed as the post-response script of the first of the two endpoints, and it writes the value of the UUID of a certain user received in the response body to the environment variable `"data_controller2_uuid"`. Then, Listing 5.13 executes after that, as the pre-request script of the second of the two endpoints, and it reads the `"data_controller2_uuid"` environment variable, in order to assign its value to the `user_uuid` path variable of the second request.

```
pm.test("Response is an array of three objects", function () {
    const responseBody = pm.response.json();
    pm.expect(responseBody).to.be.an('array').that.has.lengthOf(3);
    responseBody.forEach(function(user) {
        pm.expect(user).to.have.all.keys("uuid", "firstname", "lastname", "email", "role", "profile_picture");
        pm.expect(user).to.not.have.property("password_hash");
        pm.expect(user.email).to.not.equal("email2@example.com");
        if (user.email === "email5@example.com") {
            pm.environment.set('data_controller2_uuid', user.uuid);
        }
    });
});
```

Listing 5.12: Example of setting a Postman environment variable.

```
const userUuid = pm.environment.get("data_controller2_uuid");
if (userUuid) {
    // if user_uuid is available, set the request URL with the user_uuid path variable
    pm.request.url = pm.request.url.toString().replace(":user_uuid", userUuid);
};
```

Listing 5.13: Example of accessing a Postman environment variable.

An important use case for Postman's environment variables is that values returned within cookies in response headers, such as authentication or CSRF tokens, can be stored and reused in subsequent requests. In this way, the CSRF token obtained from the login endpoint can be automatically inserted into the `X-CSRF-TOKEN` header of following requests, while the JWT used for authentication remains stored in a cookie that Postman automatically includes in each request.

Using the features described so far, we were able to build a Postman collection made up of 303 tests that thoroughly verifies the functionalities of our endpoints, as in Figure 5.2, where the results of a complete run show that all of the tests in the collection ran without any failures.

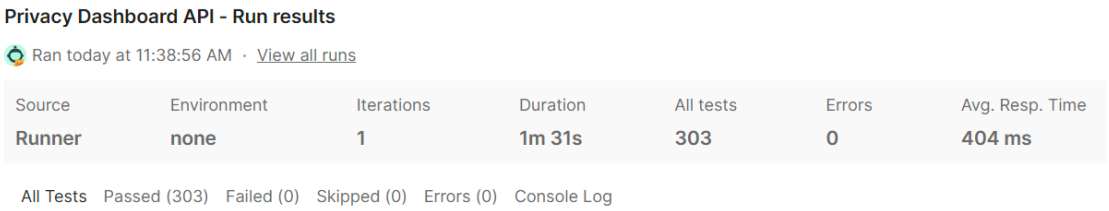


Figure 5.2: Results of a run of our Postman collection.

The adoption of Postman collections for end-to-end testing provided a practical and reproducible way to validate the behavior of the entire backend system. By automating the execution of HTTP requests, validating responses through test scripts, and chaining requests with environment variables, Postman enables the simulation of realistic user workflows across multiple endpoints. This approach is particularly valuable for the maintainability of RESTful architectures, where the correctness of the system depends on the coordinated interaction of multiple layers.

5.3 Frontend Implementation

5.3.1 Privacy Notice with ReactQuill

The Privacy Notice is a textual document that is supposed to potentially contain not only plain text, but also formatted text, such as bold, italic, and underline, and more complex elements, such as lists, external links and headings. This functional requirement made the native HTML `<textarea>` element insufficient, as it only provides a mean to insert and edit plain text. What is instead needed is a rich text editor whose produced output is not just plain text, but rather HTML code: HTML is a markup language that encodes the logical and visual structure of the document through tags such as `<p>`, `<strong>`, `<ul>`, and `<li>`; when rendered by the browser, these tags are interpreted and transformed into the formatted document displayed to the user. HTML is then suitable to handle formatted text because it can store any formatting rule the user desires, regardless of what browser and device he is using to view such content, thus greatly increasing portability [53]. However, directly editing HTML code may seem complex and cumbersome, and that is why the rich text editors used for this purpose are denoted as *WYSIWYG (What You See Is What You Get)*, which are defined as editors that allow users to create content without HTML code knowledge and show to these users exactly how the content should appear on screen [54].

The editor used in our React client to create or update a Privacy Notice is provided by the *ReactQuill* library, which is a React component wrapper for the Quill WYSIWYG rich text editor. Quill internally manages content as a structured document model and exposes it as HTML for storage or transmission, so the conversion from what the user inputs and sees as formatted text to the corresponding HTML code is performed internally, without the developer having to manage it. Once the user submits the Privacy Notice, the client retrieves the obtained HTML string and sends it as-is in the request body of the API endpoint that requests the update of a Privacy Notice for a certain Application. This HTML string is then stored in the database as text preserving its structure, so that later on, upon a request to view such Privacy Notice, that same HTML string will be returned to the client, so that the browser will be able to render it in with the same format as it was created. An example of this in action is given by Figure 5.3 and Figure 5.4: the former shows how the user can see the formatted text directly within the editor; the latter shows the corresponding HTML code that is automatically generated by ReactQuill and sent in the request body of the API endpoint, as the value of the `<content>` key of the JSON object.

One advantage of adopting ReactQuill to represent the Privacy Notice as an HTML string transmitted through the RESTful API as part of a JSON object is that the system preserves formatting information without increasing backend

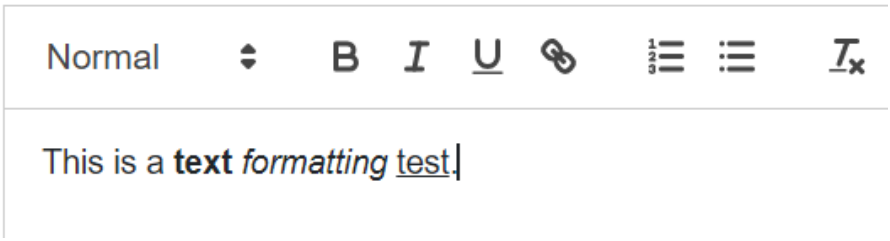


Figure 5.3: Example of writing formatted text in a ReactQuill editor.

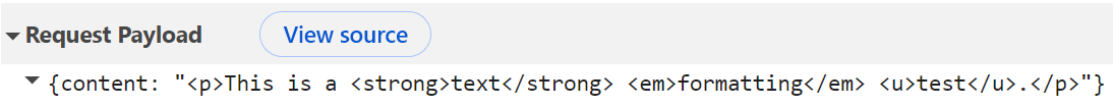


Figure 5.4: Example of HTML string generated by a ReactQuill editor.

complexity. This approach is useful to ensure consistency between content creation and visualization. However, this HTML representation poses an important security threat: an attacker could attempt to embed unsafe elements, such as malicious scripts or HTML attributes, within the content, leading to XSS (Cross-Site Scripting) attacks, which occur when the browser executes an untrusted input as part of the web page, potentially allowing attackers to steal user data or manipulate the user interface. To mitigate this risk, the HTML content generated by the editor is sanitized before being used by the application. In our implementation, this is achieved through the *DOMPurify* library, which removes potentially dangerous tags and attributes while preserving the legitimate formatting of the document, ensuring that only safe HTML content is processed by the client.

5.3.2 Privacy Impact Assessment with AgGridReact

Unlike the Privacy Notice described in the previous subsection, which consists of structured textual content, a PIA is a more complex document, composed of structured records that must be organized, visualized, and edited in tabular form. The one we ask our users to fill out was created by the French Data Protection Authority in 2018, which provided it as a template in a PDF document [55]. The problem raised in this case was to find a reliable way to create a replica of this template in our client, and to have, at the same time, the users filling out these tables, with more advanced features such as adding or removing rows where possible, although without removing or editing any important initial data provided by the template. The native HTML `<table>` element would not allow to implement these structured interactive tables, which is why we utilized the *AgGridReact* library. This library enables implementing React Tables with the addition of many

useful features, such as text formatting, column resizing, filtering, editing, sorting, pagination and more [56]. One key feature AgGridReact was chosen for is the possibility of building tables in the browser populating their rows and columns with some initial data given as a JavaScript object, *i.e.* mapping structured JavaScript objects describing rows and columns to a rendered table component. In this way, when we retrieve the PIA as a JSON object from the backend, we are able to build all the corresponding tables in the browser, each populated with its rows and columns: Figure 5.5 shows an example of a rendered table of the template with four columns and one initial row populated in the left-most cell, and Figure 5.6 shows the corresponding JSON representation received from the backend such table is built from.

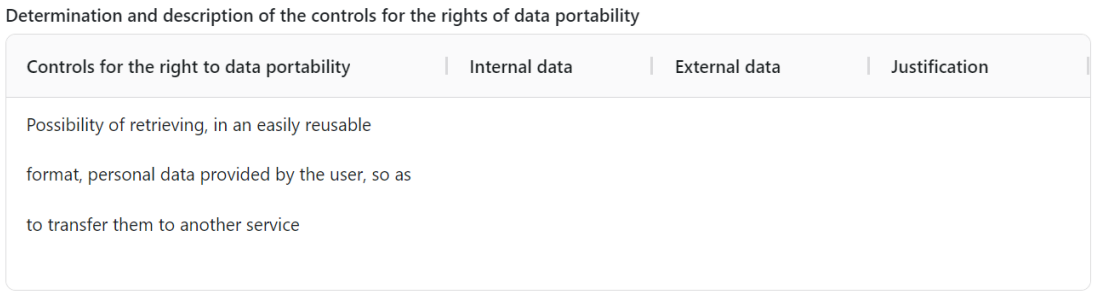


Figure 5.5: Example of an AgGridReact table with initial data in a cell.


```
{
  "table_key": "controls_for_rights_of_data_portability",
  "title": "Determination and description of the controls for the rights of data portability",
  "columns": [
    {
      "key": "controls_right_of_portability",
      "label": "Controls for the right to data portability",
      "type": "text"
    },
    {
      "key": "internal_data",
      "label": "Internal data",
      "type": "text"
    },
    {
      "key": "external_data",
      "label": "External data",
      "type": "text"
    },
    {
      "key": "justification",
      "label": "Justification",
      "type": "text"
    }
  ],
  "initial_rows": [
    {
      "controls_right_of_portability": "Possibility of retrieving, in an easily reusable format, personal data provided by the user, so as to transfer them to another service",
      "internal_data": "",
      "external_data": "",
      "justification": ""
    }
  ],
  "addOrDeleteRowsEnabled": false
},
```


Figure 5.6: Example of JSON of a table to be mapped with AgGridReact.

As mentioned, the template provided an initial row with some initial content for such table, and the user is allowed to fill out only the three empty cells of that row, and cannot add or delete new rows for that table, as stated by the `addOrDeleteRowsEnabled` flag set to `false`. However, there are tables in which the user is free to add or delete rows, using custom buttons that enable such functionalities provided by AgGridReact, since such tables do not contain any initial rows: Figure 5.7 shows an example of a rendered table of the template with two columns and no initial rows, with a "+" button that inserts new rows for the user to fill out, and Figure 5.8 shows the corresponding JSON representation received from the backend such table is built from. We can notice in the JSON object that the `addOrDeleteRowsEnabled` flag is now set to `true`, and that `initial_rows` is now an empty array.

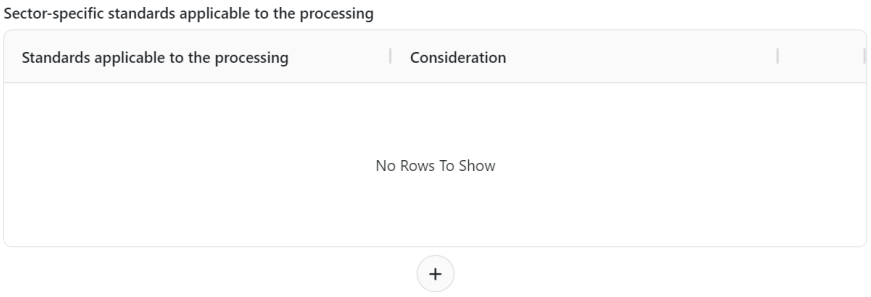


Figure 5.7: Example of an AgGridReact table without any initial rows.

```
{
  "table_key": "sector_standards",
  "title": "Sector-specific standards applicable to the processing",
  "columns": [
    {
      "key": "standards",
      "label": "Standards applicable to the processing",
      "type": "text"
    },
    {
      "key": "consideration",
      "label": "Consideration",
      "type": "text"
    }
  ],
  "initial_rows": [],
  "addOrDeleteRowsEnabled": true
}
```

Figure 5.8: Example of JSON of an empty table to be mapped with AgGridReact.

Another useful feature provided by AgGridReact is that table cells can contain not only plain text, but more advanced input elements as well, such as images and buttons. Figure 5.9 shows one of our tables where one of its columns is of type *select*, meaning that the user will not type text in the corresponding cells, but will open a `select` to choose one among some predefined options.

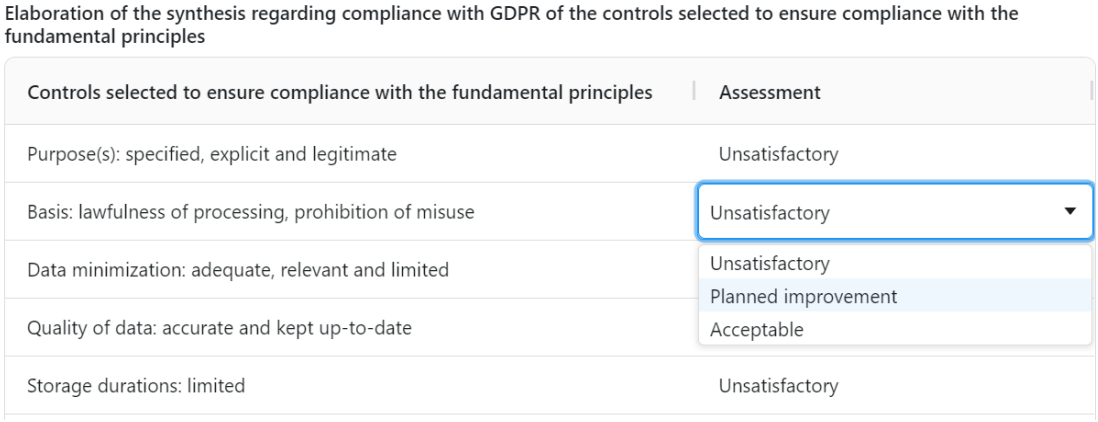


Figure 5.9: Example of an AgGridReact table with a column of type select.

We previously mentioned that we start building the PIA by mapping the received JSON of the template into the corresponding AgGridReact tables; a similar strategy is implemented for the opposite flow, *i.e.* for retrieving the user's input and sending it to the backend for a PIA submission. AgGridReact provides a way to easily access the updated rows for each table, so that we can construct a JavaScript object that follows the same structure of the template JSON, with the only difference that the `initial_rows` array of each table now contains the rows as updated by the user. Then this newly obtained object will be sent within the JSON request body of the dedicated endpoint. Validation will be performed on it by the backend and, in case of success, it will be stored as-is in the dedicated cell, aligning with the fact that PostgreSQL allows to store JSON objects in columns, as implemented in SQLAlchemy with the following line:

```
privacy_impact_assessment:Mapped[dict|None] = mapped_column(JSONB, default=None)
```

In this way, each PIA is stored and sent back to the client in the same way as we did for the initial template, thus improving consistency and maintainability and allowing a straight-forward frontend implementation for visualization and update of any previously submitted PIAs.

5.3.3 Document Downloading with react-pdf-html

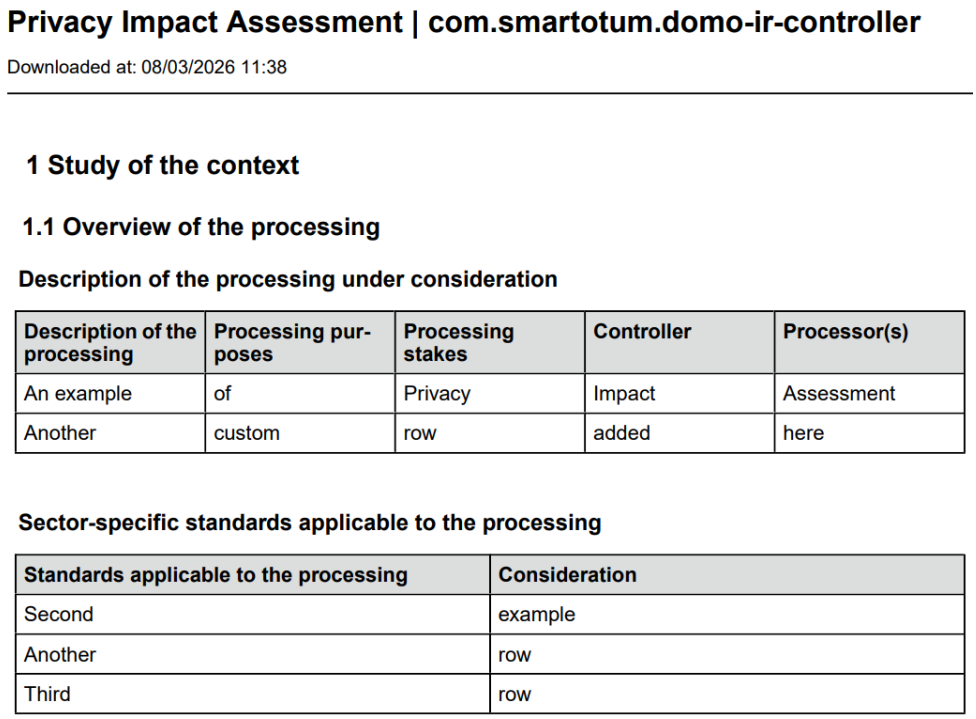
As discussed in previous sections, our web application features dynamic visualization, submission, and edit of documents related to GDPR management: Privacy Notice, PIA and Questionnaire. It is however important that these documents would be made available for offline access too, in case, for example, of temporary unavailability of the service. For this reason, our client provides the possibility of downloading those three types of documents locally on the user's device, with the click of a dedicated button. In order to maximize consistency and portability of the downloaded files, we chose to have them as PDF (Portable Document Format), since it is a file format used to represent a document in a manner independent of the application software, hardware, and operating system used to create it [57].

React does not provide native mechanisms for generating PDF documents, so we had to adopt the *react-pdf-html* library, which works within the *React-pdf* library. The React-pdf library allows to dynamically generate PDF documents using React components, such as `<Document>`, `<Page>`, and `<Text>`, which are rendered into a downloadable PDF file by the library's rendering engine. However, among these components, there is not one that directly supports rendering standard HTML code, which means that existing HTML templates cannot be used directly within React-pdf and would need to be rewritten using other React-pdf components. This limitation is solved by the react-pdf-html library, which provides a React component, `<Html>`, that is able to parse the given HTML string into a structured node tree, process styling rules, and map each HTML element to the corresponding React-pdf component. Consequently, the developer can render a standard HTML string into a PDF file with one line of code, such as:

```
<Html stylesheet={htmlStyles}>{htmlContent}</Html>
```

where `htmlStyles` is a variable containing a set of custom styling rules to be applied to the PDF content, whereas `htmlContent` contains the desired HTML string to convert, which is passed inside the react-pdf-html `<Html>` component, that in turn is used within the React-pdf `PDFDownloadLink` component, which renders the document on demand and exposes it as a downloadable file to the browser. This approach perfectly aligns with our management of the Privacy Notice: as we saw in 5.3.1, our Privacy Notices are stored in the database and returned to the client as HTML strings, so that the client can render them in the browser's web page. As a result, converting a Privacy Notice to a PDF file is as simple as assigning this returned HTML string to the `htmlContent` variable, without any further processing needed. An intermediate transformation step is instead needed for the PIA and the Questionnaire: the backend returns them as regular JSON objects, so we had to implement custom functions that would take such objects as input and transform

them in the corresponding HTML code we want to render in the PDF file. This returned HTML string would then be assigned to the `htmlContent` variable, in the same way as we did for the Privacy Notice. Figure 5.10 shows an example of the beginning of the PDF file generated for a submitted PIA. Prior to the actual content, the system automatically writes the type of document, the name of the application it was submitted for, and the date and time the file was downloaded.



Privacy Impact Assessment | com.smartotum.domo-ir-controller

Downloaded at: 08/03/2026 11:38

1 Study of the context

1.1 Overview of the processing

Description of the processing under consideration

Description of the processing	Processing purposes	Processing stakes	Controller	Processor(s)
An example	of	Privacy	Impact	Assessment
Another	custom	row	added	here

Sector-specific standards applicable to the processing

Standards applicable to the processing	Consideration
Second	example
Another	row
Third	row

Figure 5.10: Example of a PDF generated for a Privacy Impact Assessment.

This implementation approach, that consisted into having the `<Html>` component as an independent component that can be reused across the three different types of GDPR documents, ensured separation of concerns and modularity, since the logic for generating the PDF file is implemented only once in a separate file, and in turn improved the overall maintainability and scalability of the project. However, this client-side generation has the drawbacks of relying on the browser's resources, so it may perform poorly on slower user devices, and of supporting a limited set of HTML elements and styling rules, so for bigger and more complex documents a more robust server-side PDF generation may be preferred.

Chapter 6

Conclusion

6.1 Results Achieved

As discussed in Chapter 3, this project work started from the analysis of the main limitations of the original Privacy Dashboard, and aimed at transforming it into a more complete, modern, and easy-to-use platform for privacy management in Smart Homes environments.

The first major result achieved by our project was the integration with Smartotum, which made it possible to associate the Data Subject with a real Smart Home environment and with the applications actually installed in it, as discussed in Section 3.2.1.

A second key result was the adoption of the PTP (Policy Translation Point) as implemented in the SIFIS-Home project [58], that was not only integrated, but also substantially redesigned and re-implemented to operate within our backend architecture. Together, these two components are the foundation of the Policy Translation page, discussed in Section 3.2.3, which allows users to define high-level privacy policies, such as not allowing video recording in a specific room during a given time window. These policies are then translated into low-level policies expressed in the XACML (eXtensible Access Control Markup Language) formalism and finally converted into device-level rules enforced in Smartotum.

Another important result is that privacy rules are not only manually defined by the Data Subject, but can also be enforced automatically on the basis of the consents granted or withdrawn for the different applications installed in the user's Smart Home, as discussed in Section 3.2.2.

Regarding my personal contribution to the project, the integration with Smartotum in our backend is supported by the design and implementation of an ORM in SQLAlchemy, as illustrated in 5.1.1, which is able to store, using PostgreSQL, any information we are retrieving from the Smartotum system, starting from the

Smart Home itself, which we now store together with its relationships with users and applications. My work then continued in the backend with a clear separation of the three layers defined in 4.3.1, with a particular focus on the controller layer adopting flask-smorest and marshmallow, which allowed to build a reliable and scalable RESTful API, as explained in 5.1.2. Moreover, the implementation of Flask-JWT-Extended, as discussed in 5.1.3, allowed to add a layer of security by adding RBAC (Role-Based Access Control) to our endpoints.

Moving onto the frontend, the main result achieved by my contribution to the project was the addition of a page that provides the functionalities of creation, update, visualization, and download of the PIA (Privacy Impact Assessment), which is the GDPR document that was not supported in the legacy version of Privacy Dashboard and is now implemented as seen in 5.3.2. I then focused on aligning the pages for the management of Privacy Notices and Questionnaires as well, so that the resulting frontend has now a more coherent user experience and more uniform appearance regarding all of the GDPR documents we support. In particular, the implementation of a WYSIWYG editor enables users to write more advanced Privacy Notices with new text formatting, as shown in 5.3.1. Finally, the new messages page unifies the contacts and chat pages into one, thus providing a more modern-looking and more easy-to-use user interface.

6.2 Limitations

Despite the results achieved, the new Privacy Dashboard still presents some limitations.

While the architecture was designed to remain open to future integrations with more home automation systems the current implementation has only been tested with Smartotum. For this reason, further work would be required to assess its applicability to other smart home systems. Related to the Smartotum login architecture was the decision to let Smartotum credentials transit through the backend during authentication, so this solution made it possible to avoid storing credentials or tokens in the backend at the cost of introducing a trust assumption on the server.

A second limitation concerns the synchronization model that relies on synchronization operations that refresh the local snapshot. This choice keeps the platform responsive and allows it to remain usable even when the Smart Home DHT is temporarily unavailable, but it also means that the stored information may not always reflect the most recent state. Also, when the service is unavailable, changes such as new device rules are denied instead of being stored and re-tried later.

A third limitation concerns notification updates that are not delivered in real time through a push mechanism by the backend but are retrieved by the frontend

when convenient. Similarly, the messaging functionality relies on HTTP requests, which does not support real time communication as, for example, a WebSocket connection would. Another limitation regarding the messaging system is that we currently do not support the upload of attachments to messages, so an exchange of only plain-text messages is implemented.

A limitation regarding the new management of PIA documents is that, while it is possible to update an existing PIA for an application, it is not possible to revert to a previously submitted one, as our system does not currently support PIA versioning; any update of the PIA of an application results in an overwrite of the corresponding existing PIA with the new one.

A final limitation is that the functionality of personal data export as a downloadable file is not yet available as a feature that is supposed to be used whenever a Data Subject makes a right request to exercise his right of access or of portability. More generally, our software currently provides limited functionality for what regards the measures that should effectively take place whenever a Data Controller or Data Protection Officer handles a right request exercised by a Data Subject, besides our current features of changing the request status and providing an optional textual response.

6.3 Future Work

Building on the limitations presented above, several natural directions for future work emerge, especially with respect to the integration with Smartotum:

- Extending the current architecture to support additional home automation systems;
- Defining, together with Smartotum, an improved authentication process that avoids passing Data Subject credentials through the backend;
- Introducing a synchronization mechanism that is also triggered by smart home events, in order to align the Privacy Dashboard in real time with the actual state of the smart home;
- Adding a mechanism to save failed updates and resend them when Smartotum becomes available again.

The notification and the messaging systems could be extended so that updates can be delivered to users in real time, for example, by using WebSockets or Server-Sent Events.

The Marketplace concept can also be further developed by supporting the complete flow in which a Data Subject selects an application from the Marketplace

and the dashboard manages its installation into the Smart Home. This extension would require both dedicated support on Smartotum side and a proper user interface for Data Subjects to browse and install applications.

Regarding the support for file uploading, a future development can be to support file attachments to direct messages as previously mentioned, and to also enable equivalent file attachments to Privacy Notices and an optional upload of a profile picture, which could be displayed, for example, as the profile picture of contacts in the messages page.

Finally, a further development would be to implement all the missing functionalities that would be required to occur when handling a right request.

With these new developments in place, together with a more complete testing suite including unit and integration tests, Privacy Dashboard could evolve from its current demonstration environment into a production-ready real world platform.

Bibliography

[1] *SIFIS-Home Project website*. SIFIS-Home. URL: https://www.sifis-home.eu/#About (visited on 02/28/2026) (cit. on pp. 1, 8).

[2] SIFIS-Home Project. *Privacy Dashboard*. GitHub repository, accessed: 2026-03-16. 2024. URL: https://github.com/sifis-home/privacydashboard (cit. on p. 1).

[3] SIFIS-Home Project. *Policy Translation Point*. GitHub repository, accessed: 2026-03-16. 2024. URL: https://github.com/sifis-home/policy-translation-point (cit. on p. 2).

[4] Luca Ardito. *Privacy Dashboard*. GitLab repository, accessed: 2026-03-15. 2025. URL: https://git-softeng.polito.it/d023270/privacy-dashboard (cit. on p. 2).

[5] Serap Türkyılmaz and Erkut Altindag. «Analysis of smart home systems in the context of the internet of things in terms of consumer experience». In: *International Review of Management and Marketing* 12.1 (2022), p. 19 (cit. on p. 5).

[6] Olivia Haring, Sylvia Azumah, and Nelly Elsayed. «A Review of Network Evolution towards a Smart Connected World». In: *International Journal of Computer Applications* 183 (May 2021), pp. 1–8. DOI: 10.5120/ijca2021921311 (cit. on p. 6).

[7] Ibrahim Mashal, Ahmed Shuhaiber, et al. «User acceptance and adoption of smart homes: A decade long systematic literature review.» In: *International Journal of Data & Network Science* 7.2 (2023) (cit. on p. 6).

[8] I. Lella, C. Ciobanu, E. Tsekmezoglou, M. Theocharidou, E. Magonara, A. Malatras, and R. Svetozarov Naydenov. *ENISA threat landscape 2023 – July 2022 to June 2023*. Tech. rep. European Union Agency for Cybersecurity (ENISA), 2023. DOI: doi/10.2824/782573 (cit. on p. 7).

[9] Jingjing Ren, Daniel J Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. «Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach». In: *Proceedings of the Internet Measurement Conference*. 2019, pp. 267–279 (cit. on p. 7).

[10] Eric Zeng, Shrirang Mare, and Franziska Roesner. «End user security and privacy concerns with smart homes». In: *thirteenth symposium on usable privacy and security (SOUPS 2017)*. 2017, pp. 65–80 (cit. on p. 8).

[11] *D5.4: Final Version of SIFIS-Home Security Architecture Implementation*. SIFIS-Home. June 30, 2023. URL: https://www.sifis-home.eu/wp-content/uploads/2023/07/D5.4-Final-Version-of-SIFIS-Home-Security-Architecture-Implementation.pdf (visited on 02/28/2026) (cit. on p. 8).

[12] *La domotica del futuro passa da Smartotum, spin-off del Politecnico*. Politecnico di Torino. Mar. 27, 2024. URL: https://www.polito.it/ateneo/comunicazione-e-ufficio-stampa/poliflash/la-domotica-del-futuro-passa-da-smartotum-spin-off-del (visited on 03/01/2026) (cit. on p. 8).

[13] European Parliament and Council of the European Union. *Article 4: Definitions*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-4-gdpr/ (cit. on pp. 9, 10).

[14] European Parliament and Council of the European Union. *Article 37: Designation of the Data Protection Officer*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-37-gdpr/ (cit. on p. 9).

[15] European Parliament and Council of the European Union. *Article 39: Tasks of the Data Protection Officer*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-39-gdpr/ (cit. on p. 9).

[16] European Parliament and Council of the European Union. *Article 5: Principles relating to processing of personal data*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-5-gdpr/ (cit. on p. 10).

[17] European Parliament and Council of the European Union. *Article 15: Right of access by the data subject*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-15-gdpr/ (cit. on p. 10).

[18] European Parliament and Council of the European Union. *Article 16: Right to rectification*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-16-gdpr/ (cit. on p. 10).

[19] European Parliament and Council of the European Union. *Article 17: Right to erasure ('right to be forgotten')*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-17-gdpr/ (cit. on p. 11).

[20] European Parliament and Council of the European Union. *Article 18: Right to restriction of processing*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-18-gdpr/ (cit. on p. 11).

[21] European Parliament and Council of the European Union. *Article 20: Right to data portability*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-20-gdpr/ (cit. on p. 11).

[22] European Parliament and Council of the European Union. *Article 21: Right to object*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-21-gdpr/ (cit. on p. 11).

[23] European Parliament and Council of the European Union. *Article 25: Data protection by design and by default*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-25-gdpr/ (cit. on p. 11).

[24] European Parliament and Council of the European Union. *Article 12: Transparent information, communication and modalities for the exercise of the rights of the data subject*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-12-gdpr/ (cit. on p. 11).

[25] European Parliament and Council of the European Union. *Article 35: Data Protection Impact Assessment*. Regulation (EU) 2016/679 (General Data Protection Regulation). Accessed via GDPR-Info. 2016. URL: https://gdpr-info.eu/art-35-gdpr/ (cit. on p. 12).

[26] Filippo Peron. «Privacy dashboard, sviluppo di una Web App per la gestione del GDPR = Privacy dashboard, the development of a Web App for GDPR management.» MA thesis. Turin, Italy: Politecnico di Torino, 2023. URL: https://webthesis.biblio.polito.it/26833/1/tesi.pdf (cit. on pp. 12, 16).

[27] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000 (cit. on pp. 45, 46, 53).

[28] R Fielding, M Nottingham, and J Reschke. *RFC 9110 HTTP Semantics*. 2022 (cit. on p. 46).

[29] Edgar F Codd. «A relational model of data for large shared data banks». In: *Communications of the ACM* 13.6 (1970), pp. 377–387 (cit. on pp. 48, 49).

[30] Thomas M Connolly and Carolyn E Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005 (cit. on p. 50).

[31] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (cit. on p. 50).

[32] Michael Stonebraker and Lawrence A Rowe. «The design of Postgres». In: *ACM Sigmod Record* 15.2 (1986), pp. 340–355 (cit. on p. 51).

[33] Stack Overflow. *Stack Overflow Developer Survey 2025: Most Popular Technologies - Database*. Stack Overflow Developer Survey 2025. Accessed via Stack Overflow Survey. 2025. URL: `https://survey.stackoverflow.co/2025/technology#most-popular-technologies-database` (cit. on p. 51).

[34] Lloyd Shirley Jerry Welch James McDonald. *FROM MONOLITH TO MICROSERVICES: A SPRING BOOT–DRIVEN APPROACH TO MODULAR BACK-END SYSTEMS* (cit. on p. 51).

[35] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. «Foundations of JSON schema». In: *Proceedings of the 25th international conference on World Wide Web*. 2016, pp. 263–273 (cit. on p. 53).

[36] Stack Overflow. *Stack Overflow Developer Survey 2025: Most Popular Technologies – Web Frameworks and Technologies*. Stack Overflow Developer Survey 2025. Accessed via Stack Overflow Survey. 2025. URL: `https://survey.stackoverflow.co/2025/technology#most-popular-technologies-webframe` (cit. on pp. 53, 55).

[37] Miguel Grinberg. *Flask web development*. " O'Reilly Media, Inc.", 2018 (cit. on p. 54).

[38] Ali Mesbah and Arie Van Deursen. «Migrating multi-page web applications to single-page Ajax interfaces». In: *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE. 2007, pp. 181–190 (cit. on p. 54).

[39] Pete Hunt. *Why did we build React?* Accessed: 2026-03-01. June 2013. URL: https://legacy.reactjs.org/blog/2013/06/05/why-react.html (cit. on p. 55).

[40] Sanchit Aggarwal et al. «Modern web-development using reactjs». In: *International Journal of Recent Research Aspects* 5.1 (2018), pp. 133–137 (cit. on p. 56).

[41] Scott W Ambler. «Mapping objects to relational databases». In: *On the World Wide Web: http://www. AmbySoft. com* (2000) (cit. on p. 57).

[42] Wolfgang Keller. «Persistence Options for Object-Oriented Programs». In: *Proceedings of OOP2004* (2004) (cit. on p. 57).

[43] Martin Lorenz, Jan-Peer Rudolph, Günter Hesse, Matthias Uflacker, and Hasso Plattner. *Object-relational mapping revisited-a quantitative study on the impact of database technology on O/R mapping strategies*. 2017 (cit. on p. 57).

[44] SQLAlchemy Authors. *SQLAlchemy Unified Tutorial*. SQLAlchemy 2.0 Documentation. Accessed via official SQLAlchemy documentation. 2026. URL: https://docs.sqlalchemy.org/en/20/tutorial/index.html (cit. on p. 58).

[45] marshmallow Developers. *marshmallow: Object serialization and deserialization library*. marshmallow Documentation. Accessed via official marshmallow documentation. 2026. URL: https://marshmallow.readthedocs.io/en/latest/ (cit. on p. 61).

[46] Michal Trnka, Tomas Cerny, and Nathaniel Stickney. «Survey of Authentication and Authorization for the Internet of Things». In: *Security and Communication Networks* 2018.1 (2018), p. 4351603 (cit. on p. 64).

[47] Hokeun Kim and Edward A Lee. «Authentication and Authorization for the Internet of Things». In: *IT Professional* 19.5 (2017), pp. 27–33 (cit. on p. 65).

[48] Michael Jones, John Bradley, and Nat Sakimura. *Rfc 7519: Json web token (jwt)*. 2015 (cit. on p. 65).

[49] Lily Acadia Gilbert. *Flask-JWT-Extended: Extended JWT integration with Flask*. Flask-JWT-Extended PyPI Project. Accessed via PyPI package index. 2024. URL: https://pypi.org/project/Flask-JWT-Extended/ (cit. on p. 66).

[50] Adeel Ehsan, Mohammed Ahmad ME Abuhaliqa, Cagatay Catal, and Deepti Mishra. «RESTful API testing methodologies: Rationale, challenges, and solution directions». In: *Applied Sciences* 12.9 (2022), p. 4369 (cit. on p. 68).

[51] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999 (cit. on p. 68).

[52] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. «Approaches and tools for automated end-to-end web testing». In: *Advances in Computers*. Vol. 101. Elsevier, 2016, pp. 193–237 (cit. on p. 68).

[53] Nick Williams and Tim Wilkinson. «Experiences in Writing a WYSIWYG Editor for HTML». In: *Proceedings of WWW*. Vol. 94. 1994 (cit. on p. 72).

[54] Hedi Carlos Minin, Javier Jiménez Alemán, Carolina Sacramento, and Daniela Gorski Trevisan. «A WYSIWYG editor to support accessible web content production». In: *International Conference on Universal Access in Human-Computer Interaction*. Springer. 2015, pp. 221–230 (cit. on p. 72).

[55] Commission Nationale de l'Informatique et des Libertés (CNIL). *Privacy Impact Assessment (PIA): Templates*. Guidelines / Templates. February 2018 edition. CNIL, Feb. 2018. URL: `https://www.cnil.fr/sites/default/files/atoms/files/cnil-pia-2-en-templates.pdf` (cit. on p. 73).

[56] AG Grid Documentation Authors. *React Data Grid Key Features*. AG Grid React Data Grid Documentation. Accessed via official AG Grid documentation. 2026. URL: `https://www.ag-grid.com/react-data-grid/key-features/` (cit. on p. 74).

[57] Tim Bienz, Richard Cohn, and Calif.) Adobe Systems (Mountain View. *Portable document format reference manual*. Addison-Wesley Reading, MA, USA, 1993 (cit. on p. 77).

[58] SIFIS-Home Project. *Policy Translation Point*. `https://github.com/sifis-home/policy-translation-point`. GitHub repository, accessed: 2026-03-15. 2023 (cit. on p. 79).